

Geavanceerd objectgeoriënteerd programmeren

Cursusdeel 1

Open Universiteit
Faculteit Management, Science & Technology

Cursusteam

dr. ir. H.J.M. Passier, *cursusteamleider en auteur*

dr. ir. S.T.O. Jongmans, *auteur*

dr. ir. A.J.F. Kok, *auteur*

Cursusteam van de oorspronkelijke cursus Objectgeoriënteerd programmeren in Java 2 (T42241)

drs. H.J. Sint, *cursusteamleider en auteur*

drs. A.M.I. Herrewijn-van de Zande, *auteur*

dr. ir. A.J.F. Kok, *auteur*

M. Witsiers-Voglet, *auteur*

drs. J.L.C. Arkenbout, *redacteur*

Programmaleiding

prof. dr. T.E.J. Vos

Geavanceerd objectgeoriënteerd programmeren

Typen en hiërarchieën

Open Universiteit
www.ou.nl



Productie
Open Universiteit

Redactie
Arnold van der Leer

Lay-out
Maria Wienbröcker-Kampermann

Omslag
Team Visuele communicatie, Open Universiteit

Druk- en bindwerk
OCÉ Business Services



Dit materiaal is gelicentieerd onder de
Creative Commons-licentie Naamsvermelding-
NietCommercieel-GeenAfgeleideWerken 4.0.
Zie de licentie voor details:
<https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode>

This content is licensed under the Creative
Commons license Attribution-Noncommercial-
NoDerivs 4.0.
See licence for more details:
<https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode>

Eerste druk: 2017

IB0902_50153_21072017

ISBN 978 94 92231 73 4 (serie)
ISBN 978 94 92231 76 5 (deel 1)

Cursuscode IB0902

Structuur van de cursus Geavanceerd objectgeoriënteerd programmeren

Onderdeel	Blok	Leereenheid	Bladzijde
Cursusboek 1		Introductie tot de cursus	7
	1	1 Objectgeoriënteerd ontwerpen	15
	Typen en hiërarchieën	2 Overerving (1)	55
		3 Overerving (2)	101
		4 Abstracte klassen en interfaces	133
		Inleveropdracht	159
		5 Generieke typen en enumeratietypen	161
		Register	188
Cursusboek 2	2	6 Het opsporen van fouten	
	Exceptions en threads	7 Robuust programmeren	
		8 Threads	
	3	9 Koppeling met databases	
	Gegevensopslag	Register	
Cursusboek 3	4	10 De gebruikersinterface (1)	
	Gebruikers-interfaces	11 De gebruikersinterface (2)	
		12 Het Observerpatroon	
		Bijlage: Syntaxis van Java	
		Eindtoets (zie yOULearn)	
		Register	
Cursussite	http://youlearn.ou.nl		

Introductie tot de cursus

- 1 Plaats en functie van de cursus 7
- 2 Inhoud van de cursus 8
 - 2.1 Voorkennis 8
 - 2.2 Leerdoelen 8
 - 2.3 Opbouw van de cursus 9
 - 2.4 Leermiddelen 9
- 3 Aanwijzingen voor het bestuderen van de cursus 10
- 4 Software 11
- 5 Tentaminering 12
- 6 Oorspronkelijk cursusteam 12

Introductie tot de cursus

Om u wegwijs te maken in de cursus *Geavanceerd objectgeoriënteerd programmeren*, informeren wij u eerst over de bedoeling van de cursus, de opzet van het cursusmateriaal en de manier waarop u de cursus kunt bestuderen. U vindt in deze introductie praktische en studietechnische informatie die u inzicht geeft in de aard en opzet van de cursus en u helpt bij het studeren.

1 Plaats en functie van de cursus

De cursus *Geavanceerd objectgeoriënteerd programmeren* is een cursus van het tweede niveau met een studielast van 5 EC. Het is de tweede cursus in de informaticaopleidingen van de Open Universiteit Nederland die geheel aan programmeren is gewijd en is een direct vervolg op de basiscursus *Objectgeoriënteerd programmeren* (IB1102).

De twee cursussen samen bieden een brede inleiding in objectgeoriënteerd programmeren in het algemeen en in het gebruik van de taal Java in het bijzonder.

Aan het eind van de basiscursus kon u, gebruikmakend van een ontwikkelomgeving, een eenvoudige applicatie construeren die is opgebouwd uit een domeinlaag van enkele klassen en een interface-laag van één klasse, waarbij de taak van de interfacelaag beperkt blijft tot de communicatie met de gebruiker. Voor de constructie van de gebruikersinterface leerde u een visuele editor te gebruiken. U leerde ook waar nodig gebruik te maken van klassen uit de Java API, met de nadruk op veel gebruikte klassen uit `java.util` en `java.lang`. De cursus kende daarmee drie belangrijke lijnen, namelijk objectgeoriënteerd ontwerpen, de taal Java en het gebruik van de Java API.

In deze tweede cursus zetten we de drie genoemde lijnen voort. Het maken van een goed ontwerp is iets wat in de cursus aandacht krijgt. In de eerste twee blokken komen alle taalelementen aan bod die in de basiscursus nog niet zijn behandeld. In het derde en vierde blok besteden we aandacht aan twee belangrijke onderdelen van de API. Het onderwerp van het derde blok is gegevensopslag, waarbij aandacht wordt besteed aan de package `java.sql` voor de koppeling van een Java-programma aan een relationele database. Het onderwerp van het vierde blok is de constructie van gebruikersinterfaces, met aandacht voor de packages `javax.swing`, `java.awt` en `java.awt.geom`.

2 Inhoud van de cursus

2.1 VOORKENNIS

De cursus is een direct vervolg op Objectgeoriënteerd programmeren en Objectgeoriënteerd analyseren en ontwerpen en kan dus pas bestudeerd worden als deze beide cursussen zijn afgerond of als op andere wijze voldoende basiskennis van Java en UML is verworven. Concreet betekent dit dat u

- de basisbegrippen van objectgeoriënteerd programmeren (klasse, instantie, attribuut, methode, de eerste beginselen van overerving) moet begrijpen en moet kunnen toepassen
- een eenvoudige klasse moet kunnen definiëren
- zonder al te veel moeite methoden moet kunnen schrijven, gebruikmakend van de volgende elementaire taalconstructies van Java: toekenningen, expressies, keuzeopdrachten, herhalingsopdrachten, arrays en ArrayLists
- inzicht moet hebben in de manier waarop objecten in het geheugen zijn gerepresenteerd en in de consequenties daarvan (aliasing), ook bij parameteroverdracht
- overweg moet kunnen met de bij deze cursus gebruikte ontwikkelomgeving Eclipse
- de volgende UML-modellen kunt lezen en opstellen: use case model, domeinmodel, system sequence diagram, communicatiediagram en ontwerpklassediagram.

2.2 LEERDOELEN

Na het volgen van deze cursus wordt verwacht dat u

- de syntaxis en semantiek van Java kent
- met name begrip heeft van overerving (inclusief het gebruik van abstracte klassen en interfaces), van exception handling en van threads
- eenvoudige generieke klassen kunt definiëren
- begrip heeft van de wijze waarop in objectgeoriënteerde programma's gebruik gemaakt kan worden van programmeren per contract
- een eenvoudig concurrent programma kunt programmeren
- manieren kent waarop persistentie van gegevens gerealiseerd kan worden en een daarvan ook kunt toepassen, namelijk de koppeling met databases (package java.sql)
- in staat is om met behulp van de package javax.swing zelf een grafische gebruikersinterface te programmeren.

Na het volgen van de cursus bent u in staat om, vanuit een gegeven specificatie, zelfstandig een overeenkomstig objectgeoriënteerd programma te ontwerpen en te implementeren, gebruikmakend van alle elementen van de taal Java. Dat ontwerp kan verscheidene (tot circa tien) eigen klassen bevatten en maakt een zinvol gebruik van behandelde delen van de Java API (de koppeling met een database met behulp van JDBC, het programmeren van een grafische gebruikersinterface gebaseerd op Swing). Het programma is goed ontworpen (opbouw in lagen, klassen met duidelijke, beperkte verantwoordelijkheden, gebruik van packages), de code is helder en begrijpelijk, het programma is systematisch getest en het is gedocumenteerd met behulp van Javadoc.

2.3 OPBOUW VAN DE CURSUS

Blok 1 Typen en hiërarchieën

5 leereenheden
studielast: 35 uur

Verplichte
inleveropdracht

We beginnen de cursus met een *inleidende leereenheid*, waarin de vraag wordt gesteld – en deels beantwoord – wat een goed programma is. In deze leereenheid herhalen we ook delen van UML, een notatie speciaal gericht op het ontwerpen van objectgeoriënteerde programma's. De rest van het eerste blok is gewijd aan het typesysteem van Java. Ten eerste wordt *overerving* behandeld. Dynamische binding van methoden is daarbij een sleutelbegrip. Vervolgens komen abstracte klassen en interfaces aan de orde, begrippen die een belangrijke rol spelen bij het definiëren van zogeheten *ontwerppatronen*: schematische oplossingen voor veel voorkomende problemen. Het blok eindigt met een korte behandeling van generics: de uitbreiding van Java die het gebruik van typeparameters mogelijk maakt.

Na leereenheid 4 wordt u gevraagd om een ontwerp- en programmeeropdracht te maken. De opdracht is verplicht en wordt beoordeeld met een onvoldoende of voldoende. U kunt uw uitwerking op een beperkt aantal momenten inleveren. De opdracht en alle informatie hieromtrent vindt u op de cursussite.

Blok 2 Exceptions en threads

3 leereenheden
studielast: 23 uur

Het tweede blok voltooit de behandeling van de taalconcepten van Java. De eerste twee leereenheden gaan over fouten die kunnen optreden in programma's en hoe daarmee om te gaan. Leereenheid 6 bekijkt welke soorten fouten er zijn, wat er gedaan moet worden om ze op te merken (testen) en hoe hun oorzaak opgespoord kan worden (debuggen). Java biedt een mechanisme om bepaalde soorten fouten af te handelen (exception handling). Dit is het onderwerp van leereenheid 7.

Leereenheid 8 tenslotte biedt een korte inleiding in het programmeren met Threads, dat een vorm van parallelisme mogelijk maakt. We geven aan hoe een programma met meerdere threads gemaakt kan worden, maar laten ook zien dat u zich daarmee op glad ijs begeeft.

Blok 3 Gegevensopslag

1 leereenheid
studielast: 9 uur

In de blokken 3 en 4 ligt de nadruk op delen van de API. Het onderwerp van blok 3 is gegevensopslag. Behandeld wordt de koppeling van een Java-programma met een relationele database via JDBC (Java database connectivity; package java.sql).

Blok 4 Gebruikers- interfaces

3 leereenheden
studielast: 27 uur

Blok 4 behandelt de constructie van gebruikersinterfaces zonder gebruik te maken van een visual editor. De eerste twee leereenheden worden besteed aan het werken met Swing-componenten; ook het event handling mechanisme komt daarbij aan de orde.. De laatste leereenheid beschrijft het Observerpatroon, een veel gebruikt en belangrijk ontwerppatroon dat het mogelijk maakt om de domeinlaag geheel onafhankelijk te houden van de gebruikersinterfaces, ook wanneer het initiatief voor wijzigingen in die interfaces bij de domeinlaag ligt.

2.4 LEERMIDDELEN

Het cursusmateriaal bestaat uit de volgende onderdelen:

- cursusdeel 1, met blok 1
- cursusdeel 2, met blokken 2 en 3
- cursusdeel 3, met blok 4

Cursussite

Daarnaast behoort ook de *cursussite* tot het cursusmateriaal. U vindt deze via <http://youlearn.ou.nl>.

Informatie over
begeleiding en
tentaminering

De cursussite biedt de meest actuele informatie over de cursus. Op de cursussite vindt u onder meer informatie over begeleiding en tentaminering, handleidingen bij de gebruikte software (zie ook paragraaf 4), bouwstenen voor te ontwikkelen Java-programma's, een eindtoets en errata bij de cursus. Ook kunt u via deze site in contact komen met medestudenten en docenten.

3 **Aanwijzingen voor het bestuderen van de cursus**

Leereenheid

Een blok is verdeeld in *leereenheden*. Een leereenheid is een afgerond deel van de stof dat u in een of twee dagdelen kunt bestuderen. Elke leereenheid wordt voorafgegaan door een inhoudsopgave en bestaat verder uit een introductie, een leerkern, een zelftoets en een terugkoppeling. Deze onderdelen van een leereenheid omschrijven we kort.

Introductie

De *introductie* van een leereenheid geeft kort aan wat de inhoud van de leereenheid is en schept een kader waarin deze inhoud geplaatst moet worden. In de introductie vindt u de *leerdoelen*. Die stellen u in staat na te gaan welke kennis, inzichten en vaardigheden u zich eigen moet maken door bestudering van de leereenheid. De introductie wordt afgesloten met studeeraanwijzingen voor de betreffende leereenheid. Daarin wordt in elk geval de studielast vermeld.

Leerdoelen

Leerkern
Opgave
Opdracht

De *leerkern* van een leereenheid bevat de feitelijke leerstof. Op bepaalde plaatsen in de leerkern treft u *opgaven* of *opdrachten* aan. Het verschil tussen een opgave en een opdracht is dat u een opdracht achter uw pc moet uitvoeren, terwijl een opgave gewoon met pen en papier kan.

OPGAVE 0.0

Dit is een voorbeeld van een opgave.

- a Werk elke opgave *tijdens* het bestuderen van de leereenheid uit.
- b Vergelijk uw antwoord direct met het antwoord in de terugkoppeling aan het eind van de leereenheid.

Studeeropdracht

U treft in de leerkern ook ongenummerde vragen aan: de *studeeropdrachten*.

Dit is een voorbeeld van een ongenummerde vraag ofwel studeeropdracht. Wat denkt u dat de functie daarvan is?

Een studeeropdracht wordt altijd in de direct daarop volgende tekst beantwoord. De functie van studeeropdrachten is dat u zelf uw eigen antwoord op de vraag probeert te formuleren, voordat u de betreffende tekst leest. Dat helpt u om aandachtig de denkstappen in de tekst te volgen.



Samenvatting		Meteen na de leerkern staat een <i>samenvatting</i> waarin de belangrijkste zaken uit deze leerkern nog eens worden opgesomd.
Zelftoets		De <i>zelftoets</i> van een leereenheid bestaat uit opgaven die bedoeld zijn om u te helpen beoordelen of u, na bestudering van de leerkern, de leerdoelen inderdaad bereikt hebt. Het is van belang dat u na bestudering van de leerstof de gehele zelftoets nauwgezet uitwerkt.
Terugkoppeling		In de <i>terugkoppeling</i> van de leereenheid zijn de uitwerkingen opgenomen van de opgaven en opdrachten en van de zelftoets. Dit onderdeel van de leereenheid heet niet voor niets terugkoppeling: u wordt in staat gesteld uw antwoord op een opdracht te vergelijken met het gewenste antwoord.
	Belangrijk	Wij raden u ten zeerste af om te snel het antwoord in de terugkoppeling op te zoeken. U leert meer en beter als u eerst zelf een oplossing probeert te vinden en de opgave of opdracht volledig en aandachtig uitwerkt. Meer nog dan bij andere cursussen is dat bij een programmeercursus als deze van belang. <i>Programmeren leert u alleen maar door het zelf te doen!</i> Al mag u op het tentamen het cursusmateriaal gebruiken (zie paragraaf 5), dat is geen reden de opgaven en opdrachten niet te maken.
Margeteksten Kernbegrippen		Het zal u opvallen dat iedere pagina aan de linkerkant twee kolommen heeft: de <i>marge</i> . In de meest linkse kolom treft u kernbegrippen aan. <i>Kernbegrippen</i> zijn, zoals de naam al zegt, centrale begrippen. Deze kolom maakt het gemakkelijk om bepaalde tekstonderdelen snel terug te vinden en schetst tegelijkertijd de hoofdlijn van de leerinhoud.
Studeeraanwijzingen	Dit is een voorbeeld van een studeeraanwijzing.	In de tweede kolom van de marge staan korte <i>studeeraanwijzingen</i> . Dat kunnen toelichtingen op vreemde woorden zijn, aanwijzingen hoe de tekst gelezen moet worden, herhalingen of verwijzingen naar eerdere leereenheden.
Leestekst		Soms is in de leerstof een passage ingevoegd die nader op de stof ingaat of een kanttekening plaatst. Een dergelijke passage behoort niet tot de tentamenstof en is te herkennen aan het kleinere lettertype. Een leestekst wordt afgedrukt in kleine letters. Zo kunt u meteen zien dat het hier een aanvulling betreft die u alleen maar door hoeft te lezen en die niet tot de tentamenstof behoort.

4 Software

Benodigde software	Om de opdrachten in deze cursus te kunnen doen, moet u eerst de benodigde software installeren. Deze software bestaat uit de volgende onderdelen: <ul style="list-style-type: none">– de taal Java– een ontwikkelomgeving voor Java– bouwstenen bij de cursus– een database server (voor leereenheid 9)– een JDBC-driver (voor leereenheid 9).
Installatie- en gebruikershandleidingen op cursussite	Deze onderdelen zijn beschikbaar via de cursussite. Daar vindt u ook de benodigde installatie- en gebruikershandleidingen.

U moet in elk geval de eerste drie onderdelen installeren voor u begint met het bestuderen van de cursus. De laatste twee onderdelen zijn alleen nodig voor leereenheid 9; u kunt die naar keuze nu installeren of wanneer u aan die leereenheid toe bent.

Apparatuur

In de installatiehandleiding van de ontwikkelomgeving zijn ook apparatuurspecificaties opgenomen. In het algemeen gaat de Open Universiteit uit van het gebruik van computers met Windows en zal een computer die niet ouder is dan ongeveer vier jaar voldoende zijn om de software te kunnen draaien. We streven naar het gebruik van een ontwikkelomgeving die ook voor andere besturingssystemen beschikbaar is (bijvoorbeeld Linux en MacOS), zelfs al geven we daarbij geen ondersteuning. Raadpleeg voor details de installatiehandleiding; als u daar nog geen toegang toe heeft kunt u contact opnemen met de examinerator van de cursus.

U bent uiteraard vrij om een andere ontwikkelomgeving te gebruiken dan degene die we meeleveren. De Java-broncode uit de bouwstenen blijft bruikbaar, maar u moet deze dan zelf in een project invoegen. U kunt bij het gebruik van een andere ontwikkelomgeving geen aanspraak maken op ondersteuning bij het gebruik daarvan.

5 Tentaminering

Schriftelijk tentamen

Voor studenten uit de wo-opleiding Informatica wordt de cursus afgesloten met een *schriftelijk tentamen* van drie uur. Het tentamen bestaat geheel uit open vragen. Het is toegestaan om tijdens het tentamen gebruik te maken van het cursusmateriaal. De tentamendata vindt u op de cursussite. Algemene informatie over de gang van zaken bij het tentamen kunt u vinden via de website van de Open Universiteit Nederland: www.ou.nl.

Verplichte inleveropdracht

Na leereenheid 4 wordt u gevraagd om een ontwerp- en programmeeropdracht te maken. De opdracht is verplicht en wordt beoordeeld met een onvoldoende of voldoende. U moet deze opdracht met een voldoende afsluiten om een certificaat van deze cursus te krijgen.

Eindtoets

Bij de cursus hoort een eindtoets die representatief is voor het tentamen. Wij adviseren u nadrukkelijk deze pas te maken als u klaar bent met de tentamenvorbereiding. U vindt de eindtoets op de cursussite.

6 Oorspronkelijk cursusteam

De auteurs van de eerste versies van de cursus zijn, voor zover nog niet genoemd, in alfabetische volgorde:

dr. ir. J.P.H.W. van den Eijnde
dr. P.G. Kluit
C.A. Nolet

Blok 1

Typen en hiërarchieën

Objectgeoriënteerd ontwerpen

Introductie 15

Leerkern 16

- 1 Objectgeoriënteerd ontwerpen 16
 - 1.1 Softwareontwikkeling 16
 - 1.2 Wat is een goed programma? 21
 - 1.3 Objectkeuze 26
- 2 UML-diagrammen 27
 - 2.1 Klassendiagrammen 27
 - 2.2 Objectdiagrammen 33
- 3 Een eenvoudige simulatie van een bank 34
 - 3.1 Productomschrijving en gebruiksmogelijkheden 34
 - 3.2 Opstellen domeinmodel 35
 - 3.3 Opstellen ontwerpmodel 37

Samenvatting 44

Zelftoets 46

Terugkoppeling 47

- 1 Uitwerking van de opgaven 47
- 2 Uitwerking van de zelftoets 51

Objectgeoriënteerd ontwerpen

INTRODUCTIE

In de cursus Objectgeoriënteerd programmeren hebt u kleine, objectgeoriënteerde programma's leren schrijven. U werd daarbij aangemoedigd om bepaalde regels voor goed programmeren in acht te nemen.

U heeft bijvoorbeeld geleerd om de programmacode in de gebruikersinterface zo beperkt mogelijk te houden en al het echte werk over te laten aan instanties van andere klassen. In alle voorbeelden beperkte de taak van de gebruikersinterface zich daardoor tot het verzorgen van de interactie met de gebruiker van de applicatie. Ook hebben we u aangemoedigd om duidelijke namen te kiezen voor alles wat een naam moet krijgen: componenten uit de gebruikersinterface, klassen, attributen, methoden, lokale variabelen en parameters. En we wilden graag dat u de klassen als geheel en iedere methode afzonderlijk van commentaar voorzag.

In Objectgeoriënteerd programmeren is echter niet zo veel aandacht besteed aan de vraag waarom we dat willen. Wat is er tegen een applicatie waarin alles in de interfaceklasse (het frame) zelf gedaan wordt, waarin componenten `button1`, `textfield1` en `textfield2` heten, waarin alle andere attributen, variabelen en methoden zo kort mogelijke namen krijgen (a, b, c) en waarin geen regel commentaar staat? Met andere woorden: hoe ziet een goed programma er eigenlijk uit?

In paragraaf 1 van deze leereenheid gaan we deze vraag onderzoeken en er een gedeeltelijk antwoord op geven. We kijken nadrukkelijk niet alleen naar programma's die één persoon in uren of dagen kan ontwikkelen. We zoeken ook naar criteria voor programma's waar verschillende programmeurs aan werken en waarvan de ontwikkeling weken, maanden of zelfs jaren kost.

We houden ons in deze cursus, meer dan in Objectgeoriënteerd programmeren, bezig met het ontwerp van objectgeoriënteerde programma's. We maken daarbij gebruik van diagramtechnieken uit de unified modeling language (UML) die ook in de cursussen Objectgeoriënteerd programmeren en Objectgeoriënteerd analyseren en ontwerpen aan de orde zijn geweest. Met behulp van klassendiagrammen wordt de structuur van een ontwerp beschreven: de klassen en de samenhang daartussen. Met objectdiagrammen kan de toestand van objecten op een bepaald moment tijdens het verwerken van een programma worden beschreven.

In Objectgeoriënteerd programmeren werd, om de notatie zo dicht mogelijk aan te laten sluiten bij Java, afgeweken van de officiële UML-notatie. In deze cursus houden we ons echter wel aan die officiële notatie, die we in paragraaf 2 introduceren.

In paragraaf 3 gebruiken we deze diagramtechnieken en de criteria die in paragraaf 1 zijn opgesteld, in een ontwerp voor een eenvoudige simulatie van bankverkeer.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u

- kunt aangeven wat bedoeld wordt met softwareontwikkeling in het klein en met softwareontwikkeling in het groot
- de verschillende fasen in de ontwikkeling van een programma kunt aangeven
- eisen kunt opnoemen die aan het ontwerp en de implementatie van een softwaresysteem kunnen worden gesteld
- een eenvoudig klassendiagram kunt lezen en opstellen
- de betekenis kent van de volgende kernbegrippen: voortraject, analyse, gebruiksmogelijkheid, domeinmodel, ontwerp, ontwerpmodel, implementatie, evaluatie, afronding, correctheid, robuustheid, gescheiden verantwoordelijkheden, lokaliteit, koppeling, generalisatie, associatie, rol, multipliciteit, link.

Studeeraanwijzingen

De studielast van deze leereenheid bedraagt circa 7 uur.

LEERKERN

1 Objectgeoriënteerd ontwerpen

1.1 SOFTWAREONTWIKKELING

Software-ontwikkeling in het klein

De cursussen Objectgeoriënteerd programmeren en Geavanceerd objectgeoriënteerd programmeren hebben als doel u te leren *software te ontwikkelen in het klein*. Dat betekent dat deze twee cursussen samen u voldoende basis moeten geven om, eventueel na wat extra oefening, zelfstandig kleine programma's te kunnen ontwikkelen.

Of een programma klein of groot is, meten we niet af aan het aantal regels code in het eindproduct. Bepalend voor de ontwikkeling van een klein programma zijn de volgende kenmerken:

- Het wordt ontwikkeld door één programmeur.
- Het kan worden ontwikkeld in een beperkte tijd, die eerder gemeten zal worden in uren of dagen dan in maanden of jaren.
- Er is meteen al een duidelijke productomschrijving waaruit nauwkeurig valt op te maken wat het programma moet doen.
- Er is geen gebruikersgroep voor wie het programma per se onderhouden moet worden.

Software-ontwikkeling in het groot

Als de ontwikkeling van een programma geen van deze kenmerken heeft, is er sprake van *softwareontwikkeling in het groot*. Een ontwikkeltraject dat aan slechts enkele van de eisen voldoet, valt in een overgangsgebied.

Voorbeeld:
containerverhuur

In de rest van deze paragraaf gebruiken we als voorbeeld van een groot programma regelmatig een informatiesysteem voor een bedrijf dat verschillende typen containers verhuurt. De huurder neemt de containers op de plaats van vertrek in ontvangst en levert ze weer in op de plaats van bestemming; de verhuurder zorgt ervoor dat ze weer bij een volgende huurder terechtkomen. Het bedrijf heeft twintig kantoren in verschillende landen. Het informatiesysteem moet alle containers gaan volgen en de kantoren in staat stellen om snel te zien of aan een verzoek van een huurder voldaan kan worden en welke containers daar dan het best voor gebruikt kunnen worden. Uiteindelijk moet dat tot een efficiënter gebruik van containers leiden.

Deze cursus gaat niet over softwareontwikkeling in het groot; daar zijn andere cursussen voor. Wat we in deze cursus echter wel willen, is u een programmeerstijl bijbrengen die ook bruikbaar is voor het programmeren in het groot. In een cursus over objectgeoriënteerd programmeren ligt dat ook voor de hand, omdat deze programmeerstijl bij uitstek is ontwikkeld met het oog op grote programma's. In dat licht moeten de eisen worden gezien die we in deze cursus aan programma's zullen stellen. We zullen daarom in deze paragraaf over de grens van de cursus kijken, naar softwareontwikkeling in het algemeen.

OPGAVE 1.1

Het is niet helemaal duidelijk wanneer een programma door één programmeur is geschreven. In de cursus Objectgeoriënteerd programmeren bijvoorbeeld, hebt u verschillende applicaties ontwikkeld. In zekere zin was u niet de enige programmeur van deze applicaties. U gebruikte immers klassen die door andere programmeurs waren ontwikkeld.

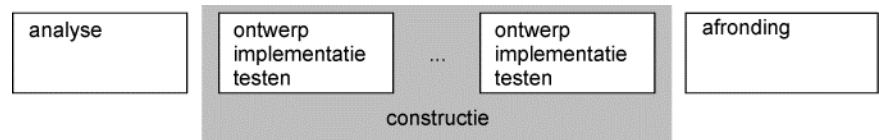
- Welke klassen waren dat zoal?
- Wat moest u van deze klassen weten om ze te kunnen gebruiken? Waar haalde u die informatie vandaan? Was deze altijd voldoende?

Hoe verloopt nu globaal de ontwikkeling van een groot softwaresysteem?

Voortraject

Voordat de ontwikkeling start, is er meestal al een heel traject afgelegd, het *voortraject*. Dit traject start bijvoorbeeld met het signaleren van een probleem of het ontstaan van een nieuwe markt. Kijk als voorbeeld naar het systeem voor het containerverhuurbedrijf. Het voortraject kan daar begonnen zijn met de constatering dat de bedrijfsresultaten verslechterden. Toen onderzocht werd hoe dat kwam, bleken de tarieven aan de hoge kant. Een analyse van het bedrijfsproces leerde, dat er kosten bespaard konden worden door de containers efficiënter te gebruiken. Het kwam te vaak voor dat er bijvoorbeeld een stel containers leeg van Rotterdam naar Liverpool werd gebracht, terwijl er in Birmingham vergelijkbare containers op een verhuurder stonden te wachten. Die vielen dan weer net onder een ander kantoor, zodat hun beschikbaarheid niet duidelijk werd. Eén informatiesysteem waarop alle kantoren zijn aangesloten, wordt als een mogelijke oplossing gezien. Als het management voldoende in het idee ziet om er geld in te steken, gaat het project pas echt van start. Tijdens het voortraject wordt meestal ook het echte ontwikkeltraject opgezet: er wordt bijvoorbeeld bepaald wat het mag kosten en hoe lang het mag duren.

Figuur 1.1 toont een overzicht van een mogelijke fasering vanaf dat moment.



FIGUUR 1.1 Fasering van een project waarin software iteratief wordt ontwikkeld

We gaan nu uitgebreider op de verschillende fasen in.

Analyse

In de eerste fase, de *analyse*, moet worden vastgesteld wat er nu eigenlijk precies ontwikkeld moet worden en aan welke eisen het product moet voldoen; pas daarna wordt besloten of het project verder gaat. Een precieze specificatie is nodig om vast te kunnen stellen of het systeem wel met bestaande methoden en technieken gerealiseerd kan worden en of dat kan binnen de randvoorwaarden die in het voortraject zijn bepaald. Ontwikkelaars, opdrachtgevers en toekomstige gebruikers moeten dus rond de tafel gaan zitten om die specificatie op te stellen. Een manier om dat te doen, is het opsporen van alle gewenste *gebruiksmogelijkheden* (*use cases*) van het systeem. Iedere gebruiksmogelijkheid is een soort scenario waarin een typische interactie tussen een mogelijke gebruiker en het systeem wordt weergegeven.

Gebruiks-
mogelijkheid

Engels: use case

Twee gebruiksmogelijkheden van het containersysteem zijn bijvoorbeeld de volgende:

- Het systeem verstrekt op verzoek van een medewerker, wiens taak het is om containers toe te wijzen, een overzicht van alle ongebruikte containers (aantal, type en locatie) op een gegeven datum in een gegeven gebied (een land, een groep landen of de hele wereld).
- Een medewerker voert een mogelijke order in (aantal containers van een bepaald type, plaats en datum van vertrek, plaats en datum van bestemming) en het systeem wijst op grond daarvan een beschikbare verzameling containers aan waarmee zo goedkoop mogelijk (volgens een gegeven kostenfunctie) aan de order kan worden voldaan.

Het is een kwestie van onderhandelen en afwegen welke gebruiksmogelijkheden in de uiteindelijke specificatie terechtkomen. In het voorbeeld kan het realiseren van de tweede gebruiksmogelijkheid zoveel kosten, dat het bedrijf besluit de uiteindelijke toewijzing toch met de hand te blijven doen en het systeem vooral te gebruiken om alle benodigde informatie centraal bij te houden en op een overzichtelijke wijze aan alle kantoren te leveren.

Een lijst van gebruiksmogelijkheden vormt de basis van verdere gesprekken tussen ontwikkelaar en opdrachtgever.

Domeinmodel

Tijdens analyse (vaak tegelijkertijd met het opstellen van de lijst van gebruiksmogelijkheden) wordt ook een *domeinmodel* opgesteld. In een dergelijk model wordt het deel van de werkelijkheid beschreven waar het systeem betrekking op heeft. Bij een objectgeoriënteerde analyse wordt die beschrijving gegeven in termen van klassen, hun relaties en hun interacties. Oppervlakkig lijkt het domeinmodel daarom op een ontwerp voor een objectgeoriënteerd programma, maar dat is het niet!

Bij het opstellen van het domeinmodel wordt nog helemaal geen rekening gehouden met de functionaliteit die de software moet gaan bieden. Een domeinmodel is bovendien vaak niet volledig; allerlei details worden weggelaten om het geheel begrijpelijk en hanteerbaar te houden. Ook de essentiële processen uit het domein worden tijdens de analyse beschreven, bijvoorbeeld: Hoe wordt een container gevolgd? Hoe wordt een order van een klant verwerkt?

Constructie

Als het project niet na de analysefase is afgeblazen, kan nu de feitelijke *constructie* van het systeem beginnen. Dat gebeurt vrijwel altijd in fasen. De lijst van gebruiksmogelijkheden kan hierbij als uitgangspunt dienen: in iedere fase wordt een deel van de gebruiksmogelijkheden gerealiseerd. Aan het eind van iedere fase is er dus een werkend systeem, dat echter nog niet alles doet wat het volgens de specificaties zou moeten doen. Iedere fase in de constructie bestaat zelf weer uit drie stappen: ontwerp, implementatie en testen.

Ontwerp

Tijdens het *ontwerp* van een objectgeoriënteerd systeem wordt vastgesteld uit welke klassen het programma zal gaan bestaan en hoe die klassen met elkaar samenhangen. Aan het eind van een ontwerpstap moet het volledig duidelijk zijn *welke* verantwoordelijkheden elke klasse in het programma tot dan toe zal hebben.

Ontwerpmodel

Wat dan nog niet vastligt, is *hoe* elke klasse haar verantwoordelijkheden zal realiseren. De resulterende klassenstructuur zullen we het *ontwerpmodel* noemen. Vaak zal het domeinmodel als uitgangspunt dienen voor het opstellen hiervan, maar het ontwerpmodel kan ook aanzienlijk afwijken van het domeinmodel. Er kunnen extra klassen aan worden toegevoegd die niet overeenkomen met elementen uit het domein, maar die louter een beheertaak hebben. Ook worden er soms andere klassen gekozen, omdat de klassen uit het domeinmodel tot een onhandige of een inefficiënte implementatie zouden leiden.

Technisch ontwerp

In deze cursus verdiepen we ons niet in zulke gevallen, maar het is goed om u te realiseren dat bij programmeren in het groot het modelleren van het domein en het ontwerpen van de software gescheiden activiteiten zijn die tot verschillende resultaten kunnen leiden. Voor het software-ontwerp wordt daarom ook wel de term *technisch ontwerp* gebruikt. Verder moet u zich realiseren dat een ontwerp vaak een uitbreiding is van een vorig ontwerp. De constructie verloopt immers in fasen; in elke fase worden gebruiksmogelijkheden toegevoegd. Soms kan daarvoor worden volstaan met het uitbreiden van bestaande klassen, soms moeten nieuwe klassen worden toegevoegd.

Implementatie

Als er een bevredigend ontwerp ligt, kan begonnen worden met de *implementatie*. Per klasse moet worden vastgesteld *hoe* deze klasse haar verantwoordelijkheden gaat verwezenlijken. Eventueel kan dit worden vastgelegd in een apart implementatiemodel waarin alle attributen met hun typen zijn vastgelegd en alle methoden met hun signatuur (de kop van de methode) en met een beschrijving van hun werking. Vervolgens wordt de klasse gecodeerd. Omdat, tijdens de ontwerpfase, de interface van een klasse gedetailleerd is vastgelegd, kunnen in deze fase verschillende programmeurs onafhankelijk van elkaar aan verschillende klassen werken.

Testen

Het *testen* van een bepaalde constructiestap verloopt ook weer in stappen. Elke klasse wordt eerst afzonderlijk getest. De programmeur zal daarvoor een kleine testomgeving construeren. In Objectgeoriënteerd programmeren maakten we hiervoor gebruik van JUnit. Als alle klassen lijken te werken, worden ze samengevoegd en wordt het programma als geheel getest. Hierbij zullen eventuele misverstanden tussen de verschillende programmeurs aan het licht komen. Als het implementatiemodel volledig en eenduidig is en alle programmeurs volmaakt zijn, dan zijn die misverstanden er niet. De praktijk leert dat aan die voorwaarden vaak niet is voldaan.

Iedereen die regelmatig met computers werkt, weet hoe moeilijk testen is: in ieder softwarepakket van enige omvang blijken toch altijd weer fouten te zitten. Om dat aantal zo klein mogelijk te houden, is een goede teststrategie van zeer groot belang. Het zal duidelijk zijn dat voor ieder programma, hoe klein ook, slechts een miniem deel van alle mogelijke combinaties van invoerwaarden daadwerkelijk kan worden uitgetest. De kunst is nu dat deel zo te kiezen, dat het zo veel mogelijk representatief is voor alle invoer.

Afronding

Er komt een moment waarop ook de laatste gebruiksmogelijkheid is toegevoegd en getest en het systeem dus in principe af is. Er is dan vaak nog een *afrondingsfase* waarin de opdrachtgever bijvoorbeeld het systeem al ter beschikking krijgt om het uit te proberen. Of, als het een nieuw pakket betreft, wordt op dat moment misschien een gratis te downloaden bètaversie op internet gezet, zodat potentiële klanten deze uit kunnen proberen. Daar komen altijd nog fouten uit, die in de afrondingsfase verbeterd kunnen worden. Tot slot wordt het systeem dan echt overgedragen of vrijgegeven voor verkoop.

Unified Process (UP)

Het hier geschetste ontwikkeltraject is niet het enig mogelijke. Er zijn bijvoorbeeld andere faseringen mogelijk. Als ook de analyse en de afronding in fasen worden uitgevoerd, bestaat het hele traject uit het herhaaldelijk (iteratief) doorlopen van de stappen analyse, ontwerp, implementatie, testen en afronding van de huidige fase. De opdrachtgever heeft dan al snel een klein werkend systeem ter beschikking. Na iedere volgende fase zullen de mogelijkheden zijn uitgebreid. Deze iteratieve aanpak is te beschouwen als een eenvoudige vorm van het *Unified Process (UP)*. Kenmerkend voor het UP is het iteratief en incrementeel ontwerpen en implementeren van een systeem.

Prototype

Op deze wijze kan ook een *prototype* worden ontwikkeld: een experimentele versie van een te realiseren systeem. Meestal wordt een prototype gebruikt om inzicht te krijgen in de werking van een uiteindelijk programma. Aan de hand van zo'n prototype kan worden getoetst of aan alle eisen is voldaan en of het programma aan de verwachtingen van de opdrachtgever voldoet. Doordat de opdrachtgever in een vroeg stadium met het prototype kan 'spelen', is het mogelijk het ontwerp tijdig in een volgende fase bij te stellen.

Aan de andere kant zou het volledige ontwerp en de implementatie in een keer kunnen worden gedaan, zodat de constructie uit slechts één stap bestaat. Hoe groter het systeem is, hoe sterker dit laatste echter moet worden afgeraden: behoorlijk testen van heel veel code ineens is vrijwel onmogelijk. Op de voor- en nadelen van verschillende faseringen gaan we hier verder niet in.

Onderhoud

Als het systeem is opgeleverd of op de markt is gebracht, is daarmee de kous nog lang niet af. Het systeem moet namelijk ook *onderhouden* worden en in de praktijk is daarmee vaak veel meer tijd en geld gemoeid dan met de oorspronkelijke constructie.

Waar bestaat dat onderhoud uit? Ten eerste zullen er, hoe goed er ook getest is, nog steeds fouten in het programma zitten, die in een volgende versie verbeterd moeten worden. Ten tweede zullen er veranderingen in het domein optreden of in de omgeving waarin het programma draait, die aanpassingen in het systeem nodig maken. Ten derde kunnen er uitbreidingen van het programma nodig zijn, wat wil zeggen dat er nieuwe gebruiksmogelijkheden aan moeten worden toegevoegd.

Ook hiervan kan de oorzaak liggen in veranderingen in het domein, maar het kan ook zijn dat het regelmatige gebruik van het systeem de gebruiker op nieuwe ideeën brengt: 'het zou toch ook wel handig zijn als ...', of dat de voortdurende voortschrijdende techniek nieuwe mogelijkheden binnen bereik brengt.

OPGAVE 1.2

Bedenk eens een paar veranderingen in de omstandigheden die een wijziging of een uitbreiding van het containersysteem nodig maken.

Een programma kan op deze wijze vele jaren meegaan, in steeds weer nieuwe gedaanten, zonder ooit echt uit de roulatie te worden genomen. Uit het oogpunt van softwarearchitectuur zou het misschien verstandig zijn om programma's elke vijf of hoogstens tien jaar te vervangen door een volledig nieuwe versie die van de grond af aan is herontwikkeld. Daar zijn echter vaak zulke hoge kosten mee gemoeid dat een dergelijke oplossing economisch niet haalbaar is. Het millenniumprobleem, waar in de laatste jaren van de vorige eeuw veel aandacht voor was, kan hier gebruikt worden als illustratie: de oerversie van sommige programma's waarin jaartallen met slechts twee cijfers gerepresenteerd werden, dateerde nog uit de jaren zestig van de vorige eeuw!

In de totale levensloop van een groot systeem is onderhoud dus minstens zo belangrijke factor als de oorspronkelijke ontwikkeling.

1.2 WAT IS EEN GOED PROGRAMMA?

Welke eisen kunnen we nu, in het licht van de zojuist geschetste levensloop van softwaresystemen, stellen aan ontwerp en implementatie van een dergelijk systeem?

Correctheid

Allereerst moet een programma uiteraard *correct* zijn: het moet doen wat het volgens de specificaties zou moeten doen. Als gebruik gemaakt wordt van formele specificatietechnieken, kan de correctheid van een programma in principe met wiskundige methoden bewezen worden. In deze cursus zullen we dat niet doen; in plaats daarvan zullen we, net als vrijwel altijd in de praktijk gebeurt, via testen fouten opsporen en die vervolgens herstellen. Realiseert u zich wel dat deze strategie beperkingen kent. Een goede teststrategie kan veel fouten aan het licht brengen, maar via testen kan nooit worden aangetoond dat het uiteindelijke programma honderd procent correct is.

Correctheid is overigens een vrij beperkt begrip. In feite moet een programma doen wat de opdrachtgever wil of wat de toekomstige gebruikers verlangen. Als de specificatie daarmee achteraf niet in overeenstemming blijkt en de gebruikers ervaren het programma als onhandig of zelfs onbruikbaar, dan is het programma niet goed, zelfs niet wanneer bewezen is dat het correct is.

Robuustheid

Correctheid alleen is bovendien niet genoeg. Een programma moet ook *robust* zijn, ofwel: het moet tegen mogelijke fouten bestand zijn. Dat kunnen fouten van de gebruiker zijn: bijvoorbeeld het invoeren van letters als een getal wordt verwacht, of het opgeven van een niet-bestaande datum, of het opvragen van gegevens uit een bestand die daar niet in zitten, of het kiezen van een verkeerde menuoptie waardoor een onbedoelde interactie wordt gestart of juist het abusievelijk afbreken van een lopende interactie.

In zulke gevallen verwachten we een redelijke respons van de software waarmee we werken: een geluid dat aangeeft dat we iets doen wat niet klopt, een heldere maar beknopte foutmelding, een mogelijkheid om de gestarte interactie meteen weer af te breken, of een waarschuwing dat we op het punt staan (veel) werk weg te gooien. Het programma moet bovendien bestand zijn tegen allerlei andere onvoorziene omstandigheden, zoals een harde schijf die vol is zodat er geen data kunnen worden weggeschreven.

Begrijpelijke code

Wil een programma onderhouden kunnen worden, dan moet de code *begrijpelijk* zijn. De programmeur die een fout moet verbeteren of een wijziging aanbrengen, is vaak een andere dan degene die de code oorspronkelijk heeft geschreven. Als de tijd tussen schrijven en veranderen langer dan een paar maanden is, maakt dat bovendien weinig verschil meer: programmeurs staan dan net zo vreemd tegenover hun eigen code als tegenover die van ieder ander.

Een manier om de begrijpelijkheid van code te verbeteren, is een goede documentatie van het programma in de vorm van uitgebreid commentaar. In onze Java-cursussen gebruiken we daarvoor veelal javadoc, aangevuld met gewoon commentaar bij lastige stukken broncode.

Uitbreidbaarheid

Ook de structuur van het programma moet dusdanig zijn, dat het *makkelijk te wijzigen en uit te breiden* is. Een wijziging is makkelijker naarmate deze op minder delen van het programma invloed heeft. Wanneer op twintig plekken iets gewijzigd moet worden, zien we er gauw een over het hoofd. Een uitbreiding is gemakkelijker naarmate het nieuwe stuk zelfstandiger te ontwikkelen is en tot minder wijzigingen elders in het programma leidt.

Herbruikbaarheid

Er zijn nog meer eisen waar een programma aan moet voldoen, maar daar besteden we in deze cursus minder aandacht aan. We noemen er nog twee.

Een gewenste eigenschap die objectoriëntatie populair heeft gemaakt, is *herbruikbaarheid* van delen van het systeem. Soms kan het veel ontwikkeltijd besparen als een klasse die voor een bepaalde toepassing gemaakt is, ook in een andere toepassing kan worden ingezet. Het belang van deze eigenschap is u in feite al bekend: de Java API is in wezen niets anders dan een verzameling herbruikbare klassen.

Efficiëntie

Tot slot is voor sommige programma's *efficiëntie* heel belangrijk. Dit geldt bijvoorbeeld voor:

- toepassingen waarbij het programmaverloop een proces in de buitenwereld moet bijhouden, zoals een industriële robot die moet reageren op aanvoer van onderdelen op een band
- toepassingen waarbij een factor tien in geheugengebruik of tijd net de grens tussen bruikbaar en onbruikbaar vormt: een weervoorspelling voor de komende week mag een dag rekentijd vragen, maar geen tien dagen
- pakketten waarbij de interactie met de gebruiker zeer intensief is en wachten al snel hinderlijk wordt: besturingssystemen, tekstverwerkers, ontwikkelomgevingen.

Al deze eisen zijn op zich redelijk voor de hand liggend. De vraag is echter, hoe we die eisen vertalen in eigenschappen van programmacode. Daarbij zullen we ons in de rest van deze paragraaf concentreren op begrijpelijkheid en gemak van wijzigen en uitbreiden. In tegenstelling tot bijvoorbeeld robuustheid en efficiëntie zijn dit namelijk de eisen waarmee van het begin af aan rekening moet worden gehouden.

Het is mogelijk om een niet-robuust, maar begrijpelijk en makkelijk te wijzigen programma, achteraf alsnog robuust te maken. Een ondoorgrondelijk en moeilijk te wijzigen programma achteraf alsnog begrijpelijk maken is niet echt onmogelijk, maar wel uiterst moeilijk. Vaak vereist het een volledig nieuw ontwerp en implementatie.

OPGAVE 1.3

Noem een paar eigenschappen van programmacode die de begrijpelijkheid ervan bevorderen.

De eigenschappen genoemd in de terugkoppeling bij opgave 1.3 liggen heel erg voor de hand, maar er zijn er meer.

Kleine klassen en methoden

Een heel belangrijke eigenschap die de begrijpelijkheid van code bevordert is *eenvoud*. In het algemeen zullen klassen en methoden moeilijker te begrijpen zijn naarmate ze groter zijn. Het is daarom goed om te streven naar *kleine klassen en methoden*. Zoals u bij het gebruik van de API specificatie gemerkt zult hebben, voldoen de klassen uit de Java API niet altijd aan deze norm. Ook is een methode begrijpelijker naarmate de *control flow* *eenvoudiger* is. Daarom proberen we een diepe nesting van *while*'s, *for*'s en *if*'s te vermijden.

Eenvoudige control flow

Voorbeeld

Een klein maar vaak gezien voorbeeld van moeilijk te lezen code is het volgende:

```
if (!test) {
    return false;
} else {
    return true;
}
```

Dit kan korter en daarmee duidelijker opgeschreven worden:

```
return test;
```

*Gescheiden
verantwoordelijk-
heden*

Een andere belangrijke manier om eenvoud te bereiken, is verwoord in het volgende principe: *ieder onderdeel van de code dient slechts één verantwoordelijkheid te hebben*. Dit principe geldt voor alle niveaus: opdrachten, methoden en klassen.

Opdrachten

– Op het niveau van individuele opdrachten gebruiken we vanwege dit principe nooit de waarden van expressies met neveneffecten. Java kent bijvoorbeeld opdrachten van de vorm $k++$ en $++k$. Beide verhogen de waarde van de int-variabele k met 1. Deze uitdrukkingen hebben echter ook een waarde. De waarde van $k++$ is de waarde van k voor verhoging; de waarde van $++k$ is de waarde van k na verhoging. We kunnen bijvoorbeeld schrijven

```
int n = 3;
int m = 3;
int s = n++ + 1;
int t = ++m + 1;
```

Hierna zijn n en m beide gelijk aan 4, s is ook gelijk aan 4 ($3 + 1$) en t is gelijk aan 5 ($4 + 1$). Sommige programmeurs gebruiken dit soort opdrachten graag om compacte code te schrijven. De kleinste deler van een geheel getal n kan bijvoorbeeld gevonden worden met behulp van het volgende programmafragment:

```
deler = 1;
while ((n % ++deler) != 0);
```

In deze opdracht heeft de test twee verantwoordelijkheden: de variabele `deler` wordt verhoogd en er wordt bekeken of de rest van n bij deling door de nieuwe waarde van deze variabele ongelijk is aan 0. De `while`-opdracht bevat alleen een test en geen opdracht. Die test wordt dus eenvoudig herhaald tot de rest nul is en `deler` dus gelijk is aan de kleinste deler van n .

De volgende code is weliswaar iets langer, maar veel begrijpelijker:

```
deler = 2;
while (n % deler != 0) {
    deler++;
}
```

De kortste manier om iets op te schrijven, is dus niet bij voorbaat de eenvoudigste!

Methoden

– Op methodeniveau proberen we methoden te vermijden die zowel een waarde opleveren als een of meer attributen wijzigen. Een dergelijke methode heeft twee verantwoordelijkheden: iets aan de toestand van het object veranderen en een resultaat teruggeven.

Er zijn op deze regel overigens wel uitzonderingen. Het is bijvoorbeeld niet verkeerd om een methode een waarde terug te laten geven die laat zien of de actie die de methode moet uitvoeren, geslaagd is. Een voorbeeld hiervan is de methode

```
public boolean add(E e)
```

in de klasse ArrayList. Deze methode geeft de waarde true terug als het toevoegen van het element geleid heeft tot een verandering van de arraylist, en false als het toevoegen niet gelukt is.

Klassen

– Op klassenniveau zullen we het principe van gescheiden verantwoordelijkheden als een van de leidraden gebruiken bij de keuze van klassen in een ontwerp. Het is een goede gewoonte om bij het ontwerp het doel van een klasse – we kunnen ook zeggen: de verantwoordelijkheid van de klasse – in een kort zinnetje expliciet te vermelden.

Als in zo’n zinnetje het woordje ‘en’ voorkomt, moet op zijn minst onderzocht worden of die klasse niet beter gesplitst kan worden, zoals in: deze klasse representeert een order en coördineert de toewijzing van containers aan die order. Het antwoord is in elk geval ‘ja’ als dat kan gebeuren zonder dat er veel extra interactie tussen die klassen nodig is.

Gescheiden verantwoordelijkheden maken code gemakkelijker te begrijpen en alleen al daarom ook gemakkelijker te wijzigen en uit te breiden. Een wijziging of uitbreiding is echter ook des te gemakkelijker, naarmate deze op minder plaatsen in de code van invloed is. Stel bijvoorbeeld dat een wijziging invloed heeft op de signatuur van drie methoden uit drie verschillende klassen, en dat elk van die drie methoden op vijf verschillende plaatsen wordt aangeroepen. Om die ene wijziging door te voeren, moeten we dan op tenminste achttien plaatsen in de code iets veranderen: de drie methoden plus de vijftien aanroepen.

Niet alleen moet elk onderdeel van de code slechts één verantwoordelijkheid hebben, iedere verantwoordelijkheid van het systeem als geheel moet bij voorkeur ook slechts op één plek in de code gerealiseerd zijn.

Als we dan iets wijzigen aan die verantwoordelijkheid, dan hoeven we alleen op die plek iets te veranderen. We noemen deze eis aan de code het principe van *lokaliteit*.

Lokaliteit

Constanten

Ook dit principe speelt op verschillende niveaus. De eis van lokaliteit is bijvoorbeeld een van de redenen dat we *constanten* definiëren. Stel er zijn in het containersysteem vijf verschillende typen containers en dat aantal komt op tien plaatsen in de code voor. Als we op al die tien plaatsen het getal 5 neerzetten en er komt een type container bij, dan moeten we tien wijzigingen aanbrengen. Is er ergens in het systeem een constante AANTALCONTAINERTYPEN gedefinieerd, dan hoeven we alleen de waarde van die constante te veranderen.

OPGAVE 1.4

Welke andere reden is er om constanten te definiëren?

Ook onze voorkeur voor attributen die van buiten de klasse niet gewijzigd kunnen worden (information hiding), is terug te voeren op het principe van lokaliteit: de verantwoordelijkheid voor het beheer van een dergelijk attribuut ligt geheel binnen de klasse, in plaats van verspreid door het hele programma. Met het oog daarop, maken we attributen vrijwel altijd *private* en zijn we voorzichtig met het exporteren van referenties naar attributen van een objecttype.

Het bereiken van lokaliteit is in de praktijk een van de moeilijkste zaken bij objectgeoriënteerd programmeren (en bij programmeren in het algemeen).

Voorbeeld

Op grond van het principe van lokaliteit willen we aan een klasse gemakkelijk een nieuwe subklasse kunnen toevoegen. Stel bijvoorbeeld dat er in het containersysteem een klasse *Container* is, met een subklasse voor elke soort container. Als we nu een nieuw soort container willen toevoegen, zullen we in elk geval een nieuwe subklasse van *Container* moeten definiëren. Daarnaast zijn wijzigingen nodig op plekken in de code waar instanties van de subklassen van *Container* gecreëerd worden en op plekken waar het nodig is na te gaan met welk type container we nu precies te maken hebben. Lokaliteit vereist dat dit op zo min mogelijk plaatsen voorkomt. In de volgende leereenheid zullen we zien hoe Java ons daarbij helpt.

Lage koppeling

Nog een manier die we hier willen noemen om een programma eenvoudiger te maken, is het aantal associaties tussen klassen te beperken ofwel de *koppeling* tussen de klassen laag te houden. Als we een systeem hebben met tien klassen en iedere klasse is afhankelijk van alle andere, dan is dat systeem vrijwel zeker moeilijk te doorgronden en te wijzigen. Een wijziging in een klasse maakt dan al snel wijzigingen in alle andere klassen noodzakelijk. Hergebruik van één klasse in een ander systeem is al helemaal onmogelijk.

1.3 OBJECTKEUZE

Het is lang niet gemakkelijk om programmacode te schrijven met al de gewenste eigenschappen. Het kiezen van goede namen is vooral een kwestie van zelfdiscipline, evenals het schrijven van commentaar. Maar voor het schrijven van code die ook aan de andere eisen voldoet, is ervaring nodig. Een beginnend programmeur ziet vaak maar één manier om een probleem aan te pakken. Hij zal daarom al gauw tot de conclusie komen dat deze grote klasse echt niet gesplitst kan worden, of dat deze methode echt zowel een waarde moet opleveren als de toestand van een object moet veranderen. Soms heeft een andere oplossing inderdaad meer nadelen dan voordelen, maar meestal ziet een programmeur met enige ervaring wel hoe het anders kan. Nog veel meer ervaring is nodig om klassen dusdanig te kiezen en uit te werken, dat de resulterende code voldoet aan onder andere de principes van gescheiden verantwoordelijkheden, lage koppeling en lokaliteit.

De verschillende eisen aan de code zijn bovendien soms met elkaar in conflict. Soms gaan een vergaande scheiding van verantwoordelijkheden en een grote lokaliteit ten koste van de eenvoud van een ontwerp.

Er moet dan een afweging worden gemaakt. Het is daarbij heel belangrijk om te anticiperen op het soort wijzigingen dat later nodig kan zijn.

Schrijven we bijvoorbeeld een schaakprogramma, dan mogen we de afmetingen van het bord, de aard van de stukken en de geoorloofde zetten rustig fixeren: het is een veilige aanname dat die niet zullen veranderen. Het is daarentegen zeer belangrijk om de generator van een zet zo flexibel mogelijk te maken: we willen zeker in de toekomst nieuwe strategieën toevoegen.

Naarmate een programma groter is, de verwachte levensduur langer is en het minder voorspelbaar is welke wijzigingen er later noodzakelijk zijn, wordt het belangrijker om te kiezen voor scheiding van verantwoordelijkheden en lokaliteit. In deze cursus zullen we echter ook kiezen voor de eenvoud van het ontwerp.

2 UML-diagrammen

UML klassendiagrammen spelen een belangrijke rol in deze cursus. In deze paragraaf herhalen we de theorie hierover voor zover van belang binnen deze cursus.

Unified Modeling Language (UML)

De *Unified Modeling Language*, afgekort UML, is een modelleertaal voor objectgeoriënteerde systeembeschrijvingen. UML biedt verschillende diagramtechnieken en is sinds de lancering in 1997 tot een standaard uitgegroeid. UML kan tijdens de volledige ontwikkelingscyclus gebruikt worden:

- tijdens de analyse voor het opstellen van een specificatie en van een domeinmodel
- tijdens het technisch ontwerp voor de specificatie van een programmastructuur (inclusief indeling in packages)
- tijdens de implementatie voor een volledige beschrijving van klassen en de interactie tussen objecten.

UML standaard: zie www.uml.org

UML is ontworpen en wordt verder ontwikkeld door de Object management group (OMG), een volledige beschrijving kan worden geraadpleegd via de website www.uml.org.

In paragraaf 2.1 komen klassendiagrammen aan de orde. Ter herinnering: een klassendiagram laat zien welke klassen er zijn ontworpen, welke attributen en methoden deze hebben en in welke relatie de klassen tot elkaar staan.

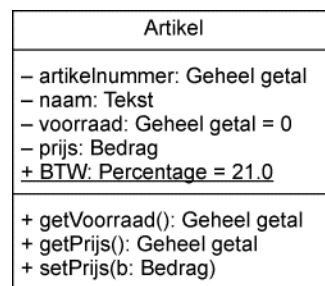
In paragraaf 2.2 besteden we kort aandacht aan de objectdiagrammen uit UML. Deze beschrijven de toestand van objecten op een bepaald tijdstip en zijn de UML-variant van de (complexere) toestandsdiagrammen die gebruikt worden in Objectgeoriënteerd programmeren.

2.1 KLASSENDIAGRAMMEN

Een klassendiagram toont klassen en hun samenhang. Klassendiagrammen kunnen tijdens analyse, ontwerp en implementatie gebruikt worden. In elke fase wordt alleen datgene in het klassendiagram opgenomen, wat in die fase belangrijk is. Het is dus belangrijk om te weten waarvoor een klassendiagram op een bepaald moment gebruikt wordt.

Tijdens de analyse wordt het domeinmodel opgesteld, waarin de belangrijke concepten voor het systeem met hun onderlinge relaties worden vastgelegd. Bij domeinmodellering bevat het klassendiagram klassen met attributen maar nog zonder methoden. Ook zullen de attributen nog niet voorzien worden van typen (dat is een latere ontwerp- of implementatiekeuze), en hebben de relaties tussen de klassen nog geen richting.

In deze cursus zullen we klassendiagrammen echter vooral gebruiken bij ontwerp en implementatie van programma's. Deze ontwerpklassendiagrammen bevatten gedetailleerdere informatie dan het klassendiagram van het domeinmodel.



FIGUUR 1.2 De klasse Artikel

Notatie van een klasse

We bekijken eerst de notatie voor een klasse. Figuur 1.2 toont een voorbeeld. Het diagram bestaat uit drie delen: in het bovenste deel staat de naam van de klasse, in het middelste deel staan attributen en in het onderste deel staan methoden.

Het middelste deel van het diagram bevat attributen. Volgens de UML-standaard staat het type van de attributen na de attribuutnaam en niet zoals in Java ervoor. Naam en type worden gescheiden door een dubbele punt.

Op ontwerpniveau worden soms geen typeaanduidingen gebruikt die direct aan Java zijn ontleend, zoals `int`, `double` en `String` maar algemenere, zoals `Geheel getal`, `Bedrag` en `Percentage` in figuur 1.2. De precieze representatie kan dan later nog worden vastgesteld.

In figuur 1.2 heeft de klasse `Artikel` de attributen `artikelnummer` en `voorraad` van het type `Geheel getal`, `naam` van het type `Tekst`, `prijs` van het type `Bedrag` en `BTW` van het type `Percentage`. Dit laatste attribuut is een klassenattribuut: een attribuut dat hoort bij de klasse en dat dus voor alle instanties van die klasse dezelfde waarde heeft. In UML worden dergelijke attributen onderstreept. `BTW` is een constante (genoteerd in hoofdletters) en krijgt de waarde `21.0`. Bij creatie van een instantie van `Artikel` krijgt het attribuut `voorraad` de waarde `0`. Dit is de zogeheten standaardwaarde van dat attribuut.

Syntaxis attribuut

Voor attributen hanteert UML de volgende syntaxis:

`[toegang] naam[: type] [= waarde]`

waarbij alles tussen rechte haken mag worden weggelaten.

Toegang geeft aan of het attribuut public (+), private (-), (~) package of protected (#) is (ter herinnering: protected betekent dat het attribuut toegankelijk is voor alle klassen uit dezelfde package en voor alle subklassen, ook wanneer deze in zich in andere packages bevinden). Wordt geen toegang vermeld, dan kan dat betekenen dat er in het diagram geen toegang gespecificeerd is; de toegang is immers niet verplicht. Dat zal gelden voor veel klassendiagrammen in deze cursus.

Naam en *type* geven de naam en het type van het attribuut weer. De aanduiding = *waarde* geeft een standaardwaarde aan, die het attribuut krijgt bij creatie van een instantie van de klasse.

UML laat de opsteller van het klassendiagram dus veel vrijheid: indien gewenst, kan van een attribuut enkel de naam worden vermeld.

OPGAVE 1.5

Geef een specificatie van een private-attribuut van de klasse Artikel met de naam verkrijgbaar en standaardwaarde onwaar.

Ook voor methoden laat UML de opsteller alle vrijheid om meer of minder informatie op te nemen. De minimale aanduiding bestaat uit de naam van de methode plus een paar haakjes. Desgewenst zullen we daar aanduidingen aan toevoegen van aantal en typen van de parameters, en van het type van de terugkeerwaarde. Ook hier staan typeaanduidingen altijd na de naam. In figuur 1.2 heeft de klasse Artikel methoden die de waarden van prijs respectievelijk voorraad teruggeven (getPrijs en getVoorraad) en een methode die een nieuwe waarde aan de prijs toekent (setPrijs).

Syntaxis methode

De volledige syntaxis voor methodeaanduidingen is

```
[toegang] naam([parameterlijst]) [: resultaattype]
```

Voorbeeld

Een methode van een klasse Persoon die gebruikt kan worden om te testen of een persoon ouder is dan een ander, zou op implementatieniveau voluit als volgt kunnen worden genoteerd:

```
+ouderDan(p: Persoon): boolean
```

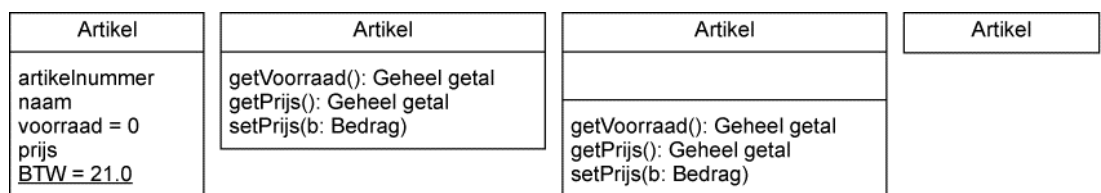
Informatie over de toegang tot attributen en methoden nemen we vaak niet op in onze klassendiagrammen, zeker niet als alle niet-constante attributen private zijn en alle methoden public.

OPGAVE 1.6

Gegeven een klasse Cirkel, met een attribuut straal met standaardwaarde 1. De klasse heeft methoden om de omtrek en het oppervlak van de cirkel te berekenen, alsmede een methode om de cirkel in een vlak te plaatsen met het middelpunt op gegeven coördinaten. Zoals gewoonlijk zijn attributen private en methoden public. Teken een klassendiagram waarin deze informatie over attributen en methoden is opgenomen.

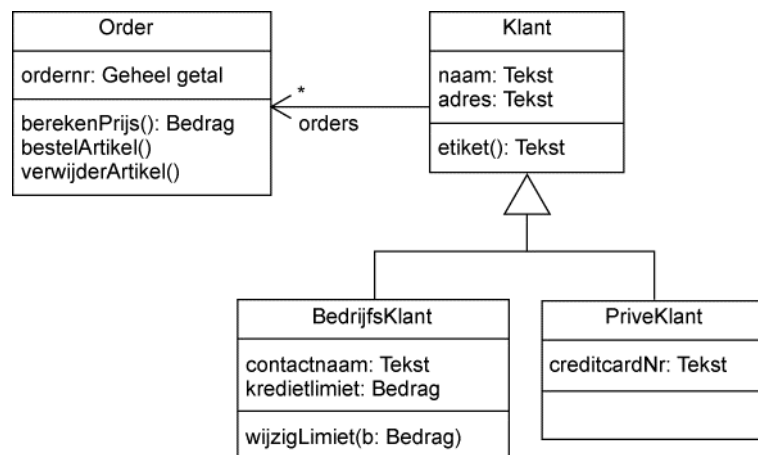
Methoden worden in UML overigens *operaties* genoemd. In deze cursus zullen we ons echter aan de Java-terminologie houden.

Attributen en/of methoden kunnen worden weggelaten uit een diagram. Figuur 1.3 toont enkele alternatieve notaties voor de klasse Artikel. Als in een diagram een klasse voorkomt zonder attributen of methoden, kan dat betekenen dat de klasse die niet heeft, dat ze nog niet bekend zijn of dat ze zijn weggelaten omdat ze in de context waarin het diagram gebruikt wordt, niet van belang zijn. In een diagram waarin tien klassen voorkomen, is het bijvoorbeeld goed vooral te concentreren op de relaties tussen die klassen; attributen en methoden kunnen dan alleen bij naam worden aangeduid of zelfs geheel worden weggelaten. Uit de context waarin het diagram gebruikt wordt moet uiteraard wel duidelijk worden wat precies het geval is.



FIGUUR 1.3 Andere notaties voor de klasse Artikel

We bekijken vervolgens de *relaties* tussen de klassen. Figuur 1.4 toont een klassendiagram met twee soorten relaties.



FIGUUR 1.4 Generalisatie en associatie

Generalisatie

De klasse Klant uit figuur 1.4 heeft twee subklassen BedrijfsKlant en PriveKlant via overerving (*generalisatie*). De superklasse Klant heeft de attributen naam en adres en een methode etiket die worden geërfd door de subklassen. Een bedrijfsklant heeft nog twee extra attributen en een extra methode; een privé-klant heeft één extra attribuut.

Associatie

UML toont associaties niet als attributen

Bij een klant hoort een lijst orders. Dit is in het diagram weergegeven als een *associatie* van Klant naar Order. Merk op dat in het diagram van de klasse Klant *geen attribuut* orders van bijvoorbeeld het type `ArrayList<Order>` is opgenomen. De pijl in figuur 1.4 betekent dat iedere instantie van klasse Klant verwijst naar een aantal instanties van Order.

De reden dat we associaties niet (meteen) als attributen in klassen willen opnemen is dat klassendiagrammen worden gebruikt tijdens verschillende fasen van de ontwikkeling. Een associatie vertalen naar een attribuut betekent vaak al kiezen voor een bepaalde implementatie, op een moment dat dat nog helemaal niet nodig is.

Laten we als illustratie kijken naar een driehoek met drie hoekpunten. Figuur 1.5 toont een mogelijk klassendiagram, gemaakt in de ontwerpfase.



FIGUUR 1.5 Ontwerp van een klasse Driehoek

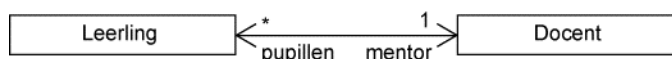
Op grond van dit klassendiagram kunnen we in de implementatie een attribuut `hoekpunten` opnemen van het type `ArrayList<Punt>` of van het type `Punt[]`. Maar we kunnen ook drie aparte attributen `puntA`, `puntB` en `puntC` van type `Punt` opnemen, of voor nog een andere implementatie kiezen. Java biedt meer mogelijkheden dan alleen de array of `ArrayList` om een meervoudige waarde te representeren. Tijdens het ontwerp willen we dat niet vastleggen.

UML kent zowel eenzijdige als tweezijdige associaties, zie figuur 1.6. De *eenzijdige associatie*, een associatie in één richting, wordt getekend met een pijl. In dit geval verwijst een instantie van klasse A naar een instantie van klasse B, maar niet omgekeerd. Bij de *tweezijdige associatie*, een associatie in beide richtingen, verwijst een instantie van A naar een instantie van B, maar omgekeerd verwijst een instantie van B ook naar een instantie van A.



FIGUUR 1.6 Eenzijdige en tweezijdige associatie

Anders dan in Objectgeoriënteerd programmeren, zullen we aan associaties vrijwel altijd namen en multipliciteiten toevoegen. Kijk als voorbeeld naar figuur 1.7, die een tweezijdige associatie toont tussen de klassen `Leerling` en `Docent`. Die associatie kan bijvoorbeeld geïmplementeerd worden door de klasse `Leerling` een attribuut `mentor` te geven van type `Docent` en `Docent` een attribuut `pupillen` van type `ArrayList<Leerling>`. UML spreekt in dit geval niet van attributen: in plaats daarvan wordt gezegd dat de associatie tussen `Leerling` en `Docent` een *rol* `mentor` heeft (die wordt ingevuld door een instantie van `Docent`) en een *rol* `pupillen` (die wordt ingevuld door instanties van `Leerling`). De naam van een rol staat aan de kant van de klasse die de rol vervult, niet aan de kant van de klasse die gebruik maakt van de rol.



FIGUUR 1.7 Een associatie met rolnamen en multipliciteitsaanduidingen

Eenzijdige
associatie
Tweezijdige
associatie

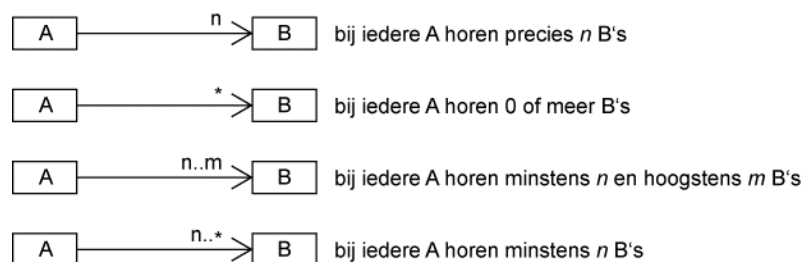
Rol

Let op!

Multipliciteit

Figuur 1.7 toont ook aan beide kanten een aanduiding van de *multipliciteit* van de rol (ook wel *kardinaliteit* genoemd). Een leerling heeft precies één mentor (de aanduiding 1), maar een docent kan mentor zijn van 0 of meer verschillende leerlingen (aanduiding *).

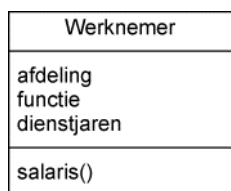
Figuur 1.8 toont de verschillende aanduidingen voor multipliciteit. Als er geen multipliciteitaanduiding aan één van de zijden van de associatie staat, dan is de multipliciteit aan die zijde gelijk aan 1.



FIGUUR 1.8 Multipliciteit

OPGAVE 1.7

Het ontwerp van een informatiesysteem voor een salarisadministratie bevat een klasse Werknemer, met attributen en een methode als getoond in figuur 1.9.



FIGUUR 1.9 Klasse Werknemer

Voor iedere werknemer moet bovendien worden bijgehouden wie daarvan de chef is (zelf ook een werknemer), en wie de eventuele ondergeschikten. Het betreft hier relaties tussen werknemers onderling en dus een associatie van de klasse Werknemer met zichzelf. Teken deze associatie, inclusief namen en multipliciteit.

De volgende opgave dient om u er nog eens duidelijk op te wijzen aan welke kant de naam en multipliciteit van een rol getekend moeten worden.

OPGAVE 1.8

Bekijk figuur 1.10.

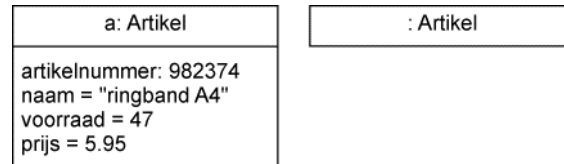


FIGUUR 1.10 Een associatie tussen A en B

Als deze associatie wordt geïmplementeerd door attributen in de klassen A en B, wat zijn dan voor de hand liggende namen en typen voor die attributen?

2.2 OBJECTDIAGRAMMEN

Uit Objectgeoriënteerd programmeren kent u de toestandsdiagrammen. Deze diagrammen komen in UML niet voor. UML kent wel een ander soort objectdiagram; zie de voorbeelden in figuur 1.11.



FIGUUR 1.11 Voorbeelden van objectdiagrammen van een instantie van klasse Artikel

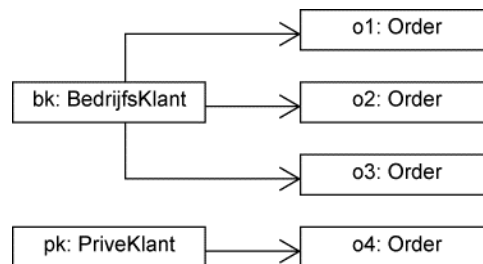
Het linkerdeel van figuur 1.11 toont een objectdiagram van een instantie a van klasse Artikel. De kop vermeldt in dit geval de naam van de instantie en het type. Het deel onder de streep toont de objectattributen en hun waarden. Klassenattributen worden in een objectdiagram niet getoond: ze horen immers bij de klasse en niet bij het object. Methoden worden in een objectdiagram evenmin getoond.

Het rechterdeel van figuur 1.11 toont een zogenaamde *anonieme instantie*: in de kop staat alleen de klasse vermeld, voorafgegaan door een dubbele punt. In dit deel zijn bovendien de attribuutwaarden weggelaten; dat is toegestaan als we daar niet in geïnteresseerd zijn (ook bij niet-anonieme instanties).

Anonieme instantie

Link

In een objectdiagram kunnen *links* voorkomen: een pijl van object a naar object b betekent dat object b bij object a hoort, of anders gezegd, dat object a object b kent en dus bijvoorbeeld een methode op object b kan aanroepen. In de implementatie betekent dit bijvoorbeeld dat object a een attribuut heeft met object b als waarde. Figuur 1.12 toont als voorbeeld links tussen twee klanten en vier orders.



FIGUUR 1.12 Een objectdiagram

Net als associaties kunnen links eenzijdig of tweezijdig zijn. De links uit figuur 1.12 zijn vanwege de pijl eenzijdig; Bedrijfsklant bk kent dus bijvoorbeeld order o1 maar het omgekeerde geldt niet. Als de link tussen deze twee objecten tweezijdig was (zonder pijl), dan zou order o1 ook BedrijfsKlant bk kennen.

OPGAVE 1.9

- Leg uit dat het objectdiagram uit figuur 1.12 in overeenstemming is met het klassendiagram uit figuur 1.4.
- Waarom staan er bij links tussen objecten geen multipliciteitsaanduidingen?

3 Een eenvoudige simulatie van een bank

In deze paragraaf bespreken we stap voor stap een ontwerp van een klein objectgeoriënteerd programma.

3.1 PRODUCTOMSCHRIJVING EN GEBRUIKSMOGELIJKHEDEN

Productom-
schrijving

Ontwerp een programma voor de simulatie van het beheer van rekeningen door een bank.

Deze bank kent twee soorten rekeningen, te weten betaalrekeningen en spaarrekeningen. Bij iedere rekening moet de rekeninghouder en een uniek rekeningnummer worden bijgehouden. Van de rekeninghouder moet in elk geval de naam bekend zijn. Van een rekening dient het saldo opgevraagd te kunnen worden.

Een betaalrekening wordt gebruikt voor het dagelijkse betalingsverkeer. Er kan contant geld op de betaalrekening worden gestort en er kan contant geld van worden opgenomen. Bovendien kan geld worden overgemaakt van een betaalrekening naar een andere rekening (betaalrekening of spaarrekening). Opname en overmaken kunnen alleen worden uitgevoerd onder de voorwaarde dat het saldo op de betaalrekening niet negatief wordt. Er wordt geen rente betaald over de tegoeden op de betaalrekening.

Ook op een spaarrekening kan contant geld worden gestort. Verder heeft de spaarrekening een aantal beperkingen ten opzichte van de betaalrekening. Het is niet mogelijk geld over te maken van een spaarrekening naar een willekeurig andere rekening. Bij opening van een spaarrekening dient een betaalrekening te worden opgegeven, de zogenaamde tegenrekening. Bij opname van geld van de spaarrekening wordt het gevraagde bedrag niet contant uitgekeerd, maar gestort op deze tegenrekening. Per kalenderjaar mag er slechts € 10.000,- worden opgenomen. Over hogere opnamen moet een boete van 3 % betaald worden. Ook bij opname van een spaarrekening geldt dat het saldo niet negatief mag worden. Over het saldo op een spaarrekening wordt per jaar 5 % rente betaald. De rente wordt per maand berekend, op grond van het saldo op de eerste van die maand, maar slechts eenmaal per jaar bijgeschreven.

De gebruiker van de simulatie moet de bank opdracht kunnen geven geld te storten en op te nemen van rekeningen en geld over te maken van de ene naar de andere rekening (alles voor zover het soort rekening dat toestaat). Ook moet saldo-informatie over een rekening opgevraagd kunnen worden.

De simulatie houdt ook een datum bij. Bij de start van de simulatie is deze gelijk aan de echte huidige datum. De gebruiker moet de datum kunnen verzetten (altijd vooruit, dus naar de toekomst) zodat er acties op verschillende data kunnen worden uitgevoerd. Steeds als de datum wordt verzet, moet het systeem de rente berekenen die de spaarrekeningen hebben opgeleverd in de tussenliggende tijd. Als de nieuwe datum in een nieuw jaar ligt, moet de rente over het afgelopen jaar (of de afgelopen jaren) ook daadwerkelijk worden bijgeschreven.

Alle gebruikersacties worden mogelijk gemaakt via een grafische user-interface.

We gaan nu eerst een lijstje opstellen van de gebruiksmogelijkheden van de simulatie. Wat moet de gebruiker er zoal mee kunnen doen?

OPGAVE 1.10

Probeer, op grond van de productomschrijving, een lijst met gebruiksmogelijkheden op te stellen. Iedere gebruiksmogelijkheid beschrijft een interactie van de gebruiker met het systeem. Om u op weg te helpen, geven we hier twee voorbeelden:

- Maak een nieuwe betaalrekening aan.
- Geef het saldo van de rekening met gegeven nummer.

3.2 OPSTELLEN DOMEINMODEL

De volgende stap is nu, om op grond van de productomschrijving domeinklassen te kiezen die zeker nodig zijn. Iedere productomschrijving zal in elk geval een paar van deze klassen opleveren.

Welke domeinklassen kunt u, op grond van de productomschrijving, onmiddellijk benoemen?

In dit geval zijn dat er minimaal drie, namelijk Bank, Betaalrekening en Spaarrekening. U kunt ook Rekeninghouder als klasse beschouwen. De productomschrijving is vaag over wat er van een rekeninghouder moet worden bijgehouden (in ieder geval de naam). In een dergelijk geval is het handig om zo'n klasse dan toch op te nemen in het klassendiagram. Later, tijdens het verdere ontwerp, kan alsnog besloten worden dat deze klasse overbodig is. Misschien ziet u nog meer klassen, maar we zullen het hier voorlopig bij laten. We zullen zien dat er andere klassen naar voren komen door de eisen aan de software goed in de gaten te houden. Klassen die betrekking hebben op de gebruikersinterface worden op dit moment van het ontwerp nog niet beschouwd. We zijn namelijk alleen de domeinlaag aan het ontwerpen.

Vervolgens moet er worden geformuleerd welke globale verantwoordelijkheid iedere klasse heeft. We doen dat in de vorm van een kort zinnetje, dat later als commentaar in de kop van de klassendefinitie opgenomen kan worden.

OPGAVE 1.11

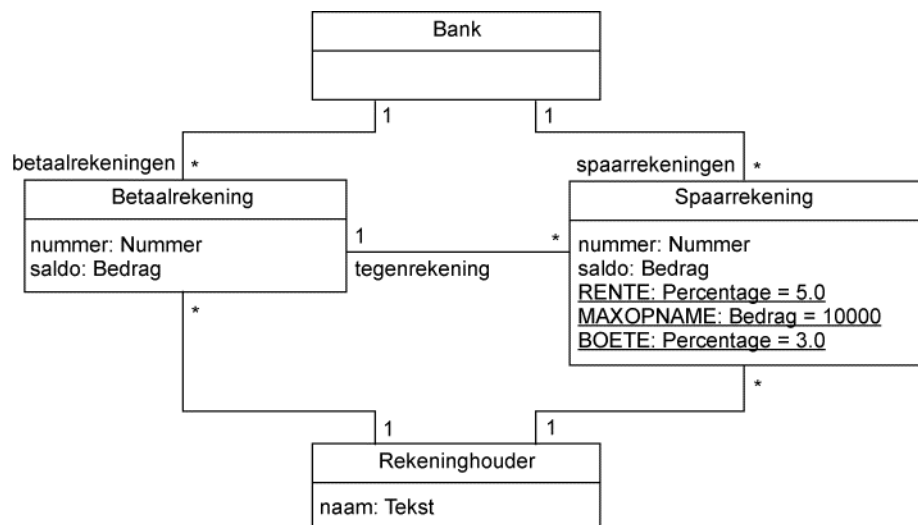
Formuleer de verantwoordelijkheid van elk van de vier tot nu toe benoemde klassen.

Als volgende stap kijken we of er attributen zijn die we op grond van de productomschrijving *zeker* nodig zullen hebben.

Welke attributen en associaties moeten de benoemde klassen *in elk geval* hebben?

Als we om te beginnen alleen de attributen en associaties opnemen waar we in geen geval buiten kunnen, dan komen we uit op figuur 1.13. Volgens de productomschrijving moet bij iedere rekening een uniek rekeningnummer worden bijgehouden. Ook wordt expliciet het saldo van de rekeningen genoemd en dus zullen we ook dat meteen als attribuut benoemen. Van een rekeninghouder moeten we in ieder geval de naam opslaan. Verder zien we bij de spaarrekening een aantal eigenschappen staan (rentepercentage, maximum op te nemen bedrag, en boetepercentage). De klasse Spaarrekening kan deze eigenschappen vastleggen in constanten; dit zijn ook attributen.

Relaties tussen klassen worden als associaties in een klassendiagram opgenomen, dus niet als attributen. Omdat we nog bezig zijn met het domeinmodel, zullen de associaties nu nog geen richtingen krijgen. De bank beheert verschillende betaalrekeningen en verschillende spaarrekeningen. Dit komt tot uiting in een associatie van Bank naar Betaalrekening en een associatie van Bank naar Spaarrekening, beide met multiplicititeit * aan de kant van de rekeningklasse om aan te geven dat de bank 0 of meer rekeningen kan beheeren. Verder behoort bij iedere betaalrekening en bij iedere spaarrekening een rekeninghouder, aangegeven door een associatie van Betaalrekening naar Rekeninghouder en een associatie van Spaarrekening naar Rekeninghouder. Bij iedere spaarrekening hoort een betaalrekening als tegenrekening. Deze relatie wordt getoond door een associatie van Spaarrekening naar Betaalrekening. De betaalrekening heeft in deze associatie de rol tegenrekening. Let ook op de multiplicititeit bij deze associatie aan de kant van Spaarrekening. Deze is * omdat, hoewel bij iedere spaarrekening precies één betaalrekening hoort, een betaalrekening kan dienen als tegenrekening voor nul of meer spaarrekeningen.



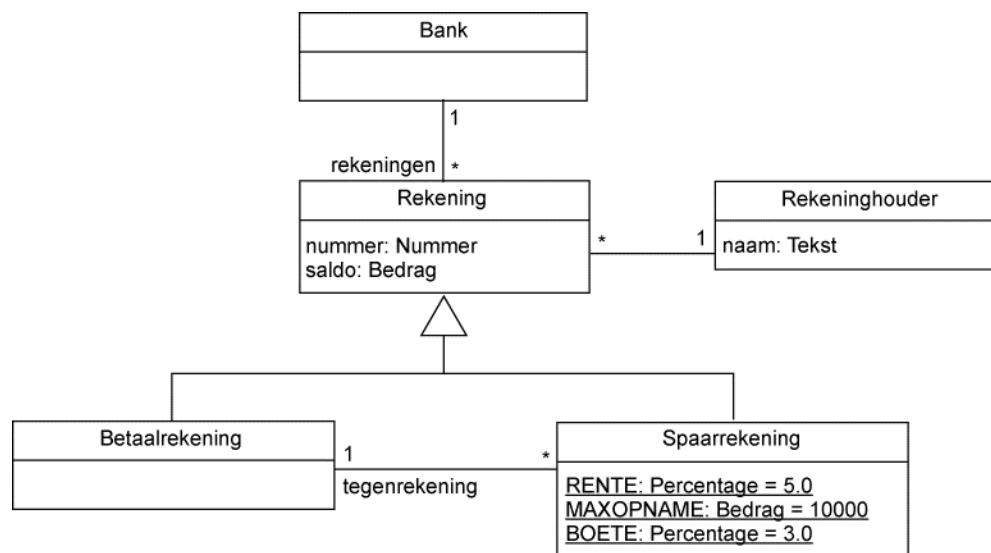
FIGUUR 1.13 Domeinmodel voor de bank, versie 1

Merk op dat tijdens het opstellen van het domeinmodel nog niet is gekozen voor een bepaalde representatie van de attributen. We gebruiken daarom algemene aanduidingen als Tekst, Nummer, Bedrag en Percentage. Maar deze hadden nu ook nog achterwege gelaten kunnen worden.

Tegen welk eerder genoemd principe van een goed programma zondigt dit domeinmodel en wat zou daar aan te doen zijn?

Dit domeinmodel zondigt tegen het principe van lokaliteit. De klassen Betaalrekening en Spaarrekening zijn voor een groot deel gelijk. Keuzen voor een representatie van een nummer en een bedrag zullen in beide klassen zichtbaar zijn en als we een dergelijke doublure kunnen vermijden, dan moeten we dat doen. Bovendien delen beide klassen de associatie met Rekeninghouder.

In dit geval kan dat eenvoudig worden verholpen door de overeenkomstige delen van beide klassen in een superklasse Rekening onder te brengen. Uiteraard handhaven we Betaalrekening en Spaarrekening als subklassen. Uit de productomschrijving volgt immers dat deze wel degelijk zullen verschillen. Figuur 1.14 toont het nieuwe ontwerp.



FIGUUR 1.14 Domeinmodel voor de bank, versie 2

Opmerking

Misschien vraagt u zich af of in het uiteindelijke programma de betaalrekeningen en de spaarrekeningen niet gescheiden moeten blijven, zodat de klasse Bank in de implementatie uiteindelijk twee attributen zal krijgen: betaalrekeningen en spaarrekeningen. Dit is niet handig en we zullen het dan ook niet doen. Waarom dat zo is, zult u pas na de volgende leereenheid begrijpen.

3.3 OPSTELLEN ONTWERPMODEL

Het klassendiagram van figuur 1.14 gaf het domeinmodel. De volgende stap is het ontwikkelen van het ontwerpmodel. In dit model worden methoden toegevoegd aan de klassen en krijgen associaties een richting. Ook kan tijdens het verdere ontwerp blijken dat er nog meer attributen nodig zijn.

We gaan nu alle gebruiksmogelijkheden van de bank na. Ieder van die mogelijkheden zal leiden tot een methode in de klasse Bank. Als we namelijk iets met een rekening willen doen, dan geven we daar een opdracht voor aan de bank en niet direct aan een rekening, net als in het echte betalingsverkeer. Een gebruiker communiceert met de bank alleen via rekeningnummers en niet via instanties van rekeningen. De bank is verantwoordelijk voor het juist toepassen van de regels voor het betalingsverkeer en het beheren van de rekeningen. Het is niet gewenst dat gebruikers direct operaties op rekeningen uitvoeren, omdat de bank dan niet kan controleren of dat wel volgens de regels gebeurt.

Figuur 1.15 toont de klasse Bank met voor iedere gebruiksmogelijkheid een methode. Alleen het verzetten van de datum is nog niet opgenomen. Alle rekeningen worden door de gebruiker aangeduid door hun rekeningnummer; het is de verantwoordelijkheid van de bank om uit te zoeken welk type rekening bij een rekeningnummer hoort, en om de juiste acties daarmee uit te voeren.

Bank
maakBetaalrekening(naam: Tekst): Nummer maakSparrekening(naam: Tekst, tegen: Nummer): Nummer stort(nummer: Nummer, bedrag: Bedrag) neemOp(nummer: Nummer, bedrag: Bedrag) maakOver(van: Nummer, naar: Nummer, bedrag: Bedrag) geefSaldo(nummer: Nummer): Bedrag

FIGUUR 1.15 Klasse Bank met zijn methoden

Opmerking

Bij het ontwerp gaan we uit van de situatie waarbij geen fouten optreden. We negeren dus alle mogelijk uitzonderingsgevallen.

We gaan nu verder als volgt te werk.

- Voor elke gebruiksmogelijkheid (in dit geval dus voor elke methode van Bank) gaan we na welke acties er voor nodig zijn, met de nadruk op creatie van objecten en noodzakelijke interactie met andere objecten.
- Door deze acties nader te analyseren, kunnen we achterhalen welke constructoren en methoden de klassen uit figuur 1.14 nodig hebben.

maakBetaalrekening

De methode maakBetaalrekening wordt aangeroepen als de gebruiker aangeeft een nieuwe betaalrekening te willen openen. Daarvoor dient de gebruiker zijn naam op te geven. Het resultaat van deze methode is dat de bank beschikking heeft gekregen over een nieuwe betaalrekening. De methode geeft het rekeningnummer van de nieuwe rekening terug aan de gebruiker, zodat deze in de toekomst dit nummer kan gebruiken voor zijn bankhandelingen.

OPGAVE 1.12

Som de acties op die in de methode maakBetaalrekening moeten worden uitgevoerd. Om u op weg te helpen, geven we vast de eerste twee:

- genereer een nog ongebruikt rekeningnummer
- creëer een instantie van Betaalrekening.

We gaan nu na wat deze acties betekenen voor het ontwerp.

Een betaalrekening bevat een nummer en een rekeninghouder. Deze informatie moet bij de creatie van een Betaalrekening-object via parameters meegegeven worden. Voor de rekeninghouder zijn er daarbij twee mogelijkheden:

- Als parameter wordt de naam van de rekeninghouder meegegeven. In dit geval dient de constructor van Betaalrekening een instantie van Rekeninghouder te maken.
- Als parameter wordt een instantie van Rekeninghouder meegegeven. In dit geval dient de Bank eerst een instantie van Rekeninghouder te maken.

We kiezen voor de tweede optie. Deze heeft als voordeel dat het dan ook mogelijk is voor de bank om eerst te zoeken of een rekeninghouder al bestaat en zo ja, de betreffende instantie als parameter aan de constructor mee te geven. Dit verhoogt de uitbreidbaarheid van het systeem (we laten het zoeken of de rekeninghouder al bestaat in deze applicatie echter verder buiten beschouwing en gaan ervan uit dat de rekeninghouder altijd gecreëerd moet worden).

We krijgen dan een extra stap in de lijst met acties:

- Creëer een instantie van Rekeninghouder.
- Genereer een nieuw rekeningnummer.
- Creëer een instantie van Betaalrekening.
- Voeg deze instantie toe aan de lijst van rekeningen.
- Geef het rekeningnummer terug.

Dit leidt tot de volgende constructoren in de klassen Rekeninghouder respectievelijk Betaalrekening

Rekeninghouder(naam: Tekst)
Betaalrekening(rekeninghouder: Rekeninghouder, nummer: Nummer)

Misschien denkt u ook al aan een methode om een ongebruikt rekeningnummer te genereren. Dat hoort echter nog niet in deze fase van het ontwerp. Het is een implementatiebeslissing om hiervoor een hulpmethode te maken. Hulpmethoden (die private horen te zijn) worden meestal pas tijdens de implementatie ontworpen.

Deze constructoren worden opgenomen in het klassendiagram (zie figuur 1.16)

maakSpaarrekening De methode maakSpaarrekening wordt op een gelijke wijze verwerkt in het ontwerp.

OPGAVE 1.13

Som de acties op die in de methode maakSpaarrekening moeten worden uitgevoerd. Tot welke constructor(en) en methode(n) in andere klassen zal dit leiden?

stort De volgende methode is stort. De gebruiker stort daarmee geld op een rekening. De bank dient daarvoor achtereenvolgens de volgende stappen uit te voeren:

- Zoek de rekeninginstantie horend bij het rekeningnummer.
- Stort het bedrag op de gevonden rekening.

Om dit mogelijk te maken wordt de klasse Rekening uitgebreid met een methode stort(bedrag: Bedrag) waarmee het saldo van een rekening wordt verhoogd (zie figuur 1.16). Omdat de subklassen deze methoden erven, kan daarmee zowel op een betaal- als op een spaarrekening geld worden gestort.

neemOp Met de methode neemOp neemt de gebruiker geld op van zijn rekening. De acties die de bank daarvoor moet uitvoeren zijn:

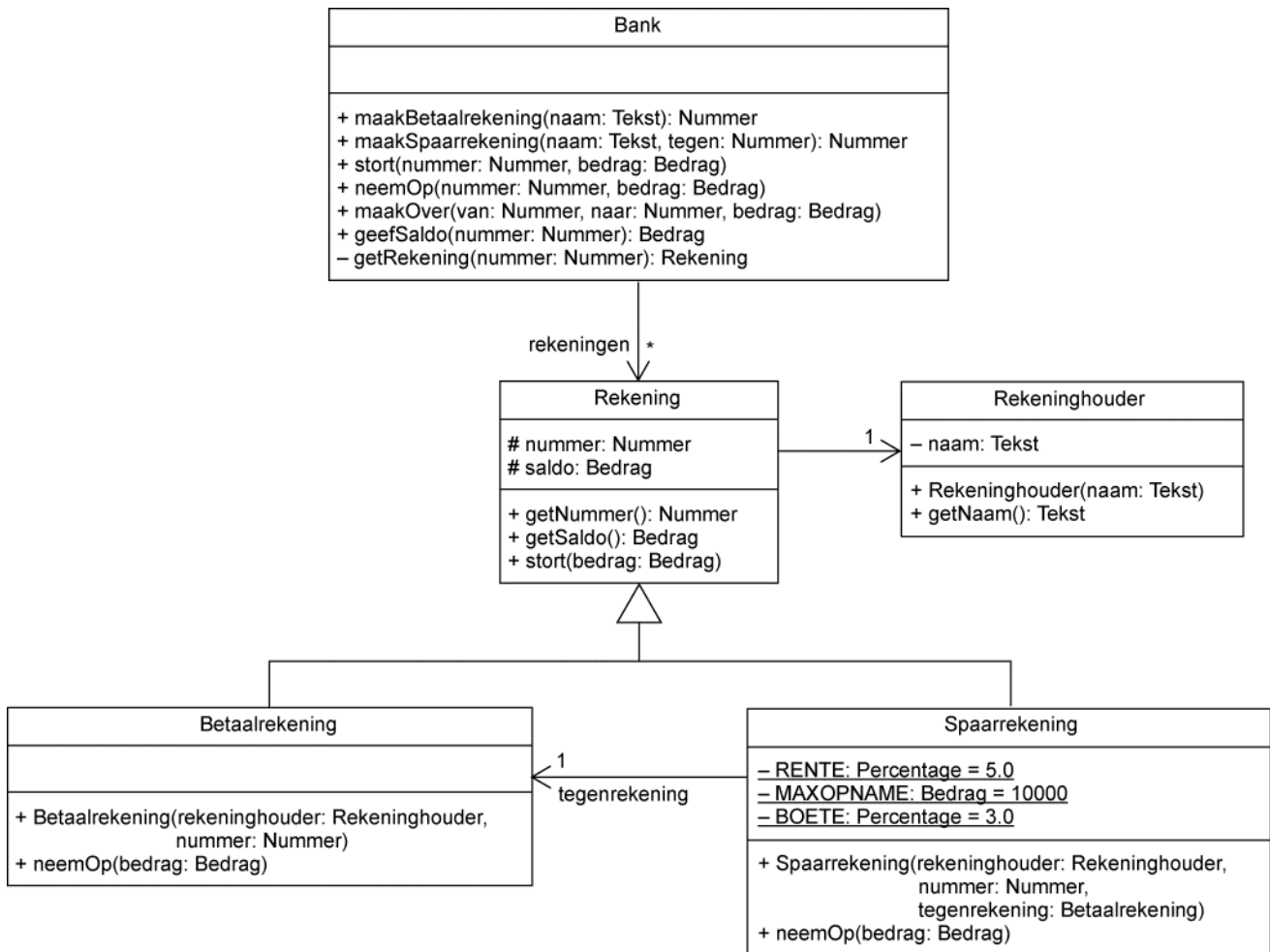
- Zoek de rekeninginstantie horend bij het rekeningnummer.
- Neem het bedrag op van de gevonden rekening.

Opnemen kan zowel van een betaalrekening als van een spaarrekening. Wat er precies bij het opnemen gebeurt, is echter afhankelijk van het type rekening. Beide typen rekening dienen daarom een eigen methode te krijgen, namelijk `neemOp(bedrag: Bedrag)` waarmee geld van de rekening wordt opgenomen (zie figuur 1.16). De implementatie van deze methoden zal verschillen.

<code>maakOver</code>	<p>De methode <code>maakOver</code> is verantwoordelijk voor het overmaken van geld van een betaalrekening naar een andere rekening. De acties die de bank moet uitvoeren zijn:</p> <ul style="list-style-type: none">– Zoek de rekeninginstanties bij de opgegeven rekeningnummers. De instantie van de rekening waarvandaan wordt overgemaakt moet een betaalrekening zijn.– Neem het bedrag op van de ene rekening.– Stort het bedrag op de andere rekening. <p>De methoden die we hiervoor nodig hebben, <code>neemOp</code> bij <code>Betaalrekening</code> en <code>stort</code> bij <code>Rekening</code>, zijn beiden al beschikbaar.</p>
<code>geefSaldo</code>	<p>De methode <code>geefSaldo</code> van de bank vraagt het saldo op van een rekening, zowel voor een betaal- als een spaarrekening. De acties van de bank daarvoor zijn:</p> <ul style="list-style-type: none">– Zoek de rekeninginstantie bij het opgegeven rekeningnummer.– Vraag aan de rekening zijn saldo.– Geef het saldo terug. <p>Omdat de methoden om het saldo op te vragen voor de betaal- en spaarrekening identiek zijn, en de benodigde informatie in de superklasse <code>Rekening</code> aanwezig is, kan de klasse <code>Rekening</code> worden uitgebreid met een methode <code>getSaldo</code> die het saldo teruggeeft.</p>

Figuur 1.16 toont een nieuwe versie van het klassendiagram, waarin alle genoemde constructoren en methoden opgenomen zijn. Hierin zijn naast de besproken constructoren en methoden, bij enkele klassen ook `get`-methoden toegevoegd. Het is namelijk gebruikelijk om bij een attribuut een `get`-methode te maken waarmee de waarde van het attribuut opgevraagd kan worden (tenzij dat om een of andere reden onwenselijk is; maar daar is hier geen sprake van).

Verder ziet u dat bij alle attributen en methoden toegangsspecificaties zijn toegevoegd. We willen normaal gesproken attributen zoveel mogelijk `private` maken om te voorkomen dat er van buitenaf ongewenste operaties op kunnen worden uitgevoerd. De attributen `nummer` en `saldo` van `Rekening` zijn `protected` gemaakt om ervoor te zorgen dat de subklassen `Betaalrekening` en `Spaarrekening` deze attributen wel direct kunnen gebruiken.



FIGUUR 1.16 Klassendiagram voor de bank, versie 3

We moeten nog een gebruiksmogelijkheid bekijken, namelijk het verzetten van de datum.

Wat moet er allemaal gebeuren als de gebruiker de datum verandert?
Kijk naar de productomschrijving!

De gebruiker kan steeds, na een aantal acties, de datum vooruit zetten. Als de nieuwe datum in een nieuwe maand ligt, moet voor alle spaarrekeningen de opgebouwde rente over de tussenliggende maanden berekend en onthouden worden. Ligt de datum in een nieuw jaar, dan moet de rente over het afgelopen jaar (of over de afgelopen jaren) ook worden bijgeschreven. Bovendien moet dan het bedrag dat in het 'huidige' jaar is opgenomen, weer op nul worden gezet. Er zijn immers in dat jaar nog geen transacties op de spaarrekening geweest.

Ook hier moeten we ons eerst afvragen, welke klasse voor welke van deze taken verantwoordelijkheid krijgt.

- Welke klasse beheert het tijdsverloop in de simulatie?
- Welke klasse coördineert de acties die voor de spaarrekeningen ondernomen moeten worden?
- Welke klasse voert die acties daadwerkelijk uit?

Het beheren van het tijdsverloop valt niet onder het beheer van de rekeningen en is dus geen taak voor de bank. We hebben dus behoefte aan een nieuwe klasse, die we Tijdsbeheer noemen. Het coördineren van de acties voor de spaarrekeningen die uit het verlopen van de tijd voortkomen is wel een taak voor de klasse Bank. Het uitvoeren ervan hoort thuis bij de klasse Spaarrekening.

Probeer te bedenken hoe het proces, dat wordt gestart door een wijziging van de datum, moet verlopen.

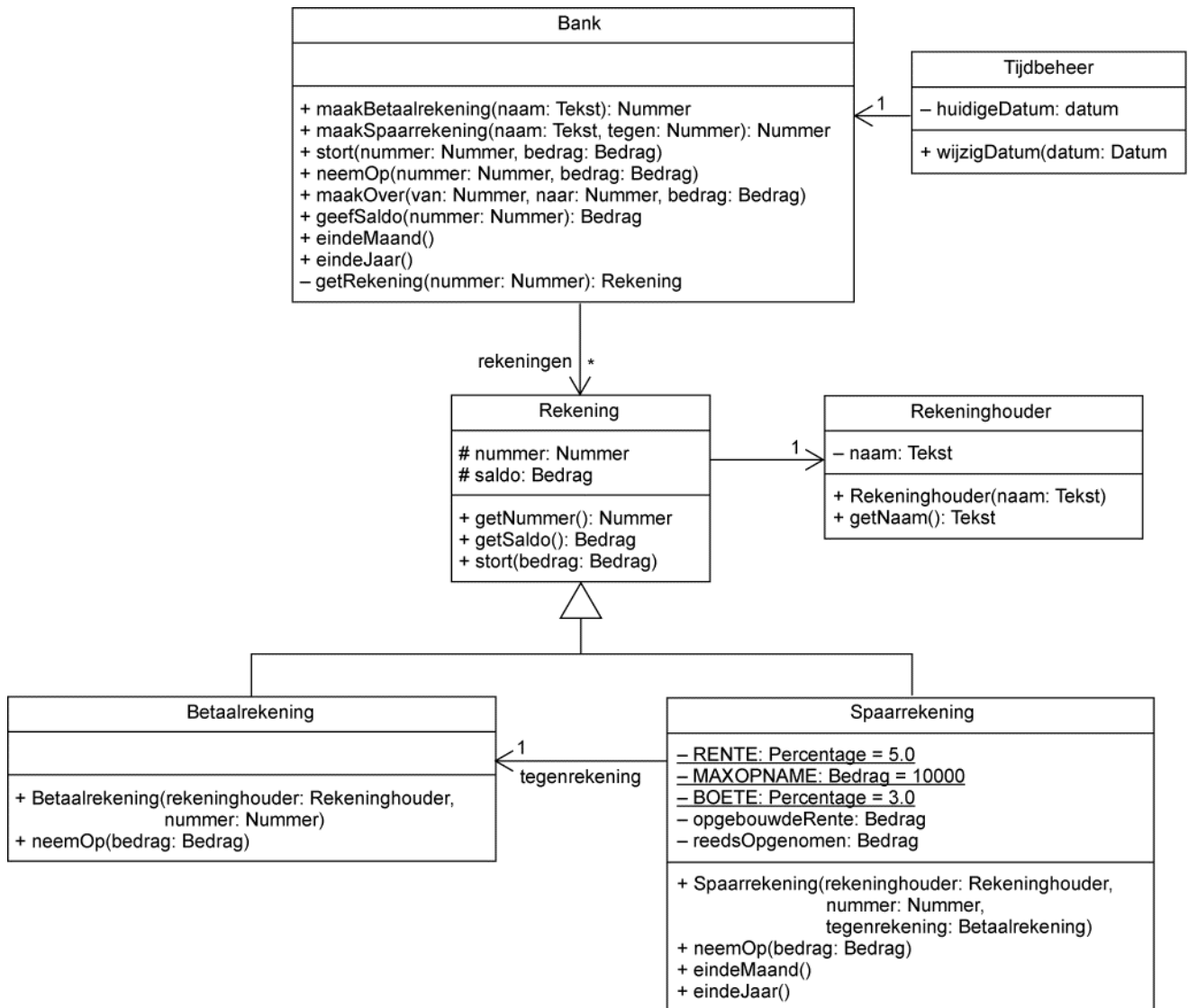
De klasse Tijdsbeheer bevat een attribuut dat de laatst opgegeven datum bijhoudt. Verder krijgt deze klasse een methode `wijzigDatum` die aangeroepen wordt met een nieuwe datum. Deze methode zet de klok maand voor maand vooruit, tot de nieuwe datum is aangebroken. Deze nieuwe datum wordt dan opgeslagen als huidige datum.

Aan het eind van iedere gesimuleerde maand wordt een methode `eindeMaand` aangeroepen op de bank; aan het eind van een gesimuleerd jaar wordt een methode `eindeJaar` aangeroepen. Omdat de datum willekeurig ver vooruit gezet kan worden, kunnen beide methoden verschillende keren worden aangeroepen.

De methode `eindeMaand` van Bank roept zelf dan weer voor alle spaarrekeningen hun methode `eindeMaand` aan. Deze methode berekent de rente over de zojuist verstreken maand. Bij het einde van een gesimuleerd jaar, gebeurt iets dergelijks: nu roept de bank voor alle spaarrekeningen de methode `eindeJaar` aan. Deze schrijft de rente over het afgelopen jaar bij.

We kunnen nu het klassendiagram gaan aanpassen. Om de beschreven procedure goed uit te kunnen voeren, heeft de klasse Spaarrekening een attribuut nodig voor de opgebouwde rente over het lopende jaar, die nog niet is bijgeschreven. Ook nemen we een attribuut op dat bijhoudt hoeveel geld er al is opgenomen in het huidige jaar; dit wordt in de methode `eindeJaar` weer op nul gezet. We hadden deze attributen ook al eerder kunnen introduceren omdat al uit de productomschrijving volgt dat ze nodig zullen zijn, maar pas bij het uitwerken van deze gebruiksmogelijkheid blijken ze ook echt relevant.

De wijzigingen in het ontwerp zijn getoond in figuur 1.17. Omdat een instantie van de klasse Tijdsbeheer de bank opdracht geeft maanden en jaren af te sluiten, moet de klasse Tijdsbeheer de klasse Bank kennen.



FIGUUR 1.17 Klassendiagram voor de bank, versie 4

OPGAVE 1.14

Een alternatief ontwerp voor het verwerken van het verzetten van de datum is als volgt. Tijdbeheer roept een methode `wijzigDatum` aan op de bank met de oude en nieuwe datum als parameter. Een methode met dezelfde signatuur wordt vervolgens aangeroepen op alle spaarrekeningen. In de spaarrekening zelf wordt pas het verstrijken van de tijd gesimuleerd.

Welke van de twee ontwerpen (het oorspronkelijke of het alternatieve) verkiest u en waarom?

Als laatste moet een gebruikersinterface aan het systeem worden toegevoegd. De overeenkomstige klasse heeft een associatie met zowel Bank, om de functies van bank aan te roepen, als met Tijdbeheer, om de datum te kunnen verzetten. We tonen geen nieuw klassendiagram.

Het ontwerp van dit systeem is hiermee afgerond.

We houden ons in deze leereenheid niet bezig met het omzetten van dit ontwerp in een implementatie. Wel kunnen we kort de stappen noemen die nog gedaan moeten worden.

- De gebruikersinterface moet worden ontworpen.
- Van iedere klasse moet de interface gespecificeerd worden: constructoren, publieke attributen en methoden. Van iedere methode wordt de signatuur gegeven en wordt omschreven wat de methode doet en/of welke waarde deze berekent.

Merk op dat hetgeen in figuur 1.17 getoond is, *niet* die interface is.

Figuur 1.17 bevat immers ook de private attributen (en één private methode) die in een ontwerp onontbeerlijk zijn maar in een interface-specificatie van een implementatie niet thuishoren. Bovendien ontbreken in figuur 1.17 de omschrijvingen van de methoden (wat doet de methode?). Een programmeur moet zowel over het klassendiagram als over de interfacespecificatie kunnen beschikken.

- Dan moet iedere klasse gecodeerd en getest worden. Het kan handig zijn om daarbij voor sommige klassen een aparte testomgeving te maken.

Een implementatie van dit ontwerp vindt u bij de bouwstenen in het project Le01Bank.

OPDRACHT 1.15

- a Open het project Le01Bank.

Start de applicatie (methode main staat in de klasse BankFrame). Open enige rekeningen en voer wat transacties uit. Verzet de datum naar een volgend jaar en bekijk het saldo op de spaarrekeningen.

In welk opzichten schiet deze applicatie ernstig tekort?

- b Bekijk de attributen en methoden van alle klassen uit het programma. Zijn er verschillen met het ontwerp?

SAMENVATTING

Paragraaf 1

Wat is een goed programma? Deze vraag kan eigenlijk alleen zinvol beantwoord worden als we niet alleen kijken naar kleine programma's maar ook naar grote. We hebben een mogelijke fasering bekeken voor het ontwikkelen van een programma, die bestaat uit een voortraject, een analysefase, een constructiefase en een afronding van het project:

- Tijdens het voortraject wordt een probleem gesignaleerd en ontstaat het plan een informatiesysteem te ontwikkelen om dat probleem op te lossen.
- Tijdens de analysefase wordt vastgesteld wat dat systeem precies moet doen en wordt een projectplan gemaakt.
- De constructie bestaat uit een aantal deelstappen: in elke stap kan bijvoorbeeld een deel van de gebruiksmogelijkheden van het systeem gerealiseerd worden.

- Iedere constructiestap omvat het ontwerp, de implementatie en het testen van de toevoegingen aan het systeem.
- In de afrondingsfase wordt het systeem als geheel door de gebruikers getest.

Als het systeem voltooid is, gaat het de onderhoudsfase in. Grote systemen gaan vaak jaren mee en behoeven voortdurende aanpassingen. Een goed programma is correct (het voldoet aan de specificatie), robuust (het is bestand tegen gebruikersfouten en onverwachte situaties), het heeft begrijpelijke code die makkelijk te wijzigen en uit te breiden is en het springt efficiënt om met processortijd en geheugenruimte. Tot slot is het wenselijk om delen van de code te kunnen hergebruiken in andere toepassingen.

Moderne projectaanpakken kenmerken zich door het iteratief doorlopen van de stappen analyse, ontwerp, implementatie en testen. Daarmee wordt de uiteindelijke functionaliteit incrementeel ontwikkeld. De opdrachtgever heeft zo steeds een werkend systeem ter beschikking.

Bepaalde eigenschappen van de programmacode kunnen het onderhoud vergemakkelijken. Daartoe horen goed commentaar, goed gekozen namen en zo eenvoudig mogelijke onderdelen. Voor een objectgeoriënteerd programma zijn daarnaast vooral de volgende twee eigenschappen van belang.

- *Gescheiden verantwoordelijkheden*: elke klasse, methode en zelfs opdracht moet bij voorkeur één gemakkelijk te omschrijven verantwoordelijkheid in het geheel hebben.
- *Lokaliteit*: iedere verantwoordelijkheid van het systeem als geheel moet bij voorkeur op één plek gerealiseerd zijn, zodat wijziging van die verantwoordelijkheid op zo min mogelijk plaatsen in het programma tot wijziging van de code leidt. Ook kleine programma's willen we liefst aan deze eisen laten voldoen.

Paragraaf 2

Bij het ontwerpen van programma's maken we gebruik van diagramtechnieken uit de unified modeling language (UML). Een klas-sendiagram toont de statische structuur van het programma: de klassen met hun attributen en methoden, en de relaties tussen de klassen. We bekeken twee soorten relaties:

- de overervingrelatie of (in UML-termen): generalisatie
- de associatie, waarbij instanties van de betrokken klassen elkaar kennen.

Bij associaties worden vaak rollen en multipliciteiten getoond.

Een objectdiagram toont de waarde van instanties op een gegeven moment en de links tussen deze instanties.

Paragraaf 3

In paragraaf 3 is als voorbeeld van een klein programma een eenvoudige banksimulatie ontworpen.

ZELFTOETS

- 1
 - a Noem vijf eisen waaraan een goed programma moet voldoen.
 - b Welke eigenschappen van de code zijn daarbij van belang?

- 2 Gegeven is de volgende productomschrijving van een snoepmachine. Een snoepmachine bevat 12 dispensers (houders van artikelen) en een betaalmechanisme. Iedere dispenser bevat maximaal 20 artikelen met dezelfde prijs. Deze dispensers kunnen worden (bij)gevuld met nieuwe artikelen.

Een gebruiker kan het tegoed van de automaat verhogen. Het tegoed wordt bijgehouden in het betaalmechanisme. Dit betaalmechanisme kan geen geld teruggeven.

Een gebruiker van de automaat kan met behulp van het nummer van de dispenser het gewenste artikel kopen. Als het tegoed in het betaalmechanisme voldoende is, wordt het tegoed verlaagd en het artikel uitgeleverd.

 - a Welke klassen kunt u direct uit de productomschrijving halen. Teken een domeinmodel (zonder methoden).
 - b Geef de gebruiksmogelijkheden van het systeem.
 - c Stel een ontwerpmodel op.

TERUGKOPPELING

1 Uitwerking van de opgaven

- 1.1
 - a U heeft in Objectgeoriënteerd programmeren onder meer gebruik gemaakt van klassen uit de packages `java.lang` (`Random`, `Math` en natuurlijk ook `String` en de verpakingsklassen), `javax.swing` (bijvoorbeeld `JButton`, `TextField` en `JLabel`), `java.awt` (`Color`), `java.util` (`ArrayList`, `GregorianCalendar`) en `java.text` (`DecimalFormat`, `SimpleDateFormat`).
Daarnaast hebt u ook gebruik gemaakt van klassen die door het cursusteam waren geschreven, bijvoorbeeld de klassen uit de `verkiezingenpackage`.
 - b Om een klasse te kunnen gebruiken, moest u de interface van die klasse kennen: de publieke attributen (meestal constanten) en de methoden plus een beschrijving van hun betekenis of werking. Deze werd beschreven in de specificatie. De beschrijving was soms niet volledig. Zo wordt binnen de API Specification bijvoorbeeld niet duidelijk uitgelegd welk coördinatenstelsel Swing gebruikt. Ook zult u vast wel eens de beschrijving van een methode verkeerd begrepen hebben, zodat deze heel iets anders bleek te doen dan u gedacht had. Eigenlijk werd u in zulke gevallen dus al geconfronteerd met een probleem dat hoort bij programmeren in het groot, namelijk de noodzaak voor een goede communicatie tussen de programmeurs van verschillende onderdelen.
- 1.2 Voorbeelden van wijzigingen zijn:
 - Er komt een nieuw type container bij.
 - Er wordt een nieuw tariefsysteem ingevoerd: er wordt bijvoorbeeld een nieuw type korting ingevoerd dat voorheen niet bestond.
 - Er komt een nieuwe Java-versie, waardoor bepaalde functies met behulp van standaardklassen gerealiseerd kunnen worden. Men besluit het systeem daaraan aan te passen om zo de hoeveelheid intern te onderhouden code te verkleinen.
 Een paar voorbeelden die nieuwe gebruiksmogelijkheden nodig of gewenst maken:
 - Het bedrijf besluit niet alleen containers maar ook schepen te gaan verhuren.
 - Een deel van het personeel van de verhuurder wordt ingezet bij het vervoer van de lege containers. Het bedrijf bedenkt dat het inroosteren van dat personeel ook best door het systeem ondersteund zou kunnen worden.
- 1.3 Twee zeer voor de hand liggende eigenschappen hebben we al in de inleiding genoemd: een programma wordt begrijpelijker als de programmeur betekenisvolle namen kiest en de programmacode voorziet van duidelijk verklarend commentaar. Verder is ook een standaardlay-out van het programma van belang (bijvoorbeeld de accolades altijd op dezelfde manier plaatsen en ieder blok twee spaties laten inspringen).
- 1.4 Definitie van constanten bevordert ook de begrijpelijkheid van de code en vermindert het risico van onopgemerkte tikfouten: het abusievelijk vervangen van 12 door 112 leidt niet tot een foutmelding, maar het abusievelijk vervangen van DOZIJN door DDOZIJN wel.

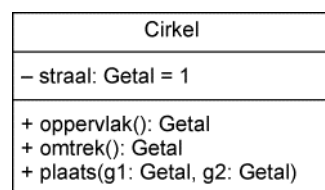
- 1.5 De volgende specificatie benadrukt dat dit een ontwerpmodel is en gebruikt daarom niet de typeaanduiding boolean en de standaardwaarde false:

– verkrijgbaar: Waarheidwaarde = onwaar

Een alternatief is

– verkrijgbaar: boolean = false

- 1.6 Figuur 1.18 toont de gevraagde klasse Cirkel.

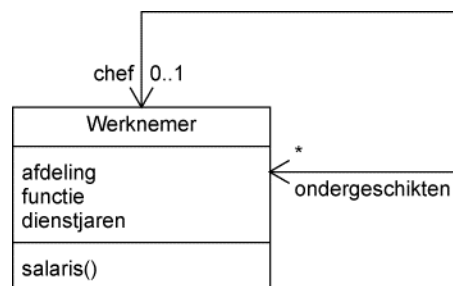


FIGUUR 1.18 De klasse Cirkel

- 1.7 Het betreft een tweezijdige associatie, met als rolnamen bijvoorbeeld chef en ondergeschikten. Werknemers hebben in principe één chef, maar omdat de hiërarchie niet tot in het oneindige doorloopt, moet er ook minstens één werknemer zijn zonder chef. De multipliciteit van die rol is dus 0..1.

Een werknemer heeft 0 of meer ondergeschikten: de multipliciteit van die rol is dus *.

Figuur 1.19 toon het klassendiagram.



FIGUUR 1.19 De klasse Werknemer

- 1.8 Klasse A heeft een attribuut y, dat bijvoorbeeld van type B[] of van type ArrayList is: er horen verschillende instanties van B bij iedere A. Klasse B heeft een attribuut x van type A: er hoort precies één instantie van A bij iedere B.
- 1.9 a Volgens het klassendiagram horen bij iedere klant (dus ook bij bedrijfsklanten en privé-klanten) nul of meer orders. In deze figuur zien we twee klanten: bij klant bk horen drie orders en bij klant pk hoort één order. Dat valt allebei onder nul of meer. Bovendien is de associatie tussen Klant en Order eenzijdig. Dat betekent dat alle links ook eenzijdig moeten zijn, en dat is ook zo.

b Een multipliciteitsaanduiding bij de associatie tussen twee klassen geeft aan hoeveel links er tussen de bijbehorende instanties kunnen zijn. Een link tussen twee objecten is er of is er niet. Vanuit de implementatie bezien kunnen er meer links tussen dezelfde objecten zijn (object a heeft twee attributen die beide hetzelfde object b als waarde hebben), maar in UML geven we dat niet weer.

1.10 We hebben de volgende lijst gebruiksmogelijkheden:

- Maak een nieuwe betaalrekening.
- maak een nieuwe spaarrekening.
- Stort een bedrag op de rekening met gegeven nummer.
- Neem een bedrag op van de rekening met gegeven nummer .
- Maak een bedrag over van de rekening met gegeven nummer naar een andere rekening met gegeven nummer.
- Geef het saldo van de rekening met gegeven nummer.
- Wijzig datum (altijd naar later).

Bij de gebruiksmogelijkheden stort, neem op, maak over en geef saldo wordt geen onderscheid gemaakt tussen betaal- en spaarrekeningen. De gebruiker geeft alleen het nummer op van de rekening waarop de operatie moet worden uitgevoerd. Het is de verantwoordelijkheid van het systeem om te bepalen of de operatie op het desbetreffende type rekening is toegestaan en wat er dan precies moet gaan gebeuren. Het is echter niet fout als u bij het opsommen van de gebruiksmogelijkheden wel onderscheid heeft gemaakt tussen de rekeningtypen.

1.11 – De klasse Bank beheert alle rekeningen en regelt het gehele betalingsverkeer.

- De klasse Betaalrekening beheert het saldo van een rekeninghouder volgens de regels voor betaalrekeningen.
- De klasse Spaarrekening beheert het saldo van een rekeninghouder volgens de regels voor spaarrekeningen.
- De klasse Rekeninghouder beheert de gegevens van een rekeninghouder.

1.12 De volledige lijst van acties die de methode maakBetaalrekening moet uitvoeren, is:

- Genereer een nog ongebruikt rekeningnummer.
- Creëer een instantie van Betaalrekening.
- Voeg deze instantie toe aan de lijst van rekeningen.
- Geef het rekeningnummer terug.

1.13 De acties zijn voor een groot deel gelijk aan de acties voor het maken van een betaalrekening. Extra is het gebruik van de tegenrekening:

- Creëer een instantie van Rekeninghouder (ook hier gaan we er voor de eenvoud vanuit dat de rekeninghouder wordt gecreëerd in plaats van opgezocht).
- Zoek de instantie van de tegenrekening aan de hand van het tegenrekeningnummer (dit moet een betaalrekening zijn).
- Genereer een rekeningnummer.
- Creëer een instantie van Spaarrekening.
- Voeg deze instantie toe aan de lijst van rekeningen.
- Geef het rekeningnummer terug.

Dit leidt tot de volgende constructor (in de klasse `Spaarrekening`):

```
Spaarrekening(rekeninghouder: Rekeninghouder,
              nummer: Nummer,
              tegenrekening: Betaalrekening)
```

In de omschrijving staat ook de stap om een rekeninginstantie te zoeken bij een rekeningnummer. We kunnen nu al voorzien dat ook andere operaties op de bank deze stap gaan gebruiken. De bank accepteert namelijk alleen rekeningnummers als invoer, maar om de operaties uit te voeren dienen methoden op rekeninginstanties te worden uitgevoerd. Het is de verantwoordelijkheid van de bank om bij een rekeningnummer de juiste rekening op te zoeken. Het is daarom zinvol om de klasse bank uit te breiden met de methode

```
getRekening(nummer: Nummer): Rekening
```

die gegeven een rekeningnummer de rekeninginstantie teruggeeft. Deze methode moet echter wel de toegangsspecificatie `private` krijgen. We moeten namelijk voorkomen dat gebruikers van de bank direct de beschikking krijgen over rekeninginstanties. Het is niet fout als u deze methode nog niet opneemt in het ontwerp. Deze methode kan namelijk ook beschouwd worden als een implementatieoplossing.

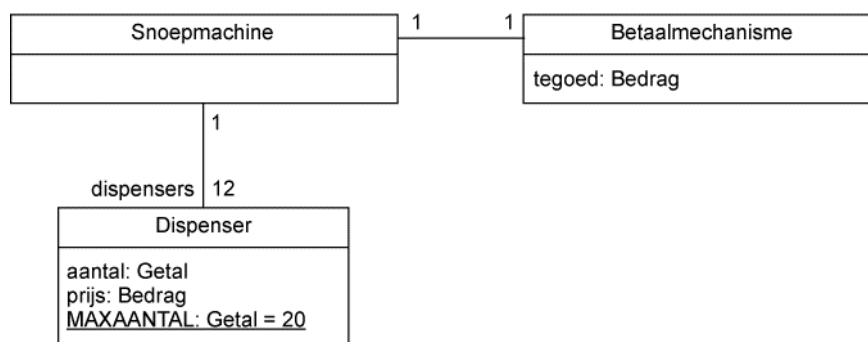
- 1.14 In het alternatieve ontwerp simuleert de klasse `Spaarrekening` het verstrijken van de tijd. Dit is echter een taak die niet onder de verantwoordelijkheid van deze klasse hoort, maar die typisch thuishoort bij `Tijdbeheer`.
In dit ontwerp is daardoor de lokaliteit aangetast omdat een aspect van het systeem (het simuleren van het tijdsverloop) nu over meer klassen is verdeeld. Ook is de scheiding van verantwoordelijkheden aangetast omdat `Bank` en `Spaarrekening` nu meer doen dan alleen rekeningen beheren respectievelijk representeren.
Het oorspronkelijke ontwerp is daarom beter.
- 1.15
 - a De applicatie werkt volgens de productomschrijving. U zult echter merken dat het programma niet robuust is. Bij ongeldige invoer zal het programma exceptions opgooien. Later in de cursus zult u leren hoe deze afgevangen kunnen worden. Een andere ernstige tekortkoming is het feit dat alle rekeningen weg zijn als het programma gesloten wordt. Pas later in de cursus leert u hoe dat voorkomen kan worden.
 - b Er zijn een aantal verschillen die illustratief zijn voor de overgang van een ontwerp naar een implementatie.
In het programma zijn representaties gekozen. `Nummer` is geïmplementeerd als `int`, `Bedrag` en `Percentage` zijn geïmplementeerd als `double`, `Tekst` als `String` en `Datum` als `GregorianCalendar`. Het attribuut `rekeningen` van de klasse `Bank` is geïmplementeerd met behulp van de API-klasse `ArrayList`, waarin gemakkelijk een lijst van objecten kan worden bijgehouden.

Verder heeft de methode neemOp in zowel Betaalrekening als Spaarrekening een boolean terugkeerwaarde gekregen om aan te geven of het opnemen geslaagd is. Bovendien heeft de klasse Rekening een methode neemOp, die vrijwel niets doet en die ook niet in het ontwerp voorkwam. In leereenheid 2 over overerving zullen we de noodzaak van deze methode duidelijk maken.

Van de klasse BankFrame was geen ontwerp gemaakt. Deze klasse heeft veel private methoden, zowel voor het opbouwen van de interface als voor event-handling. Deze methoden horen mogelijk in een implementatiemodel van de klassen thuis, maar zeker niet in het ontwerp.

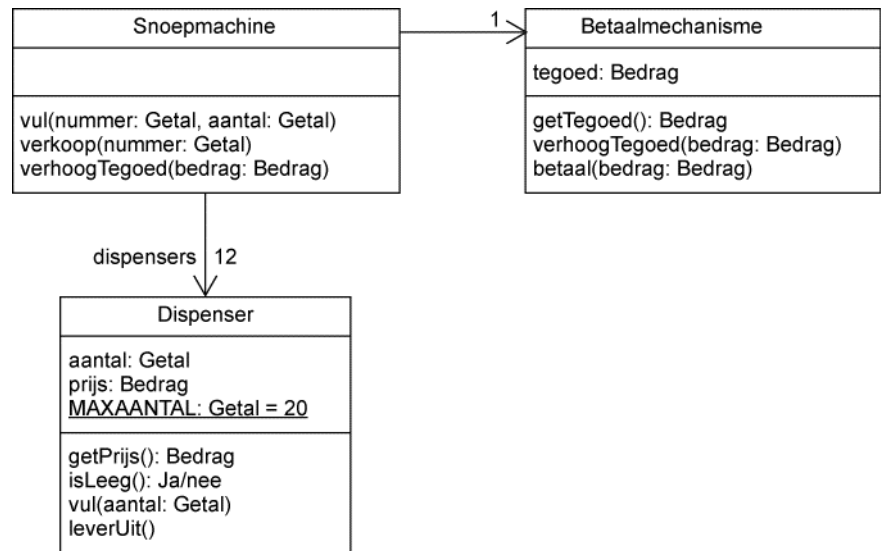
2 Uitwerking van de zelftoets

- 1
 - a Een programma moet correct, robuust, makkelijk te wijzigen en makkelijk uit te breiden zijn. Verder is het gewenst dat onderdelen herbruikbaar zijn. Slechts in enkele gevallen is efficiëntie van doorslaggevend belang.
 - b De volgende eisen aan de code zijn genoemd.
 - Een programma waarin de namen goed zijn gekozen en dat van geschikt commentaar is voorzien, is beter te begrijpen en dus makkelijker te wijzigen.
 - Eenvoud van klassen en methoden (niet al te groot, geen diepe nesting) maakt code begrijpelijker en dus makkelijker te wijzigen.
 - Ieder onderdeel van de code moet slechts één verantwoordelijkheid hebben (gescheiden verantwoordelijkheden).
 - Iedere verantwoordelijkheid moet zoveel mogelijk op één plek in de code gerealiseerd zijn (lokaliteit).
 - De onderlinge afhankelijkheid van klassen moet zo beperkt mogelijk zijn (lage koppeling).
- 2
 - a Uit de productomschrijving volgen drie klassen: Snoepmachine, Dispenser en Betaalmechanisme. Het domeinmodel van de snoepmachine is gegeven in figuur 1.20.



FIGUUR 1.20 Klassendiagram snoepmachine, versie 1

- b De gebruiksmogelijkheden zijn:
- vul de dispenser met gegeven nummer met een aantal artikelen
 - koop een artikel uit dispenser met gegeven nummer
 - verhoog het betaaltegoed.
- c Een mogelijk ontwerpmodel is gegeven in figuur 1.21.



FIGUUR 1.21 Klassendiagram snoepmachine, versie 2

Uitleg:

Iedere gebruiksmogelijkheid leidt tot een methode in de snoepmachine.

- Met de methode `vul` wordt de dispenser met gegeven nummer gevuld. Deze methode zoekt de juiste dispenserinstantie en vult deze (methode `vul` in `Dispenser`).
- De methode `verkoop` wordt aangeroepen als een gebruiker een artikel uit de dispenser met gegeven nummer wil kopen. Deze methode zoekt de juiste dispenserinstantie en vraagt daaraan of deze nog artikelen bevat en wat de prijs van een artikel is (methoden `isLeeg` en `getPrijs` in `Dispenser`). Aan het betaalmechanisme wordt dan het tegoed gevraagd om te bepalen of er genoeg geld betaald is (methode `getTegoed` in `Betaalmechanisme`). Als hieruit blijkt dat de koop mogelijk is wordt het artikel door de dispenser uitgeleverd (methode `leverUit` in `Dispenser`) en wordt voor het artikel betaald (methode `betaal` in `Betaalmechanisme`).
- Met de methode `verhoogTegoed` wordt het betaaltegoed in de snoepmachine verhoogd. Omdat het `Betaalmechanisme` het tegoed bijhoudt moet de verhoging aan het `Betaalmechanisme` worden doorgegeven (methode `verhoogTegoed` in `Betaalmechanisme`).

Overerving (1)

Introductie 55

Leerkern 56

- 1 Specialisatie en generalisatie 56
- 2 Functionaliteit aan een klasse toevoegen 58
 - 2.1 Toegangsspecificaties 59
 - 2.2 Definitie van subklassen 61
 - 2.3 Constructie van instanties van een subklasse 62
 - 2.4 Overerving en het typesysteem 66
- 3 Herdefinitie 72
 - 3.1 Herdefinitie en overloading van methoden 72
 - 3.2 Dynamische binding 75
 - 3.3 Herdefinitie van attributen 78
 - 3.4 Het sleutelwoord super 79
 - 3.5 Verbieden van subklassen en herdefinitie methoden 80
- 4 Een toepassing: uitbreiding van de bank 81

Samenvatting 84

Zelftoets 86

Terugkoppeling 89

- 1 Uitwerking van de opgaven 89
- 2 Uitwerking van de zelftoets 95

Overerving (1)

INTRODUCTIE

In de cursus Objectgeoriënteerd programmeren hebt u kennisgemaakt met de basisprincipes van het mechanisme van overerving. De kracht van dit mechanisme is daarna in die cursus op verschillende manieren getoond.

Een nieuwe applicatie met een grafische gebruikersinterface kunnen we maken door een eigen klasse te definiëren als subklasse van de klasse JFrame. Daarmee heeft deze eigen klasse onmiddellijk veel functionaliteit die niet geprogrammeerd hoeft te worden. De methoden van de superklasse JFrame en de superklassen daarvan (Frame, Window, Container, Component en Object) staan door overerving meteen tot onze beschikking.

We hebben ook kunnen zien dat de verschillende componenten die we op de gebruikersinterface kunnen plaatsen, zoals knoppen, labels en tekstvelden, allemaal eigenschappen hebben als kleur, afmeting en positie op scherm. Deze gemeenschappelijke eigenschappen en de methoden om deze te veranderen, zijn niet voor iedere componentklasse (JButton, JLabel, JTextField, ...) opnieuw gecodeerd. Ze zijn eenmalig geprogrammeerd in een superklasse waarvan al deze componenten erven.

De vluchtige behandeling van het onderwerp in Objectgeoriënteerd programmeren liet nog vele vragen onbeantwoord. Hoe ziet de definitie van een subklasse er precies uit? Tot welke attributen en methoden van de superklasse heeft een subklasse toegang? Mogen we in een methode uit een subklasse van JFrame, direct naar de private attributen van de klasse JFrame verwijzen?

Hoe gaat de constructie van een instantie van een subklasse? Hoe worden de attributen geïnitieerd die in de superklasse gedefinieerd zijn? En hoe zit het eigenlijk met typen? Mag bijvoorbeeld een waarde van type JButton worden toegekend aan een variabele van type Object? En mag daar dan nog een methode setSize op worden aangeroepen? Op al deze vragen zullen we een antwoord geven.

We beginnen deze leereenheid met een algemene inleiding waarin we een paar mogelijke manieren bekijken waarop overerving een rol kan spelen in het ontwerp van een programma. In de volgende twee paragrafen onderzoeken we de technische aspecten van overerving. In paragraaf 2 beperken we ons tot subklassen die de functionaliteit van de superklasse alleen uitbreiden. In paragraaf 3 bekijken we subklassen die deze functionaliteit ook veranderen. In de volgende leereenheid volgt het praktische werk; we zullen daar enkele toepassingen van overerving construeren.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u

- de syntaxis van een subklassedefinitie kent
- weet wat de toegangsspecificatie ‘protected’ inhoudt en wanneer het gebruik daarvan zinvol is
- weet hoe de constructie van instanties van een subklasse gaat en welke rol de constructor van de superklasse daarin speelt
- de constructoraanroepen `super()` en `this()` kunt gebruiken
- weet welke toekenningen tussen variabelen van verschillende typen zijn toegestaan, en weet wanneer deze veilig of onveilig zijn
- weet wat type casting is en weet wanneer expliciete casting nodig is
- het onderscheid begrijpt tussen het gedeclareerde en het actuele type van een variabele
- de operator `instanceof` kunt gebruiken
- weet wat herdefinitie van methoden inhoudt en wat het verschil is met overloading
- weet wat herdefinitie van attributen inhoudt
- weet hoe de binding verloopt bij de aanroep van geherdefinieerde methoden
- van een gegeven klasse een subklasse kunt definiëren die de functionaliteit van die klasse op een vooraf gespecificeerde manier uitbreidt en/of wijzigt
- weet hoe de herdefinitie van een methode of klasse kan worden voorkomen
- de betekenis kent van de volgende kernbegrippen: overerving, subklasse, superklasse, generalisatie, specialisatie, `super`, `protected`, gedeclareerd type, actueel type, `upcast`, `downcast`, herdefinitie, binding, dynamische binding, `final`.

Studeeraanwijzingen

De studielast van deze leereenheid bedraagt circa 8 uur.

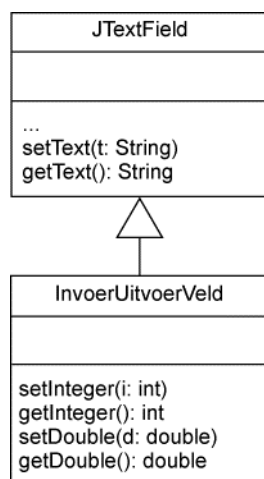
LEERKERN**1 Specialisatie en generalisatie**

Het mechanisme van overerving wordt beschouwd als een van de belangrijkste kenmerken van objectgeoriënteerde talen. Het verhoogt het gemak waarmee objectgeoriënteerde programma's uitgebreid kunnen worden. In de vorige leereenheid hebben we al aangegeven dat dit een wenselijke eigenschap is.

Voordat in de paragrafen 2 en 3 de technische details van het mechanisme behandeld worden, onderzoeken we eerst enkele manieren waarop het gebruikt kan worden. We bekijken daartoe een aantal voorbeelden.

Voorbeeld

Stel, u wilt graag de beschikking hebben over een klasse InvoerUitvoerVeld, die u kunt gebruiken om niet alleen tekstwaarden van het scherm te lezen en naar het scherm te schrijven, maar ook int- en double-waarden. U definieert daarvoor een subklasse van de bestaande klasse JTextField en voegt de methoden getInteger, setInteger, getDouble en setDouble toe, die respectievelijk een int-waarde inlezen, een int-waarde tonen, een double-waarde inlezen en een double-waarde tonen (zie figuur 2.1). De klasse InvoerUitvoerVeld beschikt daarnaast door overerving over alle methoden van JTextField, zoals getText en setText om tekstwaarden in te lezen en te tonen, en over alle methoden om eigenschappen van het veld te zetten.



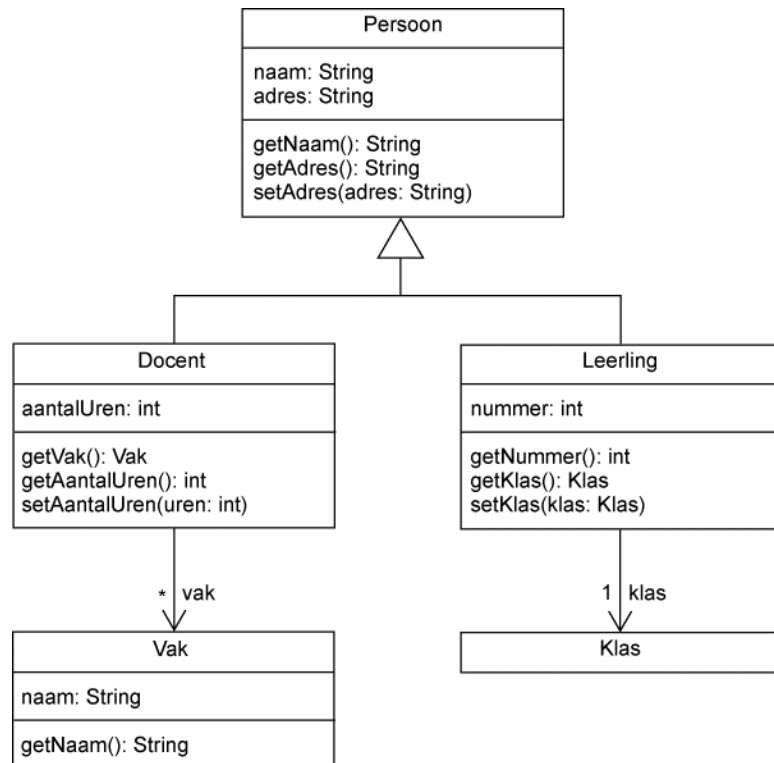
FIGUUR 2.1 Voorbeeld van specialisatie

In dit voorbeeld wordt de functionaliteit van een bestaande klasse uitgebreid door een subklasse te definiëren. We spreken wel van *specialisatie*: de superklasse vormt het uitgangspunt, de programmeur bedenkt er een subklasse bij.

Specialisatie

Voorbeeld

In een systeem voor de administratie van een school komen leraren en leerlingen voor. Van leraren moet worden bijgehouden: naam, adres, het gedoceerde vak en het aantal uren dat wordt lesgegeven. Van leerlingen moet worden bijgehouden: leerlingnummer, naam, adres en de klas waartoe de leerling hoort. De gemeenschappelijke gegevens naam en adres kunnen worden ondergebracht in een superklasse Persoon, de rest wordt ondergebracht in subklassen. Figuur 2.2 toont een mogelijk implementatiemodel met enkele voor de hand liggende methoden.



FIGUUR 2.2 Voorbeeld van generalisatie

Generalisatie

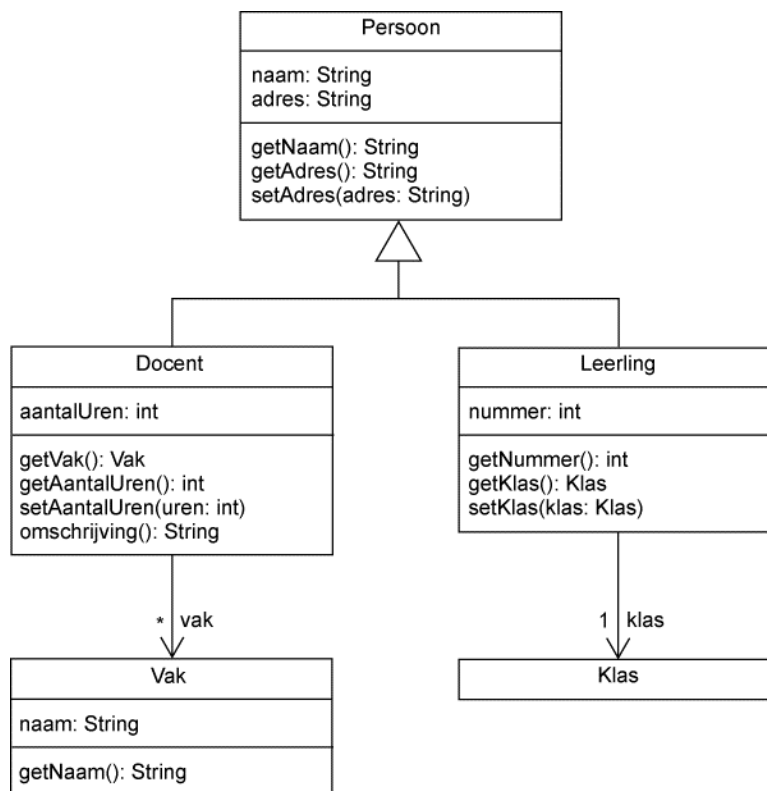
In dit voorbeeld worden eerst twee klassen ontworpen, vervolgens worden de gemeenschappelijke delen daaruit gelicht en ondergebracht in een superklasse. We spreken wel van *generalisatie*: de programmeur definieert een superklasse bij twee eerder ontworpen (sub)klassen.

In deze twee voorbeelden werd de functionaliteit van de superklasse in de subklasse(n) alleen maar uitgebreid, maar niet gewijzigd. Wijziging van de functionaliteit kan echter ook nodig zijn. Dit kan worden bereikt door bepaalde methoden van een superklasse in de subklasse te herdefiniëren. Herdefinitie zal verder worden behandeld in paragraaf 3.

2 Functionaliteit aan een klasse toevoegen

In deze paragraaf zullen we laten zien hoe in een subklasse de functionaliteit van een superklasse kan worden uitgebreid en welke consequenties dit heeft voor de toegang tot attributen en methoden.

We gaan uit van het tweede voorbeeld uit paragraaf 1, waarin de gemeenschappelijke elementen van de klassen *Docent* en *Leerling* worden ondergebracht in een superklasse *Persoon*. Figuur 2.3 toont nogmaals het klassendiagram. Aan de klasse *Docent* is een methode omschrijving toegevoegd; deze levert een *String*-representatie van het object, bijvoorbeeld “van Asperen, Zonnewei 45, Natuurkunde, 22 uur”.



FIGUUR 2.3 Een eenvoudige klassenhiërarchie

We gaan onderzoeken hoe een dergelijke hiërarchie gedefinieerd moet worden.

OPGAVE 2.1

Teken instanties van de klassen Docent en Leerling, waarin waarden voor alle attributen zijn opgenomen (verzin die waarden zelf).

Belangrijk!

Opgave 2.1 illustreert een uiterst belangrijk punt. De overervingrelatie is een relatie tussen *klassen* en niet tussen *objecten*. Als een subklasse zoals Docent geïntanceerd wordt, is het resultaat de creatie en constructie van één object (in dit geval van type Docent). Dit object is weliswaar gebaseerd op verschillende klassendefinities (die van Docent en Persoon), maar bestaat niet uit verschillende instanties. De creatie van een instantie van type Docent leidt dus *niet* tot een aparte instantie van type Persoon waarin de naam en het adres van de docent zijn opgenomen.

2.1 TOEGANGSSPECIFICATIES

We houden ons eerst bezig met de definitie van de superklasse. Tot nu toe hebben we attributen van een klasse bijna altijd *private* gemaakt, zodat instanties van andere klassen er geen toegang toe hebben. Nu moeten we ons, bij het ontwerpen van een klasse, afvragen of er subklassen van gemaakt zullen worden. Is dat niet het geval, dan zullen we de attributen ook nu *private* maken. In het andere geval moeten we per attribuut over de toegangsspecificatie beslissen.

Er zijn de volgende mogelijkheden.

- Het attribuut krijgt toegangsspecificatie *protected*, zodat het toegankelijk is voor de subclasses (en ook voor alle andere klassen binnen dezelfde package). Hiervan hebben we in leereenheid 1 al een voorbeeld gezien in de klasse *Rekening* (zie figuur 1.16). We zullen deze oplossing in het vervolg regelmatig kiezen.
- Het attribuut krijgt de toegangsspecificatie *private* maar wel set- en/of get-methoden. Deze oplossing is de beste als we controle willen houden over de toegankelijkheid. Zo zouden we het attribuut naam van de klasse *Persoon* *private* kunnen maken omdat die naam nooit mag veranderen. Dit attribuut heeft daarom wel een get- maar geen set-methode.
- Het attribuut krijgt toegangsspecificatie *private* en geen set- en get-methoden. Het is dan dus ook voor de subclasses ontoegankelijk. Dit geldt bijvoorbeeld voor de attributen van de klasse *JFrame*, die vanuit de subclasses die we definiëren niet toegankelijk zijn.

We herhalen nog eens de mogelijke toegangsspecificaties:

private

- Een attribuut of methode met toegangsspecificatie *private* is alleen toegankelijk vanuit programmacode die tot dezelfde klassendefinitie behoort. Private attributen en methoden zijn dus *niet* toegankelijk vanuit code die tot een subclassendefinitie behoort.

package

- Een attribuut of methode met toegangsspecificatie *package* is toegankelijk vanuit programmacode die behoort tot een klassendefinitie binnen dezelfde package. Dit is de standaard die geldt voor attributen en methoden zonder expliciete aanduiding van een toegangsspecificatie. Java gebruikt hier niet het sleutelwoord *package*; een attribuutdeclaratie als

Fout

```
package int i = 10;
```

is dus niet juist. In plaats daarvan moet gewoon geschreven worden

```
int i = 10;
```

protected

- Een attribuut of methode met toegangsspecificatie *protected* is toegankelijk voor alle code uit dezelfde package én voor alle subclasses van de klasse, ongeacht in welke package die staan.

public

- Een attribuut of methode met toegangsspecificatie *public* is zonder beperkingen toegankelijk.

De toegangsspecificatie *protected* is soms ruimer dan we zouden willen: *protected* attributen zijn niet alleen toegankelijk voor subclasses, maar ook voor alle andere klassen binnen dezelfde package. Maken we alle attributen in de superklasse *private*, dan zijn we verplicht soms extra toegangsmethoden te definiëren die we eigenlijk niet nodig hebben. In de praktijk zullen we attributen in de superklasse soms *private* en soms *protected* maken.

OPGAVE 2.2

Geef een volledige implementatie in Java van de klasse *Persoon* uit figuur 2.3. Zorg dat het attribuut *adres* bij constructie de waarde "onbekend" krijgt. Geef het attribuut naam de toegangsspecificatie *private*, en adres toegangsspecificatie *protected*.

OPGAVE 2.3

Waarom is de volgende definitie van de methode omschrijving uit de klasse Docent onjuist? Hoe moet de definitie wel luiden?

```
public String omschrijving() {           // Fout!
    return naam + "\n" +
        adres + "\n" +
        vak.getNaam() + "\n" +
        aantalUren;
}
```

2.2 DEFINITIE VAN SUBKLASSEN

De syntaxis van de definitie van een subklasse is eenvoudig: we voegen aan de kop het sleutelwoord `extends` toe, gevolgd door de naam van de superklasse.

Een subklassendefinitie ziet er dus als volgt uit:

```
[toegang] class klassennaam extends superklassennaam
    blok
```

Net als bij iedere andere klasse, kan de romp declaraties van constanten en attributen bevatten en definities van constructoren en methoden.

Let op

In Java kan een klasse van slechts één superklasse erven, dus achter `extends` mag maar één naam van een superklasse staan. We spreken daarom van enkelvoudige overerving. In een aantal andere talen is het wel mogelijk dat een klasse erft van meer dan één superklasse; in dat geval spreken we van meervoudige overerving.

OPGAVE 2.4

- Probeer een implementatie te geven van de klasse `Leerling` uit figuur 2.3. Geef `Leerling` een constructor met nummer en naam als parameters. Welk probleem komt u tegen bij het opstellen van de code voor deze constructor?
- Kunt u een oplossing bedenken voor dit probleem?

Klasse Object

Iedere klasse erft impliciet van de klasse `Object`; met andere woorden, iedere klasse is, direct of indirect, een subklasse van de klasse `Object`. Dit volgt ook uit de beschrijving in de API van de klasse `Object`:

```
Class Object is the root of the class hierarchy. Every class
has Object as a superclass.
```

We hoeven hiervoor zelf niets te doen; de toevoeging `extends Object` in de kop van de klasse is dus overbodig. `Object` bevat methoden die voor alle klassen van belang zijn. Twee methoden van de klasse `Object` zullen we geregeld in onze programma's gebruiken, namelijk de methoden `toString` en `equals`. De signatuur van `toString` luidt:

methode `toString` **public** String `toString()`

Deze methode geeft een stringrepresentatie van het object. Deze methode wordt geërfd door alle subklassen van Object, dus door iedere klasse. De string die we van de geërfde methode terugkrijgen geeft echter niet veel bruikbare informatie over het object. Wanneer bijvoorbeeld de methode wordt aangeroepen op een spaarrekeningobject uit de banksimulatie van leereenheid 1 levert dat als antwoord:

```
bank.Spaarrekening@11a698a
```

Hierin zien we de naam van de package (bank), de naam van de klasse (Spaarrekening) en een nummer dat aan het object is toegekend (dit heet een hashcode). We zijn meestal meer geïnteresseerd in de waarden van de attributen van een object. Daarom wordt deze methode vaak gedefinieerd, zie paragraaf 3.1.

methode equals

De signatuur van de methode equals van Object luidt:

```
public boolean equals(Object obj)
```

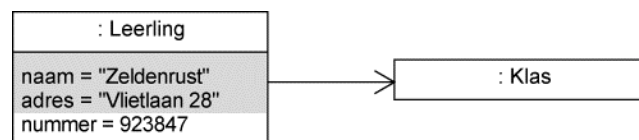
Deze methode vergelijkt het object met een ander object, en geeft true terug wanneer de objecten gelijk zijn, anders false. De implementatie in de klasse Object, die standaard door alle klassen geërfd wordt, is:

```
public boolean equals(Object obj) {  
    return this == obj;  
}
```

Deze test dus alleen of parameter obj verwijst naar dezelfde instantie als het object waarop de methode wordt aangeroepen. Dit is meestal niet wat we willen. We willen namelijk dat deze methode true oplevert als de inhoud van de objecten gelijk is. De methode equals van String bijvoorbeeld levert true op als beide stringobjecten dezelfde reeks karakters bevatten; het mogen daarbij best verschillende stringinstanties zijn. De methode equals is daartoe in de klasse String gedefinieerd. In paragraaf 3.1 zal worden getoond hoe we de methode equals kunnen herdefiniëren voor onze eigen klassen.

2.3 CONSTRUCTIE VAN INSTANTIES VAN EEN SUBKLASSE

Figuur 2.4 toont een objectdiagram van de instantie van Leerling uit de terugkoppeling bij opgave 2.1.



FIGUUR 2.4 Instantie van Leerling

Bovenin staan de attributen die geërfd zijn van Persoon (deze zijn grijs gemaakt). Daaronder staat het attribuut nummer en verder is er de link met een (anoniem) Klas-object. Constructie van het gedeelte van de instantie in het grijs gearceerde stuk wordt overgelaten aan de klasse Persoon: *iedere constructor van Leerling roept om te beginnen de constructor van Persoon aan.*

Dit heeft twee voordelen. Ten eerste kan op die manier bij constructie toch een waarde worden toegekend aan private attributen van de superklasse (zie de terugkoppeling bij opgave 2.4). Ten tweede is het zeker dat het deel dat uit de superklasse afkomstig is, correct geconstrueerd wordt.

Voor de klasse Persoon is dit laatste niet zo van belang: de constructor doet niets meer dan het attribuut naam een waarde geven (zie de terugkoppeling bij opgave 2.2).

Stel nu dat we een subklasse definiëren van een klasse Breuk met attributen teller en noemer. De constructor van Breuk gaat na of de noemer ongelijk 0 is, de andere methoden van Breuk controleren dat niet meer. Als we nu een instantie van de subklasse kunnen construeren zonder eerst de constructor van Breuk aan te roepen, dan kan die test opeens nagelaten worden en werken methoden van Breuk voor instanties van de subklasse mogelijk niet meer correct.

Of stel dat we een subklasse definiëren van een Swing-klasse als JButton. We willen er dan zeker van zijn dat alle attributen van JButton die in de constructor een waarde moeten krijgen, deze ook echt gekregen hebben. Anders werken de methoden van JButton mogelijk niet correct voor instanties van de subklasse.

Constructie van een instantie van de subklasse begint daarom altijd met het laten construeren van het deel dat afkomstig is uit de superklasse, door een constructor van die superklasse. Java kent daarvoor een aparte opdracht.

Aanroep
superklasse-
constructor

Syntaxis

De *aanroep van een constructor van de superklasse* ziet er als volgt uit:

```
super(parameterlijst);
```

De parameters moeten in aantal en type overeenstemmen met die van een constructor uit de superklasse. De aanroep moet voorkomen als *eerste* opdracht in de constructor van een subklasse en is verplicht wanneer de superklasse alleen constructoren met parameters heeft. De aanroep mag worden weggelaten als de superklasse (ook) een parameterloze constructor heeft. Er wordt dan impliciet een aanroep naar die parameterloze constructor toegevoegd. De verwerking van een constructor van een subklasse begint dus altijd met een aanroep naar een constructor van de superklasse.

Voorbeeld

De constructor van de klasse Leerling ziet er dus als volgt uit:

```
public Leerling(int nummer, String naam) {  
    super(naam);  
    this.nummer = nummer;  
}
```

Omdat de klasse Persoon geen parameterloze constructor heeft, mag de opdracht `super(naam)` in dit geval niet worden weggelaten.

OPGAVE 2.5

Geef nu de volledige definitie van de klasse Docent uit figuur 2.3, met twee constructoren: een met als enige parameter de naam van de docent en een met als parameters naam, vak en aantalUren.

Tot slot van deze paragraaf over constructie, kijken we naar een paar details.

Wanneer moet een subklasse een constructor definitie bevatten?

Binnen een gewone klasse hoeft niet altijd een constructor gedefinieerd te worden omdat iedere klasse zonder constructor een standaard parameterloze constructor heeft met een lege romp. Hoe zit dit nu met subklassen? De regels hiervoor zijn als volgt.

Laten we voor het gemak uitgaan van een superklasse A met een subklasse B.

Als de superklasse A een parameterloze constructor heeft, dan hoeft de subklasse B geen constructoren te bevatten. Java voorziet dan in een standaardconstructor, die uitsluitend bestaat uit de aanroep van de parameterloze constructor van A, dus alsof B de volgende constructor bevatte:

```
public B() {
    super();
}
```

Merk op dat die parameterloze constructor van A mogelijk veel werk verzet: het hoeft niet de lege standaardconstructor van A te zijn, maar het kan ook een constructor zijn die binnen A gedefinieerd is. Heeft de superklasse A echter uitsluitend constructoren met parameters, dan moet de subklasse B tenminste één constructor bevatten. De programmeur zal dan immers zelf in een aanroep naar een constructor van A, waarden op moeten geven voor de parameters.

Constructie-
volgorde

De *volgorde van de constructiewerkzaamheden* is als volgt:

- Eerst wordt de constructor van de superklasse aangeroepen.
- Dan worden de attributen geïnitieerd.
- Tot slot wordt de rest van de code uit de constructor uitgevoerd.

Uiteraard kan de superklasse zelf ook weer een superklasse hebben. Wordt er dus bijvoorbeeld een instantie van de Swing-klasse `JTextField` gecreëerd, dan wordt eerst de constructor van de superklasse `JTextComponent` aangeroepen, die op zijn beurt weer als eerste de constructor van `JComponent` aanroept. Dit proces gaat door totdat de constructor van `Object`, de klasse die boven in de klassenhiërarchie staat, is aangeroepen.

Voorbeeld

We geven een voorbeeld van constructie. Dit voorbeeld bevat twee klassen, A en B, waarbij B een subklasse van A is.

```
public class A {
    protected int a = 5;

    public A() {
        System.out.println("constructor A: a = " + a);
        a = 7;
    }

    public int getA() {
        return a;
    }
}
```



```
public class B extends A {
    private int b = a + 10;

    public B() {
        a = 2;
        System.out.println("constructor B: a = " + a +
                           ", b = " + b);
    }

    public int getB() {
        return b;
    }

    public static void main(String[] args) {
        B b = new B();
        System.out.println("main: a = " + b.getA() +
                           ", b = " + b.getB());
    }
}
```

OPGAVE 2.6

Wat is de uitvoer van dit programma?

Omdat eerst de constructor van de superklasse wordt aangeroepen, kan de initialisatiecode van de attributen uit de subklasse gebruik maken van de waarden van de attributen uit de superklasse. Omdat de attributen van de subklasse geïnitieerd worden voordat de romp verwerkt wordt, kunnen we in die romp eventuele standaardwaarden van alle attributen gebruiken en/of vervangen.

Dit is soms handig en het zal ook de reden zijn waarom voor deze verwerkingsvolgorde is gekozen, ondanks het wat onverwachte feit dat verwerking van de constructor van de subklasse nu wordt onderbroken om de attributen te initialiseren (en wel na de aanroep naar de constructor van de superklasse).

In plaats van een aanroep naar de constructor van de superklasse, kan als eerste opdracht van een constructor ook een aanroep naar een andere constructor in *dezelfde* klasse staan. Deze aanroep heeft de vorm:

this(..)

this(parameterlijst);

Dit kan handig zijn wanneer verschillende constructoren nodig zijn.

Voorbeeld

De klasse Point uit de package java.awt representeert een punt in een plat vlak en heeft de attributen x en y die de coördinaten van het punt representeren. Deze klasse heeft 3 constructoren:

- een parameterloze constructor; in dat geval krijgen x en y de default waarde 0
- een constructor met twee waarden (voor x en y) als parameters
- een constructor met een ander punt als parameter (een zogenaamde copy-constructor, die een kopie maakt van het object dat als parameter wordt meegegeven).

Een deel van de code ziet er als volgt uit (merk op dat deze klasse bij uitzondering public attributen heeft):

```

public class Point {
    public int x;
    public int y;

    Point() {
        this(0, 0);
    }

    Point(Point p) {
        this(p.x, p.y);
    }

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ... // de rest van de implementatie van deze klasse
}

```

In de parameterloze constructor en in de constructor met een Point als parameter wordt de derde constructor met twee parameters aangeroepen door middel van `this`. Dit is een voorbeeld van het toepassen van het principe van lokaliteit. De code voor het toekennen van waarden aan attributen is op één plaats opgenomen. Als later deze code moet wijzigen, dan hoeft dat slechts op één plaats te worden doorgevoerd.

2.4 OVERERVING EN HET TYPESYSTEEM

Het typesysteem van Java vereist dat bij een toekenning de variabele aan de linkerkant en de expressie aan de rechterkant van hetzelfde type zijn. Ook bij parameteroverdracht moeten de typen van de formele parameter en de actuele parameter hetzelfde zijn. Als in de formulering van een opdracht die typen niet gelijk zijn, is het in sommige gevallen toch mogelijk de typen gelijk te maken. Dit wordt *type casting* genoemd.

Een voorbeeld is het volgende programmafragment:

```

int i = 5;
double d = i;

```

In de tweede toekenning staat links een variabele van type `double`, en rechts een waarde van het type `int`. Toch is deze toekenning toegestaan, omdat een waarde van het type `int` altijd, zonder verlies van informatie, naar een waarde van type `double` geconverteerd kan worden. We spreken daarom van een *veilige conversie*.

Een veilige conversie wordt automatisch uitgevoerd; de programmeur hoeft er niets voor te doen. Het wordt daarom *impliciete cast* genoemd.

Er bestaan ook *onveilige conversies*. De conversie van `double` naar `int` wordt bijvoorbeeld als onveilig aangemerkt, omdat hier wel verlies van informatie kan optreden. Conversie kan dan toch worden afgedwongen door een *expliciete cast* te gebruiken. Een expliciete cast wordt aangegeven door het gewenste type tussen haakjes (). Een voorbeeld:

```

double d = 5.032;
int i = (int)d;

```

Type casting

Veilige conversie

Impliciete cast

Onveilige conversie

Expliciete cast

Zonder deze expliciete cast is de typeconversie in de tweede toekenning niet toegestaan.

Ook bij het gebruik van referentietypen is het in een aantal gevallen mogelijk om variabelen van verschillende typen aan elkaar toe te kennen (via een toekenningsopdracht of via parameteroverdracht). Alleen variabelen van typen met een directe of indirecte superklasse-subklasse relatie kunnen aan elkaar toegekend worden via type casting. In dit geval zullen we echter niet spreken over conversies, zoals bij primitieve typen, omdat de instanties waar de variabelen (referenties) naar wijzen niet daadwerkelijk veranderen. We onderzoeken deze vorm van casting aan de hand van twee voorbeelden.

Voorbeeld 1 Bekijk de volgende opdrachten, ontleend aan het voorbeeld van de bank:

```
Rekeninghouder rh = new Rekeninghouder("van Ende");
Rekening r = new Betaalrekening(rh, 1004);
```

In de tweede regel staat links een variabele van type Rekening, rechts een waarde van type Betaalrekening. Iedere instantie van de klasse Betaalrekening is door overerving ook een instantie van klasse Rekening, en daarom bevat de instantie van Betaalrekening ook alle attributen en methoden van Rekening. Het is daarom toegestaan een instantie van Betaalrekening toe te kennen aan een variabele van het type Rekening. Er wordt bij deze toekenning een impliciete cast uitgevoerd.

Belangrijk! Het is belangrijk om te beseffen dat er bij deze toekenning geen informatie verloren gaat. Er worden geen gegevens geconverteerd: *een cast naar een super- of subklasse is geen conversie*. De volgende opgave verduidelijkt dit.

OPGAVE 2.7

a Bekijk de opdrachten:

```
int i = 5;
double d = i;
```

Moet de waarde van i een andere interne representatie krijgen om aan d te kunnen worden toegekend?

b Teken een toestandsdiagram (zoals gebruikt in Objectgeoriënteerd programmeren) voor de situatie na afloop van de volgende toekenningen (zie ook figuur 2.3 en de terugkoppelingen bij opgaven 2.2 en 2.4):

```
Persoon persoon1 = new Persoon("van Aspen");
Leerling leerling = new Leerling(1232, "van Aspen");
Persoon persoon2 = leerling;
```

Wordt er hier iets aan de waarde van leerling veranderd bij toekenning van de waarde aan persoon2?

Bij conversie tussen primitieve typen vindt dus een echte omzetting van de waarde plaats; bij toekenning van een instantie van een subklasse aan een variabele van de superklasse blijft de waarde gelijk.

In opgave 2.7b zijn de waarden van `persoon2` en `leerling` aliassen, ondanks het feit dat de ene variabele van het type `Persoon` is en de andere van het type `Leerling`.

Laten we nog eens kijken naar dezelfde opdrachten:

```
Rekeninghouder rh = new Rekeninghouder("van Ende");
Rekening r = new Betaalrekening(rh, 1004);
```

Gedeclareerd type
Actueel type

Van welk type is `r` na de toekenning? Er is feitelijk sprake van twee typen. De variabele `r` is gedeclareerd met als type `Rekening`, maar de toegekende waarde is van type `Betaalrekening`. We spreken ook wel van het *gedecclareerde* en het *actuele* type. Het gedeclareerde type van `r` is `Rekening`, het actuele type van `r` na de toekenning is `Betaalrekening`.

Het actuele type van een object wordt automatisch bij het object opgeslagen in de vorm van een referentie naar de desbetreffende klasse. Deze referentie is één van de attributen van de klasse `Object`, en kan worden opgevraagd met de methode `getClass`.

OPGAVE 2.8

De klasse `Bank` uit leereenheid 1 heeft een attribuut rekeningen met als waarde een lijst waarin instanties van zowel `Betaalrekening` als `Spaarrekening` voorkomen. Figuur 2.5 toont een aanschouwelijke voorstelling van een mogelijke inhoud van deze lijst. `Betaalrekeningen` zijn voorgesteld als cirkels en `spaarrekeningen` als vierkanten



FIGUUR 2.5 Een lijst met rekeningen

Stel nu dat de volgende opdracht, ontleend aan de banksimulatie, twee keer wordt uitgevoerd:

```
Rekening rekening = bank.getRekening(...);
```

De eerste keer is de rekening het tweede element uit de lijst en de tweede keer het vijfde. Wat is in beide gevallen het gedeclareerde en wat het actuele type van de variabele rekening?

Voorbeeld 2

Stel dat we meteen na de toekenning uit het vorige voorbeeld, hetzelfde object ook willen toekennen aan een variabele van type `Betaalrekening`. Dit kan alleen met gebruik van een expliciete cast:

```
Rekeninghouder rh = new Rekeninghouder("van Ende");
Rekening r = new Betaalrekening(rh, 1004);
Betaalrekening b = (Betaalrekening)r;
```

Op grond van de toekenning in de tweede regel weten wij dat het actuele type van `r` `Betaalrekening` is, maar dit is in het algemeen niet uit de code te achterhalen. Bij de toekenning aan `b` kijkt de compiler alleen naar het gedeclareerde type van `r`, en dat is `Rekening`. Omdat niet iedere instantie van `Rekening` ook een instantie van `Betaalrekening` is, is deze toekenning onveilig en dus is er een expliciete cast vereist.

Ook in dit geval leidt die cast, anders dan bij primitieve typen, niet tot een echte omzetting van de waarde. De cast moet beschouwd worden als een belofte van de programmeur aan de compiler: straks, tijdens verwerking, zal het actuele type van `r` Betaalrekening blijken te zijn.

Tijdens verwerking wordt gecontroleerd of de programmeur die belofte gehouden heeft. Is dit niet het geval, dan wordt een `ClassCastException` gegenereerd. De compiler accepteert dus de volgende, incorrecte code. Tijdens verwerking blijkt dat `r` niet van het beloofde type Spaarrekening is, maar van het type Betaalrekening: er volgt een foutmelding

ClassCast-Exception




Fout!

```
Rekeninghouder rh = new Rekeninghouder("Paauw");
Rekening r = new Betaalrekening(rh, 1005);
Spaarrekening s = (Spaarrekening)r;
```

Regels voor casting bij referentietypen

Tabel 2.1 geeft regels voor de toepassing van casts bij het gebruik van referentietypen, waarbij steeds wordt uitgegaan van de toekenning van een object met gedeclareerd type B aan een variabele van het type A

TABEL 2.1 De toekenning van een object van gedeclareerd type B aan een variabele van type A

Relatie tussen klassen	Toekenning
<p>Tussen A en B bestaat geen superklasse-subklasse relatie</p> 	<p><i>Toekenning nooit toegestaan</i></p> <p>Deze fout wordt door de compiler ontdekt.</p>
<p>A is een directe of indirecte superklasse van B</p> 	<p><i>Veilige toekenning</i></p> <p>Een instantie van B heeft zeker ook alle eigenschappen van klasse A. De toekenning is daarom veilig. Er is geen expliciete cast nodig. Omdat er hier sprake is van een toekenning van een type lager in de hiërarchie aan een type hoger in de hiërarchie wordt dit ook wel een <i>upcast</i> genoemd.</p> <p>Voorbeeld:</p> <pre>B b = new B(); A a = b;</pre>
<p>A is een directe of indirecte subklasse van B</p> 	<p><i>Onveilige toekenning</i></p> <p>Het is niet zeker dat een instantie van klasse B alle eigenschappen van klasse A heeft. De toekenning is daarom onveilig en vereist een expliciete cast. Deze kan bij verwerking leiden tot een <code>ClassCastException</code>. Omdat er hier sprake is van een toekenning van een type hoger in de hiërarchie aan een type lager in de hiërarchie wordt dit ook wel een <i>downcast</i> genoemd.</p> <p>Voorbeeld:</p> <pre>B b = new B(); A a = (A)b;</pre>

Veilige toekenning

Upcast

Onveilige toekenning

Downcast

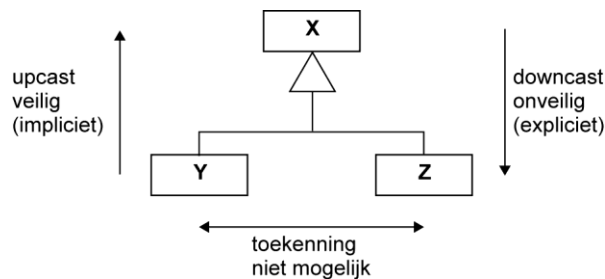
Merk op dat het bij deze regels om statische eigenschappen gaat van de code. Daarom spreken we over de toekenning van een object met *gedecclareerd* type B aan een variabele van *gedecclareerd* type A. Het actuele type van B is uit de code immers niet altijd te achterhalen.

Deze regels zijn niet alleen van toepassing op toekenningsoopdrachten maar ook bij parameteroverdracht, dus op de toekenning van een actuele parameter aan een formele parameter bij een methodeaanroep.

Bij een veilige toekenning is een cast overbodig maar niet verboden. Als B een subklasse is van A, is dus ook het volgende correct Java:

```
B b = new B();
A a = (A)b;
```

Figuur 2.6 toont een samenvatting van de casting-regels.



FIGUUR 2.6 Samenvatting van de regels voor casting

We laten u in de volgende opgaven oefenen met deze regels.

OPGAVE 2.9

a Gegeven de volgende declaraties en toekenningen, gebaseerd op de hiërarchie uit figuur 2.3:

```
Docent docent1 = new Docent("Paauw");
Docent docent2;
Persoon persoon1 = new Docent("Welters");
Persoon persoon2;
```

Geef opdrachten om de waarde van docent1 toe te kennen aan persoon2 en ook om de waarde van persoon1 toe te kennen aan docent2.

b Gegeven is nu ook de volgende declaratie:

```
Leerling leerling;
```

Is er een toekenning mogelijk van de waarde van docent1 aan de variabele leerling?

OPGAVE 2.10

Gegeven zijn de volgende declaraties en toekenningen, gebaseerd op de hiërarchie uit figuur 2.3:

```
Object obwaarde = new Object();
Persoon pwaarde = new Persoon("Aartsen");
Docent dwaarde = new Docent("Paauw");
Leerling lwaarde = new Leerling(224, "van Erkens");
```

Geef voor elk van de volgende programmafragmenten aan, of deze geaccepteerd worden door de compiler en zo ja, of ze dan toch tot foutmeldingen leiden bij verwerking.

- a `Persoon pvar = obwaarde;`
- b `Persoon pvar = (Persoon)obwaarde;`
- c `Persoon pvar = (Persoon)lwaarde;`
- d `Persoon pvar = dwaarde;`
`Docent dvar = pvar;`
- e `Persoon pvar = lwaarde;`
`Docent dvar = (Docent)pvar;`
- f `Object obvar = pwaarde;`
`Docent dvar = (Docent)obvar;`

Cast in expressie

Een cast kan ook voorkomen in een expressie. U moet er in dat geval rekening mee houden, dat de punt van de methodeaanroep een hogere prioriteit heeft dan de cast. We geven een voorbeeld:

Voorbeeld

```
Persoon p = new Docent("Stevens");
Vak v = ((Docent)p).getVak();
```

Het gedeclareerde type van `p` is `Persoon`, het actuele type is `Docent`. We mogen op `p` alleen de methode `getVak` aanroepen als we een cast gebruiken. De klasse `Persoon` heeft immers, in tegenstelling tot de subklasse `Docent`, geen methode `getVak`. Omdat de punt een hogere prioriteit heeft dan de cast, moet er om `(Docent) p` nog een extra paar haakjes staan.

Met behulp van een cast kan de programmeur aangeven tot welke subklasse van de superklasse een bepaalde variabele behoort. Soms moet de programmeur dit echter eerst zelf uitzoeken. Stel bijvoorbeeld dat we een lijst van rekeningen hebben van het type `ArrayList<Rekening>`. Sommige elementen van deze lijst zijn betaalrekeningen, andere zijn spaarrekeningen. We willen de methode `eindeJaar` aanroepen voor alle spaarrekeningen uit deze `arraylist`. Maar wat zijn de spaarrekeningen in de lijst?

Operator instanceof

Java kent een operator die ons kan helpen om daarachter te komen, en wel de operator *instanceof*. De linkeroperand van *instanceof* is een variabele van een referentietype en de rechteroperand is de naam van een klasse. Het resultaat is van het type boolean. De waarde is *true* als het *actuele* type van de instantie links gelijk is aan of een subklasse is van de klasse rechts.

Voorbeeld

In de banksimulatie bevat de klasse `Bank` een methode `eindeJaar`, die voor iedere spaarrekening de methode `eindeJaar` aanroept. We kunnen deze methode als volgt implementeren:

```
public void eindeJaar() {
    for (Rekening r : rekeningen) {
        if (r instanceof Spaarrekening) {
            ((Spaarrekening)r).eindeJaar();
        }
    }
}
```

We testen van iedere rekening van de bank of deze een spaarrekening is. Is dat het geval, dan wordt de methode `eindeJaar` van de klasse `Spaarrekening` aangeroepen. Merk op dat de cast naar `Spaarrekening` niet overbodig is. Als de test slaagt, is het zeker dat deze cast geoorloofd is. De taaldefinitie eist echter nog steeds dat deze er staat; de klasse van het gedeclareerde type van `r`, `Rekening`, bevat namelijk geen methode `eindeJaar`.

3 Herdefinitie

In paragraaf 1 hebben we al aangegeven dat het vaak zinvol is om de functionaliteit van een superklasse niet alleen uit te breiden, maar ook te wijzigen.

3.1 HERDEFINITIE EN OVERLOADING VAN METHODEN

Herdefinitie van een methode

Definitie

Een toegankelijke methode uit een superklasse kan in een subklasse *geherdefinieerd* worden, dat wil zeggen dat de implementatie van de methode vervangen wordt door een eigen implementatie van de subklasse. *De signaturen van de oorspronkelijke en de geherdefinieerde versie moeten identiek zijn.* Dus naast de naam van de methode moeten ook het aantal parameters en het type van deze parameters in beide versies exact gelijk zijn. Ook de terugkeertypen moeten identiek zijn.

Op deze laatste eis (identieke terugkeertypen) is echter een uitzondering: als het terugkeertype van de oorspronkelijke methode een referentietype (een klasse) is, dan mag het terugkeertype van de geherdefinieerde methode een subtype daarvan zijn.

Verder is het volgende relevant: binnen deze cursus is het begrip *signatuur* zo gedefinieerd, dat het terugkeertype van een methode daar toe behoort. In de Java-literatuur is dit echter niet gebruikelijk; daar worden alleen naam en formele parameters tot de signatuur gerekend. Dit heeft invloed op de omschrijving van de begrippen herdefinitie en overloading.

Voorbeeld 1

De methode `toString` uit de klasse `Object` wordt geërfd door alle andere klassen. In paragraaf 2.2 hebben we gezien dat de implementatie van deze methode in de klasse `Object` meestal niet voldoet. Deze implementatie geeft geen nuttige informatie over de waarden van attributen van een object. Daarom wordt de methode `toString` vaak geherdefinieerd door de schrijver van een klasse. Die kan dan zelf bepalen wat belangrijke informatie is om terug te geven. Voor de klasse `Spaarrekening` zou dit de herdefinitie kunnen zijn:

```
public String toString() {
    return "Spaarrekening [nummer: " + nummer +
        ", rekeninghouder: " + rekeninghouder.getNaam() +
        ", tegenrekening: " + tegenrekening.getNummer() +
        ", saldo: " + saldo + "]\n";
}
```

In de klasse `Docent` uit figuur 2.3 kan de methode omschrijving vervangen worden door een herdefinitie van `toString`, met dezelfde implementatie (zie de uitwerking van opgave 2.3). We hebben in paragraaf 2 voor de naam 'omschrijving' gekozen omdat we daar nog geen herdefinities wilden gebruiken.

Voorbeeld 2

De volgende code toont een eenvoudig voorbeeld van een klasse met een subklasse waarin een methode geherdefinieerd is.

```
public class Som1 {
    private int term1 = 0;

    public Som1(int t) {
        term1 = t;
    }

    public int getTerm1() {
        return term1;
    }

    public int plus(int extraTerm) {
        return term1 + extraTerm;
    }
}

public class Som2 extends Som1 {
    private int term2 = 0;

    public Som2(int t1, int t2) {
        super(t1);
        term2 = t2;
    }

    public int plus(int extraTerm) {
        return getTerm1() + term2 + extraTerm;
    }
}
```

OPGAVE 2.11

Wat zijn, na verwerking van de volgende opdrachten, de waarden van s1 en s2? In regel 3 wordt de implementatie van plus uit Som1 gebruikt, en in regel 4 die uit Som2.

```
Som1 t1 = new Som1(5);
Som2 t2 = new Som2(3, 4);
int s1 = t1.plus(2);
int s2 = t2.plus(2);
```

In de definitie aan het begin van de paragraaf wordt vermeld dat alleen een toegankelijke methode uit een superklasse kan worden geherdefinieerd. Herdefinitie wijzigt de functionaliteit die wordt geboden door de superklasse. Alleen functionaliteit die ook daadwerkelijk geboden wordt, kan worden gewijzigd. Het heeft daarom geen zin om van herdefinitie van een methode te spreken wanneer deze methode in de superklasse private is.

Een programmeur weet in het algemeen niet welke private methoden een bepaalde klasse heeft en zou in een subklasse toevallig een methode met dezelfde signatuur op kunnen nemen. Dat mag, maar er is dan geen sprake van een wijziging van geboden functionaliteit en dus is er ook geen sprake van herdefinitie. De methode in de superklasse is immers niet zichtbaar in de subklasse.

De toegang van een geherdefinieerde methode mag niet strenger zijn dan de toegang van de methode in de superklasse. Neem als voorbeeld superklasse Persoon en subklasse Student. Neem aan dat klasse Persoon

een methode `getNummer` heeft met toegang `public` en dat klasse `Student` deze methode heeft met toegang `private`. In de volgende situatie is dan onduidelijk wat er moet gebeuren:

```
Persoon s = new Student();
s.getNummer();
```

Java laat dit dan ook niet toe.

In de definitie is verder benadrukt dat, wil er sprake van herdefinitie zijn, de signaturen van de oorspronkelijke en de geherdefinieerde methode gelijk moeten zijn. Die nadruk is van belang om verwarring te vermijden met een ander verschijnsel in Java, namelijk *overloading*. We gaan hier kort in op het verschil tussen herdefinitie en *overloading*.

Overloading

Van *overloading* is sprake wanneer er – in eerste instantie binnen één klasse – verschillende methoden (of constructoren) zijn gedefinieerd met dezelfde naam, maar met verschillende signaturen.

Ook hier is de eerdergenoemde uitzondering relevant: als alleen het terugkeertype verschilt en het terugkeertype van de ene methode is een subklasse van het terugkeertype van de tweede methode, dan is er geen sprake van *overloading*. Binnen één klasse is dat niet toegestaan.

Voorbeelden

Voorbeelden van *overloading* zijn we in Objectgeoriënteerd programmeren tegengekomen. We hebben gezien dat een klasse verschillende constructoren kan hebben, bijvoorbeeld `JButton()` en `JButton(String label)`. We hebben ook gezien dat de klasse `String` verschillende methoden heeft met de naam `valueOf`, die allemaal hun argument naar een `String` converteren. In de klasse `Math` uit de package `java.lang` komen vier methoden min voor, om het minimum te bepalen van achtereenvolgens twee `int`-waarden, twee `long`-waarden, twee `float`-waarden en twee `double`-waarden. De precieze regels bij *overloading* (Wat mag wel en wat mag niet? Hoe wordt bij een methodeaanroep bepaald welke methode nu precies bedoeld wordt?) zijn echter vrij ingewikkeld; we behandelen ze in deze cursus niet.

De reden om het onderwerp hier toch ter sprake te brengen, is dat *overloading* en herdefinitie dicht bij elkaar kunnen liggen. Definieert u namelijk in een subklasse een methode met dezelfde naam als een methode uit de superklasse, maar met een andere parameterlijst, dan is er sprake van *overloading*. Dat is niet verboden, en meestal zal er ook wel gebeuren wat u al verwachtte – we hebben daarom bijvoorbeeld zonder al te veel plichtplegingen constructoren overladen – maar er is dan *geen* sprake van herdefinitie.

equals moet
geherdefinieerd
worden

We zullen één geval noemen waarin u *overloading* zeker moet vermijden, namelijk bij definitie van eigen `equals`-methoden. Wilt u bijvoorbeeld de klasse `Persoon` een dergelijk methode geven, dan moet deze de volgende signatuur hebben:

```
public boolean equals(Object obj)
```

Het overladen van deze methode, bijvoorbeeld met een definitie

Niet doen!

```
public boolean equals(Persoon p)
```

kan tot onverwachte resultaten leiden. Aan het eind van paragraaf 3.2 zult u begrijpen waarom dat zo is.

OPGAVE 2.12

Geef een volledige implementatie van een methode `equals` voor de klasse `Persoon` (zie de terugkoppeling bij opgave 2.2 voor een definitie van deze klasse).

De implementatie van `equals` uit de terugkoppeling van opgave 2.12 gebruikt `instanceof` om het type te controleren. Dat heeft een onverwacht gevolg. Stel we hebben een instantie `p` van `Persoon` en een instantie `d` van `Docent` met dezelfde waarde voor naam en adres. De aanroep `p.equals(d)` levert `true` op: `d` is immers (ook) een instantie van `Persoon`. Of dit gewenst is, hangt af van hoe gelijkheid geïnterpreteerd wordt. Als naam en adres een persoon uniek bepalen, dan is docent `d` kennelijk dezelfde als persoon `p`. Als de klasse `Docent` geen eigen implementatie van `equals` bevat, levert ook de aanroep `d.equals(p)` `true` op. Dat is zoals het hoort: als `p` gelijk is aan `d`, moet `d` ook gelijk zijn aan `p`. Maar hier zit wel een risico in: als we `Docent` wél een eigen methode `equals` geven die vereist dat het object waarmee vergeleken wordt, ook een instantie is van `Docent`, dan kan het dat `p.equals(d)` `true` is maar `d.equals(p)` `false`.

Het risico op asymmetrie wordt vermeden door gebruik te maken van de methode `getClass` van `Object`, die het actuele type oplevert van het object waarop de methode wordt aangeroepen. Dat type is een instantie van de klasse `Class`. De implementatie ziet er dan als volgt uit.

```
public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }
    String naam2 = ((Persoon)obj).getNaam();
    String adres2 = ((Persoon)obj).getAdres();
    return (naam.equals(naam2) && adres.equals(adres2));
}
```

Met deze implementatie leveren zowel `p.equals(d)` als `d.equals(p)` `false` op: de `Class`-objecten van `d` en `p` zijn niet gelijk.

Binnen de Java-community bestaat geen overeenstemming over welke implementatie de juiste is. De API Specification vermeldt bij `Object` dat een herdefinitie van `equals` symmetrisch hoort te zijn; dat pleit voor gebruik van `getClass`. Echter, de specificatie van `equals` bij `Set` (een interface; zie leereenheid 4) eist implementaties met `instanceof`. Ook in de API zelf wordt veelvuldig `instanceof` gebruikt.

3.2 DYNAMISCHE BINDING

In opgave 2.11 was er tweemaal sprake van een aanroep van de methode `plus`. In het eerste geval werd de methode `plus` aangeroepen op een instantie van `Som1` en werd dus ook de implementatie van `plus` uit `Som1` uitgevoerd. In het tweede geval werd de methode aangeroepen op een instantie van `Som2` en dus werd nu implementatie van `plus` uit `Som2` uitgevoerd.

Met de naam `plus` werd dus in beide gevallen een verschillende implementatie van de methode `plus` verbonden. Het verbinden van een bepaalde implementatie van een methode met een naam heet *binding*.

Binding

Als een methode wordt gedefinieerd, bestaan er in feite in verschillende klassen verschillende implementaties van dezelfde methode. Dan doet zich de vraag voor, hoe Java bij een methodeaanroep vaststelt welke implementatie nu precies bedoeld wordt.

Laten we nog eens kijken naar de lijst met rekeningen van figuur 2.7 en de methode `neemOp` van de klasse `Bank`. Volgens het ontwerp van de bank in leereenheid 1 heeft ieder type rekening een eigen methode `neemOp`, zie figuur 1.16.



FIGUUR 2.7 Een lijst met rekeningen

```
public void neemOp(int nummer, double bedrag) {
    Rekening rekening = getRekening(nummer);
    if (rekening != null) {
        // zoek uit van welk type de gevonden rekening
        // daadwerkelijk is en voer daarop de juiste
        // neemOp methode uit
        ...
    }
}
```

In dit voorbeeld zien we dat binnen de methode `neemOp` van `Bank` de ene keer een `betaalrekening` en een andere keer een `spaarrekening` aan de variabele `rekening` van het type `Rekening` wordt toegewezen. Het feit dat een variabele naar objecten van verschillende klassen kan wijzen, wordt *polymorfisme* genoemd. Dit kan echter niet onbeperkt; aan een variabele van een bepaalde klasse kunnen alleen objecten van zijn subklassen worden toegekend.

Bij het verwerken van de methode is na de toekenning in de eerste regel het actuele type van rekening bekend. Java kijkt tijdens verwerking van deze code naar dit actuele type en bindt dan de daarbij behorende implementatie van de methode `neemOp`. De methode `neemOp` van `Bank` kan daarom als volgt worden geïmplementeerd:

```
public void neemOp(int nummer, double bedrag) {
    Rekening rekening = getRekening(nummer);
    if (rekening != null) {
        rekening.neemOp(bedrag);
    }
}
```

Wat gebeurt er nu bij verwerking van deze regels? Stel dat het tweede element uit de lijst rekeningen wordt toegekend aan de variabele `rekening`. Dit object (een vierkantje in figuur 2.7) is van het type `Spaarrekening`. Het verwerkt de aanroep door de *eigen* implementatie van `neemOp` uit te voeren, dus de implementatie van `neemOp` die in klasse `Spaarrekening` is gedefinieerd.

De volgende keer dat `neemOp` van `Bank` wordt uitgevoerd, wordt aan rekening bijvoorbeeld het vijfde element van de lijst toegekend: een `betaalrekening`. De methode `neemOp` wordt nu dus op een instantie van `Betaalrekening` aangeroepen; ook deze instantie zal de eigen implementatie van `neemOp` verwerken (die uit de klasse `Betaalrekening`).

Dynamische binding

Hoewel in beide gevallen *dezelfde* opdracht `rekening.neemOp(bedrag)` wordt verwerkt, worden er dus tijdens verwerking *verschillende* implementaties van `neemOp` uitgevoerd, afhankelijk van het type object waarop de methode wordt aangeroepen.

Omdat het binden van de aanroep aan een specifieke implementatie pas tijdens de verwerking van de aanroep gebeurt, spreken we van *dynamische binding*.

Het principe van dynamische binding is dat het actuele type van het object waarop een methode wordt aangeroepen, bepaalt welke implementatie van die methode wordt uitgevoerd. Dit type is pas tijdens verwerking van de aanroep bekend. De binding van de methodeaanroep aan een specifieke implementatie gebeurt dan ook pas tijdens verwerking en dus kan dezelfde regel code, zoals `rekening.neemOp(bedrag)`, op verschillende momenten tot de verwerking van verschillende implementaties van de methode `neemOp` leiden. Dynamische binding is in Java het mechanisme om polymorfisme mogelijk te maken.

Dynamische binding maakt het uitbreiden van een klassenhierarchie een stuk eenvoudiger, omdat het aantal plaatsen in de code waar iets veranderd moet worden, beperkt kan blijven. We zullen dat zien in paragraaf 4.

Let op

Het verhaal is hiermee nog niet helemaal rond. De typeringsregels van Java eisen namelijk, *dat ook de superklasse Rekening een implementatie van methode neemOp heeft*. De compiler kijkt namelijk of de methode `neemOp` op het gedeclareerde type `Rekening` kan worden uitgevoerd, en houdt er geen rekening mee dat deze methode eventueel in subklassen is geïmplementeerd. Dat kan alleen als `Rekening` zelf (of één van zijn superklassen) de methode `neemOp` bevat.

We moeten de superklasse `Rekening` daarom een eigen methode `neemOp(double)` geven. Deze methode hoeft niets te doen (kan een lege romp krijgen), want die methode wordt toch in alle subklassen opnieuw gedefinieerd en we zullen alleen maar instanties van die subklassen maken. U hebt deze toevoeging al gezien toen u, in opgave 1.15b, ontwerp en implementatie vergeleek. Deze methode dient ook aan het ontwerp te worden toegevoegd (zie paragraaf 4).

OPGAVE 2.13

Aan het eind van paragraaf 2.4 hebben we gezien, hoe we de methode `eindeJaar` kunnen aanroepen op alle spaarrekeningen uit een lijst van rekeningen. Daarbij moest steeds eerst worden getest of een rekening wel een spaarrekening was.

a Bij een iets ander ontwerp van de klasse `Rekening` hoeven ook de implementaties van de methoden `eindeMaand` en `eindeJaar` geen onderscheid meer te maken tussen de verschillende typen rekeningen. Beschrijf deze wijziging in het ontwerp.

Aanwijzing: neem als uitgangspunt de manier waarop de methode `neemOp` in het ontwerp is opgenomen.

b Geef implementaties van de methoden `eindeMaand` en `eindeJaar` van de klasse `Bank`, gebaseerd op het gewijzigde ontwerp.

In paragraaf 3.1 werd gezegd dat de methode `equals` niet overladen maar geherdefinieerd moet worden. Dit heeft te maken met dynamische binding. Sommige API-klassen maken gebruik van de methode `equals`. Dit geldt bijvoorbeeld voor de klasse `ArrayList`. We kunnen bijvoorbeeld vragen of een `ArrayList` een bepaald object bevat, bijvoorbeeld met de methode `contains(Object o)`; de implementatie van deze methode gebruikt `equals` om objecten te vergelijken.

Stel nu dat u een instantie van `ArrayList` gebruikt om een tabel van personen bij te houden, en dat u de klasse `Persoon` een methode `equals` hebt gegeven waarin is vastgelegd dat twee instanties van `Persoon` gelijk zijn (naar dezelfde ‘echte’ persoon verwijzen) als hun naam en hun adres gelijk zijn. Als u nu wilt weten of een bepaalde instantie `p1` van `Persoon` al in de tabel zit, dan wilt u het antwoord `true` krijgen als de tabel een instantie `p2` van `Persoon` bevat met dezelfde naam en hetzelfde adres, ongeacht of `p1` en `p2` naar hetzelfde object verwijzen. U wilt dus dat de `equals` uit de klasse `Persoon` wordt gebruikt en niet die uit de klasse `Object`. Dit gebeurt alleen, wanneer `equals` van `Persoon` een herdefinitie is van `equals` van `Object`, want alleen dan zal de aanroep van `equals` in de betreffende methode van `ArrayList` worden gebonden aan de `equals` van `Persoon`. Heeft u de methode `equals` in `Persoon` slechts overladen, dan zal het proces van dynamische binding deze versie over het hoofd zien, en de methode `equals` uit `Object` gebruiken.

3.3 HERDEFINITIE VAN ATTRIBUTEN

Tot nu toe hebben we het alleen gehad over herdefinitie van methoden. Over herdefinitie van attributen zullen we kort zijn. Net als bij methoden, kunnen alleen toegankelijke attributen worden geherdefinieerd. Een programmeur kan toevallig in een subklasse een attribuut opnemen met dezelfde naam als een ontoegankelijk (bijvoorbeeld `private`) attribuut uit de superklasse. Daar is niets tegen, maar evenmin is er dan sprake van herdefinitie.

Een toegankelijk attribuut uit een superklasse wordt in een subklasse geherdefinieerd wanneer deze een attribuut bevat met dezelfde naam. Het attribuut uit de superklasse wordt daarmee in de subklasse onzichtbaar. Er is nooit een goede reden om een attribuut op die manier te herdefiniëren en we raden het dan ook ten sterkste af.

Als we in een subklasse een attribuut opnemen met dezelfde naam als een toegankelijk attribuut uit de superklasse, wordt het attribuut van de superklasse binnen de subklasse onzichtbaar. De situatie is vergelijkbaar met declaratie van een lokale variabele met dezelfde naam als een attribuut: binnen de scope van die lokale variabele is het attribuut onzichtbaar. Herdefiniëren we een attribuut van de superklasse in een subklasse, dan is – op precies dezelfde manier – het attribuut van de superklasse binnen de subklasse onzichtbaar.

Bij herdefinitie van toegankelijke attributen krijgen we ook weer te maken met binding. Stel dat we de volgende klassendefinities hebben:

```
public class A {
    public int waarde = 0;

    public A(int w) {
        waarde = w;
    }

    public int getWaarde() {
        return waarde;
    }
}
```

```
public class B extends A {

    public int waarde = 0;

    public B(int w) {
        super(w);
        waarde = getWaarde() + 1;
    }
}
```

Als nu van buitenaf wordt gerefereerd aan het attribuut `waarde`, dan is in dit geval het *gedeclareerde* type bepalend en niet het actuele type. Na verwerking van de opdrachten

```
A a = new B(3);
int w = a.waarde;
```

zal `w` gelijk zijn aan 3 en niet aan 4. Voor attributen biedt Java dus statische binding en niet, zoals bij methoden, dynamische binding.

3.4 HET SLEUTELWOORD SUPER

Het resultaat van de methode `plus` uit `Som2` (uit het voorbeeld in paragraaf 3.2) is gelijk aan dat van de methode `plus` uit `Som1`, verhoogd met `term2`. De geherdefinieerde methode `plus` doet dus eigenlijk hetzelfde als de oorspronkelijke methode `plus`, met nog iets extra's daaraan toegevoegd. Die situatie komt vaker voor: we willen een bepaalde methode van de superklasse niet zozeer wijzigen, maar eigenlijk vooral uitbreiden. Het is dan onwenselijk om in de geherdefinieerde methode de oorspronkelijke code opnieuw op te moeten nemen:

- Het feit dat hetzelfde stuk code op verschillende plaatsen in een systeem voorkomt, druist in tegen het principe van lokaliteit.
- Als de code uit de superklasse referenties bevat aan private attributen, dan kunnen we de code niet eens altijd overnemen. Dit is het probleem waar we ook bij het opstellen van constructoren tegenaan liepen. Het is dus wenselijk om in dergelijke gevallen vanuit de subklasse nog bij de methode uit de superklasse te kunnen.

Hiervoor dient het sleutelwoord *super*. Dit sleutelwoord is te vergelijken met het sleutelwoord *this*, dat verwijst naar het object zelf dat de code verwerkt. Ook het sleutelwoord *super* verwijst naar het object zelf, maar in een methodeaanroep zal bij de binding het zoeken naar de juiste methode nu begonnen worden in de superklasse van de klasse waarin de aanduiding *super* staat en niet, zoals bij *this*, in die klasse zelf.

super
this

Voorbeeld

De methode `plus` van `Som2` kan ook als volgt gedefinieerd worden:

```
public int plus(int extraTerm) {
    return super.plus(extraTerm) + term2;
}
```

In de aanroep `super.plus(extraTerm)` zal `plus` nu gebonden worden aan de implementatie van `plus` uit `Som1`, omdat `Som1` een superklasse is van de klasse waarin de aanduiding *super* staat (`Som2`).

Alweer voor de volledigheid vermelden we ook nog het volgende. Het sleutelwoord *super* kan ook gebruikt worden om naar een attribuut uit een superklasse te verwijzen. Als `a` een toegankelijk attribuut uit een klasse `A` is, dan kan daar binnen een (directe of indirecte) subklasse van `A` altijd naar verwezen worden via de uitdrukking `super.a`. Dit geldt ook als `a` door herdefinitie onzichtbaar is geworden.

3.5 VERBIEDEN VAN SUBKLASSEN EN HERDEFINITIE VAN METHODEN

final

Het is in Java mogelijk om te verbieden dat van een klasse ooit subklassen gemaakt worden of dat een methode ooit wordt gedefinieerd, en wel door in beide gevallen in de definitie het sleutelwoord *final* toe te voegen.

Verbieden subklassen

Van een klasse kan geen subklasse gedefinieerd worden als in de kop van de klasse het sleutelwoord *final* staat:

```
public final class Klassennaam
```

In de package `java.lang` zijn veel klassen, waaronder `String` en alle verpakingsklassen, *final* gedeclareerd:

```
public final class String
public final class Integer
public final class Double
...
```

Dit betekent dat we van deze klassen geen subklassen kunnen definiëren.

Subklassendefinitie

De syntaxis van een *subklassendefinitie* wordt daarmee:

syntaxis

```
[toegang] [final] class klassennaam
           extends superklassennaam
blok
```

Waarom is het verboden om subklassen te definiëren van `String` en van alle verpakingsklassen? De literatuur noemt twee redenen.

– De eerste heeft te maken met efficiency. Dynamische binding is een krachtig maar duur mechanisme: tijdens verwerking moet iedere keer opnieuw uitgezocht worden welke methode bedoeld wordt. Het actuele type van een variabele van gedeclareerd type `String` is altijd `String`, want van een subklasse kan het niet zijn. Dus kunnen aanroepen naar methoden van `String` statisch gebonden worden en dat scheelt veel tijd omdat de Java Virtuele Machine zelf zoveel `Strings` gebruikt.

– De tweede reden is veiligheid. De JVM roept voortdurend methodes van `String` aan. Door herdefinitie van `String` te verbieden, is het uitgesloten dat een aanroep op een object met `String` als gedeclareerd type, gebonden wordt aan een methode van een subklasse die mogelijk allerlei onprettige dingen uithaalt.

Verbieden herdefinitie methoden

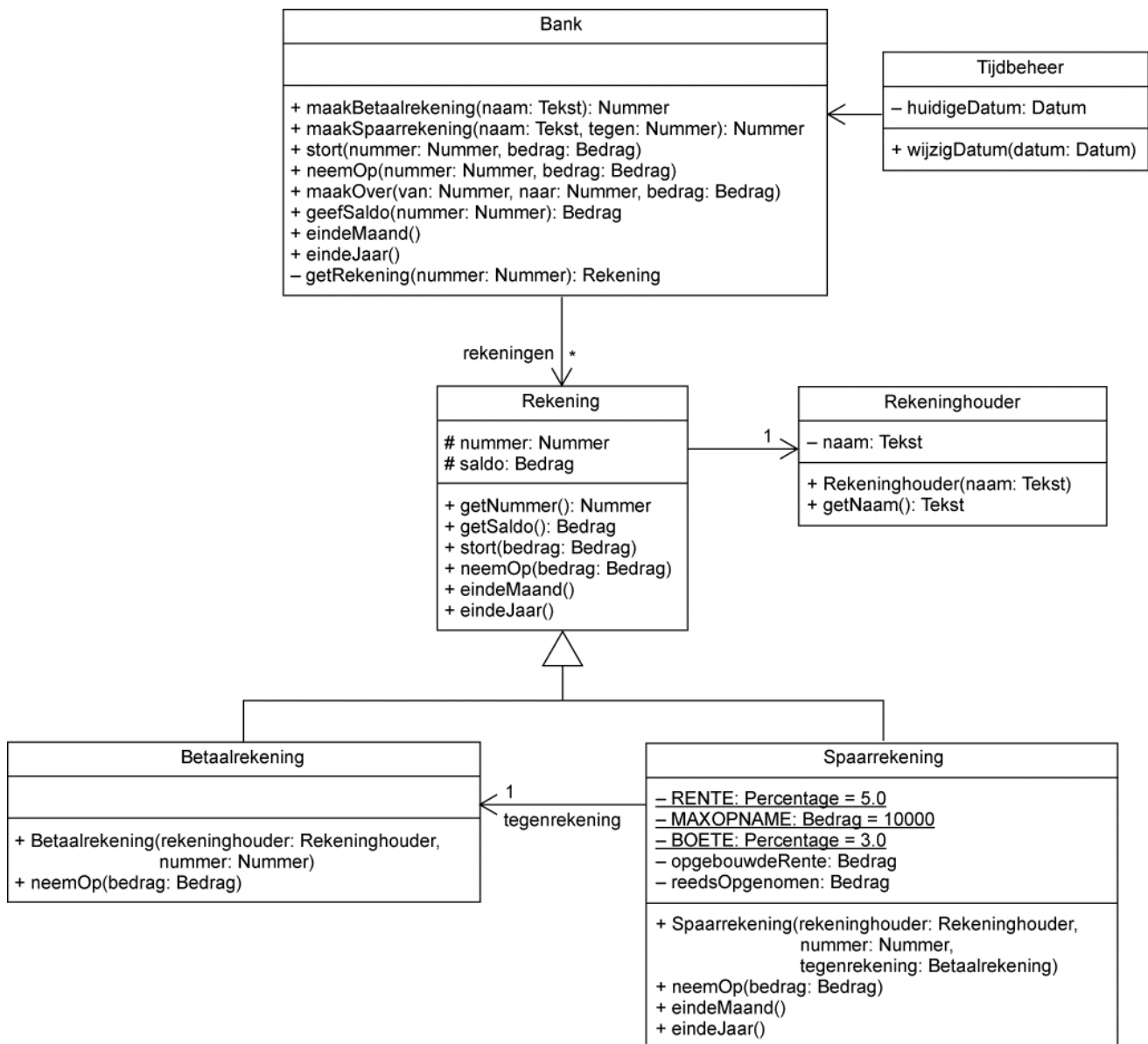
Als we het maken van subklassen van een klasse wel willen toestaan, maar herdefinitie van een bepaalde methode van die klasse willen verbieden, kunnen we in de signatuur van die methode het woord *final* opnemen:

```
toegang final terugkeertype methodenaam(parameterlijst)
```

In het algemeen is het af te raden om klassen of methoden de aanduiding *final* te geven: de herbruikbaarheid van een klasse wordt daardoor immers al bij voorbaat sterk beperkt. Een programmeur moet dus echt een heel goede reden hebben om een klasse of methode *final* te maken.

4 Een toepassing: uitbreiding van de bank

Figuur 2.8 toont het ontwerp voor de banksimulatie uit leereenheid 1. Hierin zijn een aantal wijzigingen aangebracht om dynamische binding mogelijk te maken zoals besproken in de vorige paragraaf: de klasse Rekening heeft (lege) methoden neemOp, eindeMaand en eindeJaar gekregen.



FIGUUR 2.8 Klassendiagram voor de banksimulatie

Stel dat we deze bank willen uitbreiden met een derde soort rekening: een beleggingsrekening. Bij storten of overmaken naar deze rekening onder de € 1500 moet € 25 administratiekosten betaald worden. Ook voor opnamen gelden andere regels dan bij betaalrekeningen en spaarrekeningen. Aan het eind van ieder jaar wordt dividend uitgekeerd. Dit wordt dan toegevoegd aan het saldo van de beleggingsrekening.

Laten we ons afvragen, hoe gemakkelijk we deze uitbreiding kunnen verwezenlijken. We kunnen dat bijvoorbeeld afmeten aan de hoeveelheid code die we moeten veranderen in het bestaande programma. In elk geval moet er een klasse Beleggingsrekening gedefinieerd worden als subklasse van Rekening.

OPGAVE 2.14

Welke methoden krijgt de subklasse Beleggingsrekening?

Daarmee zijn we er nog niet: de nieuwe subklasse moet nu worden ingepast in het bestaande programma.

Probeer zelf eens te bedenken wat er voor de inpassing van de nieuwe subklasse Beleggingsrekening aan het bestaande programma gewijzigd moet worden.

Er moeten beleggingsrekeningen gemaakt kunnen worden. De klasse Bank moet een methode maakBeleggingsrekening krijgen die een nieuwe beleggingsrekening aanmaakt en deze toevoegt aan de lijst met rekeningen.

Verder moet de code gewijzigd worden op alle plaatsen waar een onderscheid gemaakt wordt tussen de verschillende soorten rekeningen. We hopen natuurlijk dat er zo min mogelijk van die plaatsen zijn: hoe minder, hoe gemakkelijker de uitbreiding te realiseren is.

Het is heel belangrijk om een ontwerp zo op te stellen dat dit het gebruik toelaat van de mogelijkheden geboden door dynamische binding. Om dit te verduidelijken, onderzoeken we de (ongunstige) consequenties van twee fictieve ontwerpbeslissingen die dit gebruik in de weg staan.

– In plaats van het enkele attribuut rekeningen krijgt de klasse Bank aparte lijsten voor betaalrekeningen en spaarrekeningen, met bijbehorende methoden getBetaalrekening(nummer) en getSpaarrekening(nummer). Deze methoden geven null terug als ze geen rekening vinden met het gegeven nummer.

– Uit de klasse Rekening wordt de lege methode neemOp verwijderd. Gebruik van dynamische binding wordt daardoor onmogelijk; de methode neemOp kan nu immers alleen worden aangeroepen op een object waarvan het type al tijdens compilatie duidelijk is.

Het ongunstige effect van de eerste beslissing wordt duidelijk nu we een nieuw type rekening toe willen voegen. Bij de introductie van de beleggingsrekening moet de klasse Bank uitgebreid worden met een nieuw attribuut voor een lijst met beleggingsrekeningen en een bijbehorende methode getBeleggingsrekening(nummer). Bij een ontwerp met een gemengde lijst is dat niet nodig.

OPGAVE 2.15

Ga in deze opgave uit van een ontwerp waarin de bovenvermelde beslissingen zijn genomen.

- a Geef een implementatie van de methode neemOp van Bank, nog zonder rekening te houden met beleggingsrekeningen. In deze methode moet eerst de juiste rekeninginstantie worden gezocht. Op deze gevonden instantie moet de methode neemOp worden aangeroepen.
- b Welke wijzigingen zijn nodig in de implementatie van Bank bij het toevoegen van een klasse Beleggingsrekening?

De verschillende typen rekeningen opslaan in verschillende lijsten is dus geen aantrekkelijke optie: naast het feit dat Bank voor ieder type een lijst krijgt en voor ieder type een get-methode om rekeninginstanties te zoeken, moeten ook alle methoden die onderscheid maken tussen de verschillende rekeningtypen aangepast worden. Dit maakt het systeem slecht uitbreidbaar; vrijwel alle methoden van de klasse Bank moeten bij iedere uitbreiding gewijzigd worden. Dat is de reden dat er gekozen is om alle rekeningen, van welk type ook, in één lijst te stoppen.

Stel nu dat we deze ongelukkige ontwerpbeslissing terugdraaien en alle rekeningen weer in één lijst stoppen, maar de tweede beslissing handhaven.

OPGAVE 2.16

Stel de klasse Rekening krijgt geen methode neemOp. Hoe moet de implementatie van methode neemOp van Bank er dan uitzien, met drie typen rekeningen?

Zonder dynamische binding blijft de methode dus afhankelijk van de verschillende rekeningtypen. Als een nieuw rekeningtype wordt toegevoegd, zoals hier de beleggingsrekening, moeten alle methoden van Bank die onderscheid moeten maken tussen verschillende rekeningtypen aangepast worden. Het systeem is dan niet gemakkelijk uit te breiden.

In paragraaf 3.2 is de methode neemOp getoond wanneer wel gebruik gemaakt wordt van dynamische binding. Duidelijk is dat deze implementatie onafhankelijk is van de verschillende rekeningtypen die er zijn. Toevoegen van een nieuw rekeningtype heeft geen gevolgen voor deze methode.

OPDRACHT 2.17

- a Bekijk de code uit het project Le02Bank en vergelijk deze met de code uit Le01Bank. U zult zien dat in Le02Bank beter gebruik wordt gemaakt van het mechanisme van dynamische binding. Let vooral op de implementatie van de methoden eindeMaand en eindeJaar in de klasse Bank. Daarin wordt nu geen onderscheid meer gemaakt tussen de verschillende rekeningsoorten.
- b Stel dat we aan deze implementatie de beleggingsrekening toevoegen, zoals in deze leereenheid is beschreven. Welke wijzigingen van de bestaande code zijn nodig, naast het toevoegen van de klasse Beleggingsrekening zelf? U hoeft deze uitbreiding niet te implementeren!

Uit de voorgaande opgave is duidelijk geworden dat door dynamische binding het aantal plaatsen waarop de bestaande code gewijzigd moet worden, tot een minimum beperkt kan worden. Dit vergroot de uitbreidbaarheid van het systeem.

Open/closed-principe

Het voorgaande is een voorbeeld hoe we aan het *open/closed-principe* kunnen voldoen. Dit belangrijke principe, bedacht door Bertrand Meyer, zegt dat software open moet zijn voor uitbreiding, maar gesloten voor verandering. Oftewel, dat software uitgebreid moet kunnen worden zonder dat de bestaande broncode moet worden aangepast.

S A M E N V A T T I N G

Paragraaf 1 Als door het definiëren van een subklasse de functionaliteit van een superklasse wordt uitgebreid, is sprake van specialisatie: de superklasse vormt het uitgangspunt, de programmeur bedenkt er een subklasse bij. Als bij reeds eerder ontworpen klassen voor de gemeenschappelijke delen een superklasse wordt gedefinieerd, spreken we van generalisatie.

Paragraaf 2 Een subklassendefinitie ziet er als volgt uit:

```
[toegang] [final] class klassennaam
                        extends superklassennaam
blok
```

Het bestaan van subklassen brengt een extra toegangsspecificatie met zich mee.

Een attribuut of methode met toegangsspecificatie `protected` in een klasse A, is toegankelijk vanuit code in de package waartoe A behoort en vanuit alle subklassen van A, ook wanneer ze buiten de package zijn gedefinieerd.

Als de superklasse een parameterloze constructor bevat of geen constructor, dan hoeft een subklasse geen eigen constructoren te bevatten. In dat geval krijgt de subklasse automatisch een parameterloze constructor, die als enige opdracht de parameterloze constructor van de superklasse aanroept. Heeft de superklasse geen parameterloze constructor, dan is een constructor in de subklasse verplicht.

Een constructor van een subklasse moet als eerste opdracht hetzij een constructor van de superklasse aanroepen hetzij een andere constructor in dezelfde klasse. De aanroep van een superklasse-constructor ziet er als volgt uit:

```
super(parameterlijst);
```

De aanroep van een constructor in dezelfde klasse ziet er uit als:

```
this(parameterlijst);
```

De parameters moeten in aantal en type overeenstemmen met die van een (andere) constructor uit de (super)klasse.

Bij constructie van een instantie van een subklasse, wordt eerst de constructor van de superklasse aangeroepen, dan worden de attributen geïnitialiseerd en tot slot wordt de rest van de code uit de constructor van de subklasse uitgevoerd.

Gegeven een declaratie:

```
D d;
```

De variabele `d` heeft dan het *gedecclareerde type* `D`. Tijdens verwerking van het programma kan aan `d` een waarde worden toegekend van een type `A`, als `A` een subklasse is van of gelijk is aan `D`. `A` is dan het *actuele type* van `d`. Op verschillende momenten tijdens verwerking kan een variabele verschillende actuele typen hebben.

Wanneer aan een variabele van het ene type een waarde wordt toegerekend van een ander type, is sprake van type casting. Een cast van een type naar een type hoger in de klassenhiërarchie wordt een upcast genoemd; deze is veilig en dus impliciet. Een cast naar een type lager in de hiërarchie wordt een downcast genoemd. Deze is onveilig en moet daarom expliciet worden aangegeven.

De compiler accepteert een cast (C) als het gedeclareerde type van d een subklasse is van C, gelijk is aan C, of een superklasse is van C (in de eerste twee gevallen is de expliciete cast overbodig). De cast moet dan nog tijdens verwerking correct blijken en dit is het geval wanneer het actuele type een subklasse is van of gelijk is aan C. Voor acceptatie van een cast door de compiler is dus het gedeclareerde type bepalend maar voor acceptatie van een cast tijdens verwerking het actuele type.

Paragraaf 3

Een toegankelijke methode uit een superklasse kan in een subklasse geherdefinieerd worden, dat wil zeggen dat de implementatie van de methode vervangen wordt door een eigen implementatie van de subklasse. *De signaturen van de oorspronkelijke en de geherdefinieerde versie moeten identiek zijn.*

Bij het aanroepen van een methode op een object moet de aanroep gebonden worden aan een specifieke implementatie van die methode. Hierbij is het actuele type van het object bepalend en niet het gedeclareerde type. Dit heet *dynamische binding van methoden*.

Herdefinitie van een attribuut, dat wil zeggen opname in een subklasse van een attribuut met dezelfde naam als een toegankelijk attribuut uit een superklasse, maakt het attribuut uit de superklasse onzichtbaar in de subklasse. We raden dit af.

Geherdefinieerde methoden zijn vanuit de subklasse bereikbaar door gebruik te maken van het sleutelwoord `super`. Bij gebruik van dit sleutelwoord in een uitdrukking van de vorm `super.methodenaam(parameterlijst)` wordt gezocht naar een binding voor `methodenaam` alleen in de superklasse(n) van de klasse waarin deze uitdrukking voorkomt.

Herdefinitie van klassen en methoden kan worden verboden door bij de definitie van deze klassen of methoden het sleutelwoord `final` te plaatsen.

Paragraaf 4

Bij het toevoegen van een nieuwe subklasse van Rekening hoeven we helemaal *niets* te veranderen aan de implementatie van de methoden van Bank. De methoden maken immers in de code helemaal geen onderscheid tussen de verschillende soorten rekeningen: dat onderscheid wordt pas tijdens verwerking gemaakt. De klasse Bank zelf hoeft enkel uitgebreid te worden met een methode om een instantie van het nieuwe rekeningtype te maken en toe te voegen aan de bank.

ZELFTOETS

- 1 a Stel we willen een klasse OmkeerLabel definiëren als subklasse van JLabel, dusdanig dat in een instantie van OmkeerLabel de opgegeven tekst omgekeerd verschijnt. Waarom is de constructor in de volgende definitie van deze klasse onjuist?

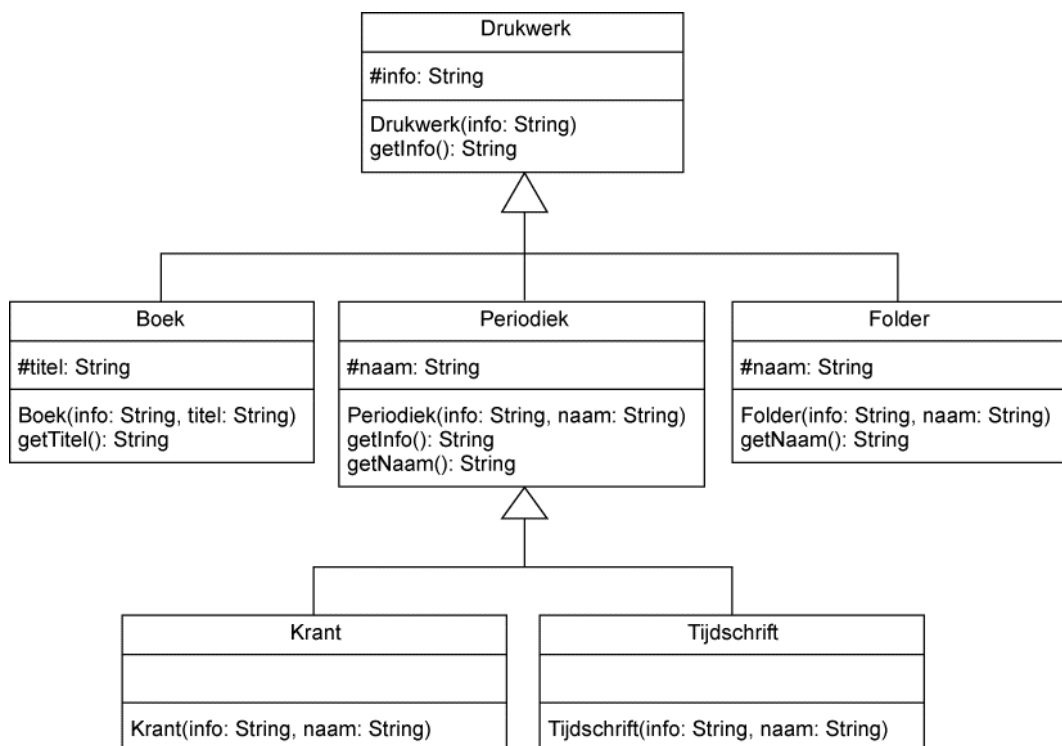
Fout!

```
public class OmkeerLabel extends JLabel {

    public OmkeerLabel(String s) {
        String omkering = "";
        for (int i=0; i < s.length(); i++) {
            omkering = s.charAt(i) + omkering;
        }
        super(omkering);
    }
    ...
}
```

- b Geef een correcte definitie van deze constructor.

- 2 Gegeven het klassendiagram van figuur 2.9.



FIGUUR 2.9 Klassendiagram Drukwerk

Alle attributen die in het diagram zijn opgenomen zijn protected, alle constructoren en methoden zijn public. De constructor van iedere klasse zorgt ervoor dat de attributen van de nieuwe instantie worden geïnitialiseerd met de gelijknamige parameters van de constructor.

De implementatie van de methode `getInfo` van de klasse `Drukwerk` is als volgt:

```
public String getInfo() {
    return "Drukwerk: " + info;
}
```

De implementatie van de methode `getInfo` van de klasse `Periodiek` luidt:

```
public String getInfo() {
    return "Periodiek: " + info;
}
```

- a Geef een opsomming van alle methoden uit het klassendiagram die op een instantie van het type `Folder` kunnen worden aangeroepen.
- b Geef code voor de constructor van `Boek`.
- c Geef van de volgende drie codefragmenten aan of ze door de compiler worden geaccepteerd, en zo ja, of er tijdens de verwerking een fout optreedt.

```
Drukwerk d = new Folder("Folder", "Hema, week 43");
Boek b = (Boek)d;
```

```
Drukwerk d = new Folder("Folder", "V&D, week 45");
String naam = d.getNaam();
```

```
Drukwerk d = new Krant("Dagblad", "De Stem");
String naam = (Krant)d.getNaam();
```

- d Geef de waarde van `s1` na verwerking van het volgende programma-fragment:

```
Drukwerk d1 = new Boek("Roman", "De Avonden");
String s1 = d1.getInfo();
```

- e Geef de waarde van `s2` na verwerking van het volgende programma-fragment:

```
Drukwerk d2 = new Krant("Dagblad", "BN De Stem");
String s2 = d2.getInfo();
```

- f De klassenhiërarchie kan verbeterd worden omdat het ontwerp niet geheel consequent is. Geef een verbeterd klassendiagram.

- 3 Gegeven is de klassenhiërarchie met de klassen `Persoon`, `Docent`, en `Leerling` van figuur 2.3.
 - a Schrijf code voor de creatie en invulling van een arraylist van personen met drie instanties: een instantie van `Persoon` met naam "Jansen", een instantie van `Docent` met naam "Stevens" en een instantie van `Leerling` met leerlingnummer 8741 en naam "van Hal". Hiermee zijn de parameters van de constructoren precies gegeven. Zie desgewenst de definities van `Persoon`, `Docent` en `Leerling` in de terugkoppelingen van opgaven 2.2, 2.4 en 2.5.

b Schrijf een methode met de volgende specificatie:

```
/**
 * Maakt een lijst van alle docenten uit de gegeven
 * lijst met personen
 * @param personen een lijst met personen
 * @return een lijst met docenten
 */
public static ArrayList<Docent>
    geefDocenten(ArrayList<Persoon> personen)
```

- 4 Gegeven is de klassenhiërarchie van de banksimulatie zoals besproken in deze leereenheid. We willen de simulatie uitbreiden met de mogelijkheid om rekeningen op te heffen. Voor ieder rekeningtype moet dat op een andere manier gebeuren.
- Bij een betaalrekening moet eerst gekeken worden of er geen andere rekeningen de betaalrekening als tegenrekening hebben. In dat geval is sluiting niet mogelijk. Kan de rekening wel worden gesloten, dan wordt het geld uitgekeerd (dit blijft buiten het programma).
 - Bij een beleggingsrekening wordt het saldo op de tegenrekening gestort.
 - Bij een spaarrekening wordt daarnaast ook nog rente berekend en uitbetaald tot de dag van opheffing van de rekening.

De klasse Bank wordt uitgebreid met de methode

```
public void sluitRekening(int nummer)
```

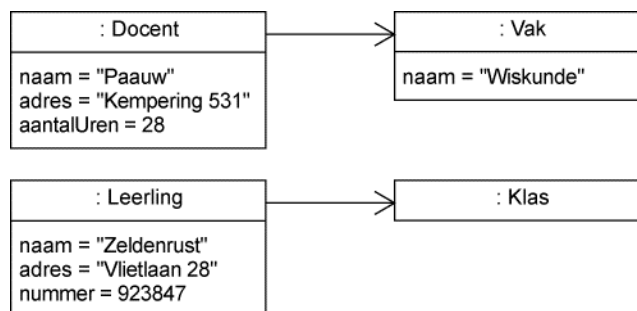
die, indien mogelijk, de rekening sluit en deze uit de lijst met rekeningen haalt.

- a Welke klassen dienen verder aangepast te worden, en welke methoden zijn daarbij nodig? U hoeft geen implementatie van deze methoden te geven.
- b Geef een implementatie van de methode sluitRekening van de klasse Bank.

TERUGKOPPELING

1 Uitwerking van de opgaven

- 2.1 Zie figuur 2.10. Realiseert u zich dat de pijlen tussen de instanties nu staan voor links, verwijzingen in het geheugen, en niet zoals in een klassendiagram, voor associaties.



FIGUUR 2.10 Voorbeelden van instanties van Docent en Leerling

- 2.2 Definitie van de klasse Persoon:

```

public class Persoon {

    private String naam = null;
    protected String adres = "onbekend";

    public Persoon(String naam) {
        this.naam = naam;
    }

    public String getNaam() {
        return naam;
    }

    public String getAdres() {
        return adres;
    }

    public void setAdres(String adres) {
        this.adres = adres;
    }
}
  
```

- 2.3 Omdat naam een private attribuut is van de klasse Persoon, is de waarde vanuit een methode van Docent niet toegankelijk, hoewel het attributen van Docent zelf zijn: zie de uitwerking van opgave 2.1. Om die waarde op te vragen, moet dus gebruik gemaakt worden van de public methode getNaam. De methode omschrijving moet er dus zo uitzien:

```

public String omschrijving() {
    return getNaam() + "\n" +
        adres + "\n" +
        vak.getNaam() + "\n" +
        aantalUren;
}
  
```

- 2.4 a Afgezien van de code voor de constructor, ziet de klassendefinitie er als volgt uit:

```
public class Leerling extends Persoon {

    private int nummer = 0;
    private Klas klas = null;

    public Leerling(int nummer, String naam) {
        ...
    }

    public int getNummer() {
        return nummer;
    }

    public Klas getKlas() {
        return klas;
    }

    public void setKlas(Klas klas) {
        this.klas = klas;
    }
}
```

Het lukt ons echter niet code voor de constructor op te stellen. We zouden willen schrijven:

```
public Leerling(int nummer, String naam) {
    this.nummer = nummer;
    this.naam = naam;
}
```

maar dat kan niet, omdat het attribuut naam binnen Persoon private is en we er dus binnen Leerling niets aan toe mogen kennen. Bovendien is er geen set-methode voor naam!

b In opgave 2.2 werd u gevraagd het attribuut naam van klasse Persoon de toegangsspecificatie private te geven. Als in plaats daarvan voor protected was gekozen, zou de zojuist getoonde code voor de constructor wel juist zijn. Het attribuut is dan immers gewoon toegankelijk vanuit de subklasse.

In paragraaf 2.3 presenteren we nog een andere oplossing.

- 2.5 Definitie van de klasse Docent:

```
public class Docent extends Persoon {

    private Vak vak = null;
    private int aantalUren = 0;

    public Docent(String naam) {
        super(naam);
    }
}
```

```

public Docent(String naam, Vak vak, int uren) {
    super(naam);
    this.vak = vak;
    aantalUren = uren;
}

public Vak getVak() {
    return vak;
}

public int getAantalUren() {
    return aantalUren;
}

public void setAantalUren(int uren) {
    aantalUren = uren;
}

public String omschrijving() {
    return getNaam() + "\n" +
        adres + "\n" +
        vak.getNaam() + "\n" +
        aantalUren;
}
}

```

2.6 De volgende tekst wordt afgedrukt:

```

constructor A: a = 5
constructor B: a = 2, b = 17
main: a = 2, b = 17

```

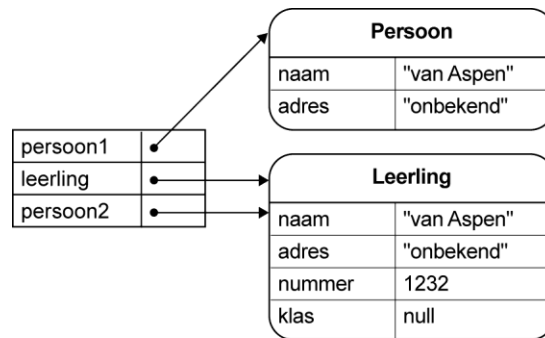
Bij het maken van de instantie b wordt de constructor van B aangeroepen. Hierbij worden de volgende stappen uitgevoerd:

- De constructor van superklasse A wordt aangeroepen.
- De attribuutinitialisatie van A wordt uitgevoerd; attribuut a krijgt de waarde 5.
- De rest van de constructor wordt uitgevoerd. De printopdracht wordt uitgevoerd; daarna krijgt attribuut a de waarde 7. De constructor van A is nu voltooid.
- De attribuutinitialisatie van B wordt uitgevoerd; attribuut b krijgt de waarde 17 (= a + 10).
- De rest van de constructor van B wordt uitgevoerd. Attribuut a krijgt de waarde 2 en de printopdracht wordt uitgevoerd. De constructor van B is nu voltooid.

In de main-methode worden daarna de waarden van de twee attributen opgevraagd en nogmaals afgedrukt.

- 2.7 a Ja, het getal 5 als gehele waarde heeft een andere interne representatie dan 5 als waarde van type double. De waarde van i kan dus niet zonder meer gekopieerd worden, maar moet ook naar een andere representatie worden omgezet.

b Figuur 2.11 toont het toestandsdiagram. De variabele `persoon1` heeft een waarde van type `Persoon`, de variabelen `leerling` en `persoon2` hebben beide een waarde van type `Leerling` (dezelfde waarde; `leerling` en `persoon2` zijn aliassen). De toekenning aan `persoon2` is geoorloofd omdat iedere waarde van type `Leerling` vanwege de subklassenrelatie ook geldt als waarde van type `Persoon`; maar aan de representatie verandert niets.



FIGUUR 2.11 Toestandsdiagram na de toekenningen aan `persoon1`, `leerling` en `persoon2`

2.8 Het gedeclareerde type is in beide gevallen `Rekening`. Het actuele type verschilt per aanroep: in het eerste geval is het `Spaarrekening`, in het tweede geval is het `Betaalrekening`.

2.9 a De opdrachten zijn:

```
persoon2 = docent1;
docent2 = (Docent)persoon1;
```

De eerste toekenning vereist casting van een waarde van type `Docent` naar `Persoon`. Omdat `Docent` een subklasse is van `Persoon`, hebben we te maken met een upcast. Deze is veilig en dus is er geen expliciete cast vereist. De tweede toekenning vereist casting van `Persoon` (het gedeclareerde type) naar `Docent`. Dit is een onveilige downcast en dus is wel een expliciete cast vereist.

b Nee, dat kan niet. Hoewel `Docent` en `Leerling` beide subklassen zijn van `Persoon`, bestaat er tussen deze twee typen geen superklasse-subklasse relatie. Een toekenning `leerling = docent1` is daarom niet mogelijk, ook niet met een expliciete cast.

2.10 a Dit leidt tot een foutmelding van de compiler: `Persoon` is een subklasse van `Object`. We hebben dus te maken met een downcast, en dus is in elk geval een expliciete cast vereist.

b Dit leidt tot een foutmelding bij verwerking: de cast is niet uitvoerbaar omdat het actuele type van `obwaarde` `Object` is en `Object` een superklasse is van `Persoon`.

c Dit is correct, al is de expliciete cast in dit geval overbodig. Het gedeclareerde type is `Persoon`, het actuele type is `Leerling` en er geldt dat `Leerling` een subklasse is van `Persoon`.

d De eerste toekenning is correct (`Docent` is een subklasse van `Persoon`; dus dit is een veilige upcast). De tweede toekenning leidt tot een foutmelding tijdens compilatie: `Docent` is een subklasse van `Persoon`. Deze downcast vereist een expliciete cast.

e De eerste toekenning is correct (een veilige upcast). De tweede toekenning wordt geaccepteerd door de compiler: het gedeclareerde type van pvar is Persoon en Docent is een subklasse van Persoon. Het leidt echter tot een foutmelding tijdens de verwerking. Het actuele type van pvar is Leerling en er bestaat tussen Docent en Leerling geen superklasse-subklasse relatie.

f Beide toekenningen worden door de compiler geaccepteerd. Tijdens verwerking van de tweede toekenning treedt er echter een fout op. Het actuele type van obvar is Persoon; obvar kan niet worden toegekend aan een variabele van zijn subklasse Docent.

- 2.11 In t1.plus(2) wordt de implementatie van plus uit Som1 gebruikt; omdat de waarde van term1 gelijk is aan 5, levert deze aanroep de waarde 7 op. In t2.plus(2) wordt de implementatie van plus uit Som2 gebruikt; omdat de waarden van term1 en term2 gelijk zijn aan 3 respectievelijk 4, levert deze aanroep de waarde 9 op.

- 2.12 Als de parameter obj niet van het type Persoon is, wordt false teruggegeven. Is dat het geval, dan vergelijken we naam en adres.

```
public boolean equals(Object obj) {
    if (!(obj instanceof Persoon)) {
        return false;
    }
    String naam2 = ((Persoon)obj).getNaam();
    String adres2 = ((Persoon)obj).getAdres();
    return (naam.equals(naam2) && adres.equals(adres2));
}
```

Merk op dat binnen deze methode equals strings vergeleken worden en wel met behulp van de methode equals van String. Daar is geen enkel bezwaar tegen.

- 2.13 a We moeten ook de klasse Rekening de methoden eindeMaand en eindeJaar geven. Net als de methode neemOp is de implementatie daarvan leeg. Figuur 2.12 toont het gewijzigde klassendiagram voor de klasse Rekening. De in grijs gegeven methoden zijn de lege methoden om dynamische binding mogelijk te maken.

Rekening
nummer: Nummer saldo: Bedrag
getNummer(): Nummer getSaldo(): Bedrag stort(bedrag: Bedrag) neemOp(bedrag: Bedrag) eindeMaand() eindeJaar()

FIGUUR 2.12 Gewijzigd klassendiagram voor de klasse Rekening

b De implementatie van de methoden `eindeMaand` en `eindeJaar` van `Bank` worden nu als volgt:

```
/**
 * Roept eindeMaand aan voor alle rekeningen.
 */
public void eindeMaand() {
    for (Rekening r : rekeningen) {
        r.eindeMaand();
    }
}

/**
 * Roept eindeJaar aan voor alle rekeningen.
 */
public void eindeJaar() {
    for (Rekening r : rekeningen) {
        r.eindeJaar();
    }
}
```

Is `r` een instantie van `Spaarrekening`, dan wordt de methode `eindeMaand` van `Spaarrekening` aangeroepen; deze zal de rente over de afgelopen maand berekenen. Is `r` daarentegen een `Betaalrekening`, dan wordt de methode `eindeMaand` van de klasse `Betaalrekening` aangeroepen. Omdat deze methode binnen de klasse `Betaalrekening` geen implementatie heeft, wordt de methode `eindeMaand` van de superklasse `Rekening` gebruikt. Deze doet niets.

Het resultaat is, dat de code van `eindeMaand` nu geen onderscheid meer maakt tussen de verschillende typen rekeningen, terwijl er alleen echt iets gedaan wordt voor spaarrekeningen.

2.14 De klasse `Beleggingsrekening` moet in ieder geval de methoden `stort(bedrag: Bedrag)`, `neemOp(bedrag: Bedrag)`, en `eindeJaar()` herdefiniëren. De klasse `Rekening` heeft al een methode `stort(bedrag: Bedrag)` omdat in het oorspronkelijke ontwerp het storten bij alle rekeningtypen op exact dezelfde manier gebeurde. Storten op een beleggingsrekening heeft een afwijkende functionaliteit; daarom moet `Beleggingsrekening` een eigen `stort`-methode krijgen.

2.15 a Deze methode zou er dan bijvoorbeeld als volgt uitzien:

```
public void neemOp(int nummer, double bedrag) {
    Betaalrekening betaalrekening =
        getBetaalrekening(nummer);
    if (betaalrekening != null) {
        betaalrekening.neemOp(bedrag);
    } else {
        Spaarrekening spaarrekening =
            getSpaarrekening(nummer);
        if (spaarrekening != null) {
            spaarrekening.neemOp(bedrag);
        }
    }
}
```

b In Bank moeten niet alleen de implementatie van `neemOp`, maar ook die van `stort` en `maakOver` gewijzigd worden. Zij zullen immers allemaal onderscheid moeten maken tussen de verschillende soorten rekeningen. Verder moet de methode `eindeJaar` worden gewijzigd. Die moet immers nu niet alleen alle spaarrekeningen langs, maar ook alle beleggingsrekeningen.

2.16 Ook nu moet er onderscheid gemaakt worden tussen de soorten rekeningen:

```
public void neemOp(int nummer, double bedrag) {
    Rekening rekening = getRekening(nummer);
    if (rekening != null) {
        if (rekening instanceof Betaalrekening) {
            ((Betaalrekening) rekening).neemOp(bedrag);
        } else if (rekening instanceof Spaarrekening) {
            ((Spaarrekening) rekening).neemOp(bedrag);
        } else {
            ((Beleggingsrekening) rekening).neemOp(bedrag);
        }
    }
}
```

2.17 a Geen terugkoppeling.

b De volgende wijzigingen zijn nodig:

– Aan de klasse `Bank` moet een methode `maakBeleggingsrekening` worden toegevoegd, waarin een instantie van de klasse `Beleggingsrekening` wordt gemaakt en wordt toegevoegd aan de rekeninglijst.

– De klasse `BankFrame` moet worden aangepast. Het moet mogelijk zijn ook te kiezen voor een beleggingsrekening. Er moet aan de keuzelijst een nieuw item `Beleggingsrekening` worden toegevoegd en de methode `openButtonAction` moet worden aangepast zodat deze de bank opdracht kan geven een beleggingsrekening te openen.

Verder zijn er geen wijzigingen nodig. Alle andere rekeningtype-afhankelijke acties in de klasse `Bank` worden door het mechanisme van dynamische binding automatisch goed verwerkt.

2 Uitwerking van de zelftoets

1 a De aanroep naar de constructor van de superklasse moet de eerste opdracht zijn in de constructor van de subklasse. We kunnen dus niet eerst de gewenste waarde voor de parameter uitrekenen.

b Een juiste versie is:

```
public class OmkeerLabel extends JLabel {

    public OmkeerLabel(String s) {
        super();
        String omkering = "";
        for (int i=0; i < s.length(); i++) {
            omkering = s.charAt(i) + omkering;
        }
        super.setText(omkering);
    }
    ...
}
```

Er wordt nu eerst een lege label gemaakt. Wanneer de juiste tekst is bepaald, wordt deze in de label geplaatst door middel van een aanroep van `setText`. De aanroep van de parameterloze constructor van `JLabel` mag ook worden weggelaten.

We hebben `super.setText(omkering)` geschreven en niet gewoon `setText(omkering)` voor het geval ook de methode `setText` in de subklasse wordt geherdefinieerd. Het is niet fout als u dit niet heeft gedaan.

- 2
 - a Dat zijn alle methoden van de klasse `Folder` zelf en de overgeërfde methoden uit zijn superklas(sen), in dit geval alleen `Drukwerk`. De gevraagde methoden zijn dus `getNaam` en `getInfo`.
 - b Deze constructor ziet er als volgt uit:

```
public Boek(String info, String titel) {
    super(info);
    this.titel = titel;
}
```

- c De compiler accepteert de code van het eerste fragment. `Folder` is een subklasse van `Drukwerk`, dus in de eerste regel hebben we te maken met een veilige upcast. `Boek` is een subklasse van `Drukwerk`; de tweede regel bevat daarom een downcast, die expliciet moet worden uitgevoerd, zoals hier is gedaan. Tijdens verwerking gaat het fout in de tweede regel; het actuele type van `d` blijkt `Folder` te zijn en geen `Boek`.

Bij het tweede fragment geeft de compiler een foutmelding op de tweede regel want de klasse `Drukwerk` kent geen methode `getNaam`. De compiler kijkt naar het gedeclareerde type.

Bij het derde fragment geeft de compiler een foutmelding op de tweede regel. De intentie van deze code is goed, alleen de expliciete cast is verkeerd. De punt heeft een hogere prioriteit dan de cast, dus deze wordt als eerste uitgevoerd. De methode `getNaam` wordt volgens deze code dus aangeroepen op een instantie van `Drukwerk`; `Drukwerk` kent echter geen methode `getNaam`. De volgende regel was wel goed geweest:

```
String naam = ((Krant)d).getNaam();
```

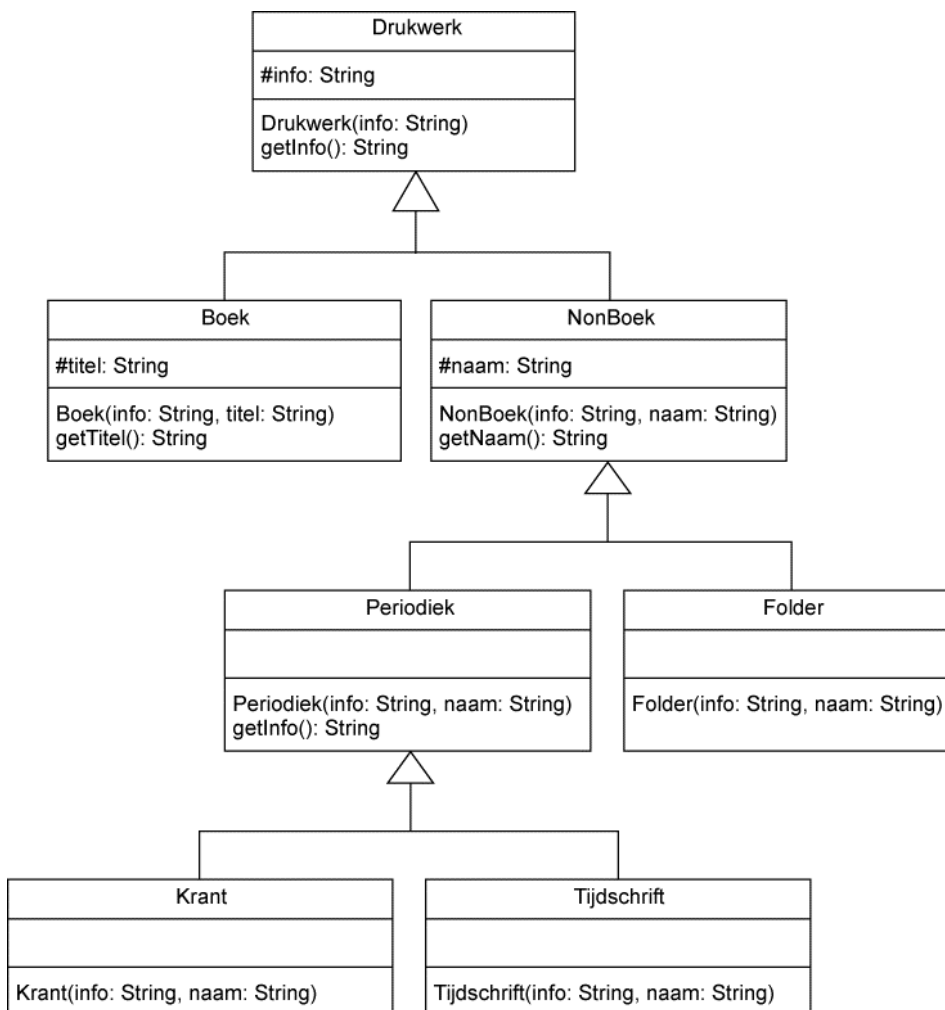
- d De methode `getInfo` behorend bij het actuele type van `d1` wordt aangeroepen. Het actuele type is `Boek`. `Boek` zelf heeft geen methode `getInfo`, maar erft deze van `Drukwerk`. De waarde van `s1` wordt daarmee:

```
Drukwerk: Roman
```

- e Het actuele type van `d2` is `Krant`. `Krant` zelf heeft geen methode `getInfo` maar erft deze van `Periodiek`. De waarde van `s2` wordt daarmee:

```
Periodiek: Dagblad
```


f De klassen Periodiek en Folder hebben een aantal gemeenschappelijke attributen en methoden. Hierop kan een generalisatie worden uitgevoerd. Het resultaat is te zien in figuur 2.13



FIGUUR 2.13 Een verbeterd klassendiagram

Puur overervingtechnisch gezien is het ook mogelijk om geen extra klasse te introduceren, door Periodiek subklasse te maken van Folder (specialisatie): dan erft periodiek de gemeenschappelijke attributen en methoden van Folder. Dit is echter semantisch niet correct: een periodiek is geen folder. Daarom is deze oplossing niet gewenst.

3 a De code is als volgt:

```

ArrayList<Persoon> personen = new ArrayList<>();
personen.add(new Persoon("Jansen"));
personen.add(new Docent("Stevens"));
personen.add(new Leerling(8741, "van Hal"));
    
```

b De implementatie ziet er bijvoorbeeld als volgt uit:

```
public static ArrayList<Docent>
    geefDocenten(ArrayList<Persoon> personen) {
    // maak lege lijst voor docenten
    ArrayList<Docent> docenten = new ArrayList<>();
    // kijk voor elke persoon of het een docent is
    // en zo ja, plaats deze in docentenlijst
    for (Persoon p : personen) {
        if (p instanceof Docent) {
            docenten.add((Docent)p);
        }
    }
    return docenten;
}
```

We onderzoeken van iedere persoon in de personenlijst met behulp van de operator instanceof of deze persoon ook een instantie van Docent is. Is dat het geval, dan wordt de persoon op de docentenlijst geplaatst. Hierbij is een expliciete cast nodig omdat we van Persoon naar Docent gaan (downcast).

- 4 a Alle rekeningklassen moeten worden aangepast (ook Rekening). Aan al deze klassen moet de volgende methode worden toegevoegd:

```
public boolean sluit()
```

die true teruggeeft als het sluiten van de rekening is gelukt, en false als het sluiten van de rekening niet mogelijk is.

Opgemerkt kan worden dat de methode sluit in de klasse Rekening een vrijwel lege romp krijgt; deze methode is er enkel om dynamische binding mogelijk te maken. Helemaal leeg kan deze methode echter niet zijn. Omdat de methode een niet-void terugkeertype heeft moet deze methode een returnopdracht bevatten. Een mogelijke implementatie zou kunnen zijn:

```
public boolean sluit() {
    return false;
}
```

In Betaalrekening, Spaarrekening, en Beleggingsrekening krijgt de methode sluit een implementatie die de regels behorende bij het sluiten van een dergelijke rekening implementeert.

b Een mogelijke implementatie is

```
public void sluitRekening(int nummer) {
    Rekening rekening = getRekening(nummer);
    if (rekening != null) {
        // sluit rekening
        if (rekening.sluit()) {
            // haal rekening uit lijst
            rekeningen.remove(rekening);
        }
    }
}
```


Overerving (2)

Introductie 101

Leerkern 101

- 1 Lijst, bag en verzameling 101
 - 1.1 De klasse Bag 102
 - 1.2 De klasse Verzameling 104
 - 1.3 Overerving en delegatie 107
- 2 De VormenApplicatie 110
 - 2.1 Productomschrijving 110
 - 2.2 Toevoegen van nieuwe vormen 111
 - 2.3 Stringrepresentaties tonen 111
 - 2.4 Totale inhoud berekenen 111
 - 2.5 Evaluatie van de applicatie 113

Samenvatting 115

Zelftoets 115

Terugkoppeling 117

- 1 Uitwerking van de opgaven 117
- 2 Uitwerking van de zelftoets 129

Overerving (2)

INTRODUCTIE

In leereenheid 2 hebt u de theorie van overerving bestudeerd. Nu krijgt u de kans om deze theorie in een paar eenvoudige projecten toe te passen. In het eerste project dient u een klasse *Verzameling* te definiëren, als subklasse van een gegeven klasse. Het tweede project betreft de definitie van een eenvoudige objecthiërarchie met een toepassing van dynamische binding. In beide gevallen is een testomgeving als bouwsteen aanwezig.

De toepassingen zijn uit het oogpunt van dagelijks gebruik weinig interessant. De nadruk in deze leereenheid ligt op het bestendigen van de theorie en niet op de toepassingen als zodanig.

Ten slotte wordt naast overerving nog een tweede manier besproken om functionaliteit van een klasse te gebruiken binnen een andere klasse, namelijk delegatie. De voor- en nadelen van beide manieren worden besproken.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u

- een subklasse van een bestaande klasse in een applicatie kunt opnemen
- een eenvoudige objecthiërarchie kunt ontwerpen
- kunt aangeven wat het verschil is tussen overerving en delegatie en weet wanneer delegatie een betere oplossing is dan overerving.

Studeeraanwijzing

De studielast van deze leereenheid bedraagt circa 6 uur.

LEERKERN

1 **Lijst, bag en verzameling**

De toepassing waarmee we ons in deze paragraaf bezighouden, betreft collecties van gehele getallen. Er zijn verschillende soorten collecties. We bekijken er hier drie.

In een *lijst* hebben de elementen een bepaalde volgorde; een element mag in een lijst vaker dan een keer voorkomen.

Een *bag* is een ongeordende collectie waarin elementen meer dan een keer voor kunnen komen.

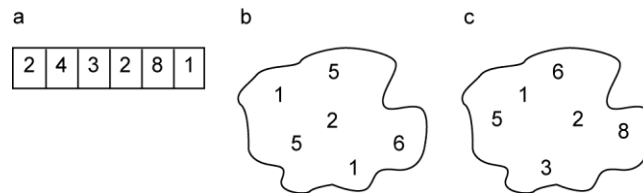
In een *verzameling* zijn de elementen niet geordend en komt elk element slechts één keer voor.

Lijst

Bag

Verzameling

Figuur 3.1 illustreert deze drie vormen van een collectie.



FIGUUR 3.1 Voorbeelden van een lijst (a), een bag (b) en een verzameling (c)

Welke representaties van een lijst in Java kent u?

Als eerste representatie van een lijst kent Java de array. Alle elementen in een array moeten van hetzelfde type zijn, bijvoorbeeld allemaal gehele getallen of allemaal instanties van de klasse `Rekening`. In het laatste geval kunnen ook instanties van subklassen van `Rekening` aan de elementen worden toegekend: er vindt dan type casting plaats. Het maximale aantal elementen ligt vast bij creatie van de array. Als we van tevoren niet precies weten hoeveel elementen een lijst zal gaan bevatten, is dit een nadeel; het is mogelijk dat er een element aan moet worden toegevoegd terwijl de array al vol is.

Een meer flexibele representatie van een lijst is de `ArrayList`. Ook in een `ArrayList` moeten alle elementen van hetzelfde type zijn. Voordeel van de `ArrayList` is dat niet van te voren opgegeven hoeft te worden hoe groot de lijst zal zijn; de lijst groeit automatisch bij toevoegen van nieuwe elementen, en krimpt bij het verwijderen van elementen.

1.1 DE KLASSE BAG

Java biedt dus verschillende manieren om een lijst te representeren. Een representatie van een bag – een ongeordende collectie waarin elementen meer keren voor mogen komen – is niet standaard voorhanden, dus hebben we deze zelf gedefinieerd. De interface van deze klasse is getoond in figuur 3.2.

De klasse bevat dus public methoden om een element aan de bag toe te voegen, een element eruit te verwijderen, na te gaan of een bepaald element in de bag voorkomt, alle elementen uit de bag te verwijderen, na te gaan of de bag leeg is, het aantal elementen te bepalen, alle elementen op te leveren in een array, de inhoud van de bag te vergelijken met de inhoud van een andere bag, en een stringrepresentatie van de bag te construeren.

De methoden `verwijder` en `voegToe` geven `true` terug als de Bag door de aanroep gewijzigd is, dus als het verwijderen of toevoegen geslaagd is, en anders `false`.

De methode die alle elementen oplevert, kunnen we gebruiken als we iets met al die elementen willen doen. We kunnen deze array *niet* gebruiken om de bag te wijzigen: het is een nieuw object dat speciaal voor de gelegenheid geconstrueerd wordt en niet een object dat ook binnen de bag zelf wordt gebruikt.

Method Summary	
int	<u>afmeting()</u> Geeft het aantal elementen in de bag.
boolean	<u>bevat (int elem)</u> Gaat na of de bag het gegeven element bevat.
int[]	<u>elementen()</u> Geeft een array met de elementen in de bag.
boolean	<u>equals (java.lang.Object obj)</u> Vergelijkt bag met een andere bag.
boolean	<u>isLeeg()</u> Gaat na of de bag leeg is.
void	<u>maakLeeg()</u> Verwijdert alle elementen uit de bag.
java.lang.String	<u>toString()</u> Geeft een string-representatie van de bag: vijf elementen per regel, gescheiden door komma's.
boolean	<u>verwijder (int elem)</u> Verwijdert een element uit de bag.
boolean	<u>voegToe (int elem)</u> Voegt een element toe aan de bag.

FIGUUR 3.2 De interface van de klasse Bag

OPGAVE 3.1

Bekijk de volgende Java-opdrachten.

```
Bag b = new Bag();
b.voegToe(1);
b.voegToe(5);
b.voegToe(1);
b.verwijder(1);
b.verwijder(5);
```

Wat is nu de waarde van de volgende aanroepen?

- a b.bevat(1);
- b b.bevat(5);

Bij de implementatie van deze klasse moest allereerst bepaald worden hoe de elementen binnen de klasse Bag worden opgeslagen. Om geen beperkingen op te leggen aan de afmeting van een bag ligt het voor de hand om te kiezen voor de ArrayList. De klasse Bag heeft daarom een attribuut elementen van het type ArrayList<Integer>.

```
public class Bag {

    private ArrayList<Integer> elementen =
        new ArrayList<>();

    // hier komen de methoden
}
```

OPDRACHT 3.2

Open het project Le03Verzameling. In de package verzameling vindt u een vrijwel complete implementatie van de klasse Bag. Bekijk deze klasse en implementeer dan de methode equals (de enige nog ontbrekende methode).

Aanwijzing:

Het vaststellen of twee bags gelijk zijn, is niet triviaal. Het kan op verschillende manieren worden aangepakt. Een handige manier is om de arrayrepresentaties te gebruiken die gemaakt worden met de methode elementen. U kunt dan de methoden sort en equals te hulp roepen van de klasse Arrays uit de package java.util.

1.2 DE KLASSE VERZAMELING

Een verzameling is een speciaal geval van een bag, namelijk een bag waarin alle elementen maar één keer voorkomen. We gaan u nu vragen een klasse Verzameling te definiëren als een subklasse van de gegeven klasse Bag.

Product-
omschrijving

De klasse Verzameling beschikt net als de klasse Bag over methoden om een element toe te voegen, een element te verwijderen, na te gaan of een bepaald element in de verzameling voorkomt, alle elementen te verwijderen, na te gaan of de verzameling leeg is, het aantal elementen te bepalen, alle elementen op te leveren in een array, de inhoud van de verzameling te vergelijken met de inhoud van een andere verzameling en een stringrepresentatie van de verzameling te construeren. De klasse moet daarnaast beschikken over twee extra methoden vereniging en doorsnede, met de specificatie getoond in figuur 3.3.

vereniging

```
public Verzameling vereniging(Verzameling v)
```

Berekent de vereniging van de verzameling met een andere verzameling

Parameters:

v - de andere verzameling

Returns:

de vereniging, een nieuwe verzameling

doorsnede

```
public Verzameling doorsnede(Verzameling v)
```

Berekent de doorsnede van de verzameling met een andere verzameling

Parameters:

v - de andere verzameling

Returns:

de doorsnede, een nieuwe verzameling

FIGUUR 3.3 Specificatie van vereniging en doorsnede

Ter verduidelijking: stel dat $v1$ als waarde de verzameling $\{1, 4, 9, 16\}$ heeft en $v2$ heeft als waarde de verzameling $\{1, 3, 5, 7, 9, 11\}$. Dan moet $v1.vereniging(v2)$ de nieuwe verzameling $\{1, 3, 4, 5, 7, 9, 11, 16\}$ zijn en $v1.doorsnede(v2)$ de nieuwe verzameling $\{1, 9\}$.

Opmerking

In de implementatie hoeven de elementen niet opklimmend gesorteerd te zijn zoals in bovenstaande voorbeelden.

OPGAVE 3.3

- U gaat Verzameling definiëren als subklasse van Bag. Welke methoden van Bag blijven bruikbaar? Welke moeten worden geherdefinieerd? Welke methoden moeten er dan nog aan de klasse worden toegevoegd? Geef de specificatie van iedere methode die u moet herdefiniëren.
- Heeft de klasse Verzameling een constructor nodig? Zo ja, specificeer deze.

Voor de implementatie en het testen van de klasse Verzameling kunt u beschikken over de superklasse Bag en over een gebruikersinterface die u als testomgeving kunt gebruiken. Deze interface is getoond in figuur 3.4.

De testomgeving werkt met twee verzamelingen, verzameling1 en verzameling2. De interface toont een stringrepresentatie van de geselecteerde verzameling, die wordt bijgewerkt bij klikken op een van de knoppen Voeg toe, Verwijder of Maak Leeg. Klikte u op Vereniging of Doorsnede, dan ziet u beide verzamelingen, plus het resultaat van de operatie. Klikken op Gelijk levert het resultaat van de vergelijking van de twee verzamelingen.

Let op

Het testen met behulp van een gebruikersinterface heeft enkele nadelen: het is onduidelijk waar een eventuele fout optreedt (dat kan ook in de interface zijn) en de tests kunnen niet automatisch opnieuw worden verwerkt, bijvoorbeeld wanneer een fout is hersteld. Het gaat hier echter om een eenvoudig voorbeeldprogramma. We laten het schrijven van een (lang) testprogramma daarom achterwege.



FIGUUR 3.4 Interface van testomgeving voor de klasse Verzameling

OPDRACHT 3.4

- a Open het project Le03Verzameling (de gui-klasse bevat fouten omdat Verzameling nog niet is geïmplementeerd). Voeg een nieuwe klasse Verzameling toe aan de package verzameling. Deze klasse erft van Bag. Geef specificaties van alle methoden. Laat de rompen nog leeg.
- b Ontwerp en implementeer nu achtereenvolgens alle methoden.

Aanwijzing:

Bij implementatie van een geherdefinieerde methode kunt u de oorspronkelijke methode uit de superklasse gebruiken (zie paragraaf 3.4 van leereenheid 2).

- c Wat is er mis met de volgende implementatie van de methode vereniging?

```
public Verzameling vereniging(Verzameling v) {
    for (int elem : this.elementen()) {
        v.voegToe(elem);
    }
    return v;
}
```

Als we een programma testen, proberen we een aantal invoercombinaties uit. De kunst is nu om die invoercombinaties zo te kiezen dat ze representatief zijn voor alle invoer. In dit voorbeeld zullen we in elk geval alle gebruiksmogelijkheden moeten uitproberen, dus toevoegen, verwijderen, leegmaken, verenigen, doorsnijden en testen op gelijkheid. We proberen nu voor elk van deze gebruiksmogelijkheden representatieve voorbeelden te vinden.

Welke gevallen kunnen worden onderscheiden bij het toevoegen van een geheel getal aan een verzameling?

Bij verzamelingen is er een belangrijk onderscheid tussen de lege verzameling en de rest. We moeten daarom in elk geval het toevoegen van een element aan een lege verzameling uitproberen. Dan is het van belang of het element al tot de verzameling behoorde of niet. In het eerste geval moet het niet worden toegevoegd, en in het tweede geval wel.

We kunnen er ook nog over denken om onderscheid te maken tussen de aard van de elementen die we toevoegen (positief, nul of negatief). Omdat er in de klasse Verzameling op geen enkele manier een onderscheid wordt gemaakt tussen die gevallen, is het niet nodig daar heel veel aandacht aan te besteden.

Omdat we nog niet proberen robuuste applicaties te maken, heeft het weinig zin om te bekijken wat er gebeurt als de gebruikersinvoer niet klopt. We zullen dus geen elementen als 'a' of 'b' proberen toe te voegen.

Op grond van deze overwegingen stellen we de volgende testgevallen samen voor de gebruiksmogelijkheid toevoegen:

- voeg -2 toe aan de lege verzameling
- voeg -1 toe aan de verzameling {-2}
- voeg 0 toe aan de verzameling {-2, -1}
- voeg -1 toe aan de verzameling {-2, -1, 0}
- voeg 1 toe aan de verzameling {-2, -1, 0}.

OPGAVE 3.5

Bedenk zelf testwaarden voor de andere vijf gebruiksmogelijkheden:

- a verwijderen
- b leegmaken
- c verenigen
- d doorsnijden
- e gelijkheid.

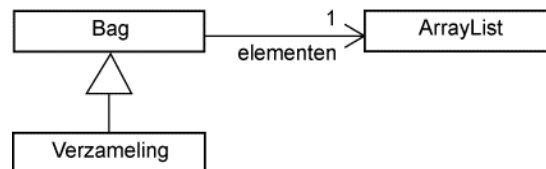
OPDRACHT 3.6

Test de klasse Verzameling met behulp van de gegeven applicatie en de testwaarden uit de terugkoppeling van opgave 3.5.

1.3 OVERERVING EN DELEGATIE

Figuur 3.5 toont het klassendiagram van de applicatie uit de vorige paragraaf (de gebruikersinterface is buiten beschouwing gelaten). Hierin ziet u dat de klasse Verzameling een subklasse is van Bag, en dat Bag gebruik maakt van de klasse ArrayList om de elementen in op te slaan. Het beheren van de waarden van de bag wordt daarmee door de klasse Bag overgelaten aan het attribuut elementen van het type ArrayList<Integer>. Dit overdragen (delegeren) van functionaliteit aan een instantie van een andere klasse wordt *delegatie* genoemd.

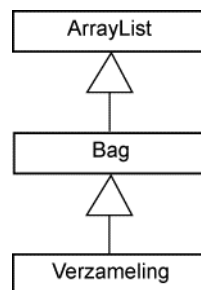
Delegatie



FIGUUR 3.5 Klassendiagram gebruikmakend van delegatie

Kunt u een andere oplossing bedenken om de klasse Bag te ontwerpen?

Een andere oplossing is om Bag te laten erven van de klasse ArrayList, zoals getoond in figuur 3.6. In dit geval erft Bag alle methoden van ArrayList en kan deze geërfde methoden gebruiken om de elementen te beheren.



FIGUUR 3.6 Klassendiagram gebruikmakend van overerving

OPDRACHT 3.7

- a Geef een implementatie voor de klasse Bag gebruikmakend van overerving.
- b Is het, vanwege de nieuwe implementatie van Bag, nodig om de klasse Verzameling aan te passen?

We hebben nu twee manieren gezien om de functionaliteit van een klasse te gebruiken binnen een andere klasse: delegatie en overerving. Welke heeft nu de voorkeur?

OPGAVE 3.8

Kunt u een nadeel noemen van overerving in voorgaand voorbeeld?

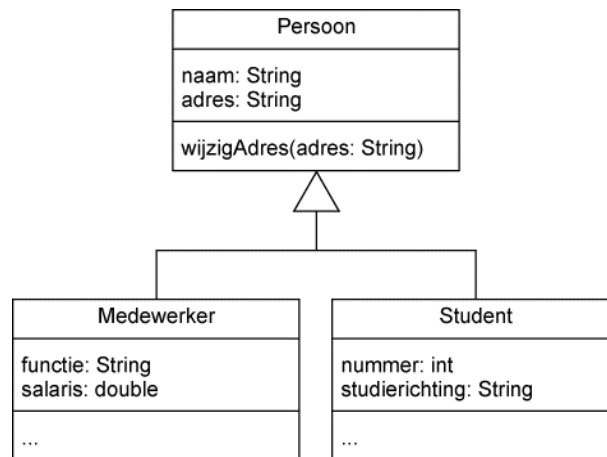
Tip: denk goed na over de functionaliteit van de klassen Bag en Verzameling.

Uit de voorgaande opgave is te concluderen dat wanneer we functionaliteit van een klasse willen hergebruiken maar die klasse ook methoden bevat die we niet willen toestaan, delegatie een betere keuze is dan overerving. Overerving is dus alleen de juiste keuze als alle protected en public attributen en methoden van de superklasse ook bruikbaar zijn in de subklasse (tenzij ze daar worden geherdefinieerd).

Laten we nog eens naar overerving en delegatie kijken in een ander voorbeeld.

Voorbeeld

Gegeven is een systeem voor de administratie van een universiteit zoals getoond in figuur 3.7. Het houdt de gegevens van studenten en medewerkers bij. Zowel de klasse Medewerker als de klasse Student erft gemeenschappelijke (persoons-)eigenschappen van de klasse Persoon.

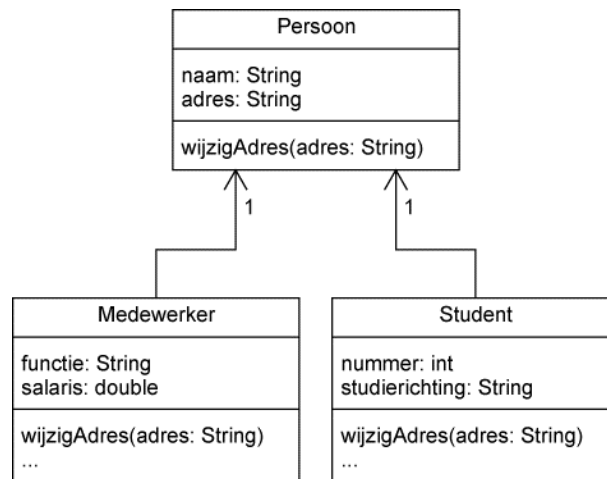


FIGUUR 3.7 Klassendiagram met overerving voor een systeem voor een universiteit

Wat moet er gebeuren als een student een baantje bij de universiteit krijgt, bijvoorbeeld als studentassistent?

Dan moet er een nieuwe instantie van medewerker gemaakt worden. Omdat deze instantie ook alle persoonsgegevens bevat, zullen de persoonsgegevens van dezelfde persoon nu twee maal in het systeem staan: eenmaal bij de medewerker en eenmaal bij de student. Naast het feit dat dit meer geheugen kost, bestaat er nu ook het gevaar dat gegevens inconsistent worden: de student geeft door dat hij is verhuisd; dat wordt doorgevoerd bij de Student-instantie maar niet bij de Medewerker-instantie.

Het is daarom beter dat de persoonsgegevens gedeeld worden. Dat betekent dat het voordelen heeft om delegatie te gebruiken in plaats van overerving. Het nieuwe klassendiagram wordt getoond in figuur 3.8.



FIGUUR 3.8 Klassendiagram met delegatie voor een systeem voor een universiteit

Dit voorbeeld toont een veelvoorkomende situatie waarbij delegatie te prefereren is boven overerving: namelijk als een object verschillende rollen kan aannemen (en/of van rol kan wisselen), zoals hier een persoon die de rol student, de rol medewerker, of zelfs beide rollen kan aannemen.

Voordelen van delegatie t.o.v. overerving

Delegatie heeft een aantal voordelen boven overerving:

- Het biedt een betere inkapseling van gegevens. In het verzamelingen-voorbeeld zorgt delegatie ervoor dat we op een verzamelingobject niet de methoden van ArrayList kunnen aanroepen.
- Het biedt meer flexibiliteit. Tijdens *verwerking* van een applicatie kan worden besloten welke gegevens worden gekoppeld (welke associatie er wordt gelegd). Bij overerving ligt dat vast.

Voordelen van overerving t.o.v. delegatie

Iedere overervingrelatie kan in principe worden omgezet in een delegatierelatie. Toch wordt nog steeds overerving gebruikt omdat het ook een aantal grote voordelen heeft ten opzicht van delegatie:

- Er is minder implementatie nodig omdat de gemeenschappelijke methoden worden geërfd. Bij delegatie zal een klasse een aantal methoden moeten krijgen die niets anders doen dan een verzoek doorgeven aan de gedelegeerde klasse. Stel dat we in bovenstaand

voorbeeld het adres van een student willen wijzigen. Dan is daarvoor in de klasse `Student` een methode `wijzigAdres` nodig. Deze methode wordt nu namelijk niet direct van `Persoon` geërfd. `Student` moet daarom een eigen methode krijgen, die de echte wijziging overlaat aan de methode `wijzigAdres` van de klasse `Persoon`:

```
public void wijzigAdres(String adres) {
    persoon.wijzigAdres(adres);
}
```

– Overerving maakt het mogelijk dynamische binding te gebruiken. We hebben in leereenheid 2 gezien hoe krachtig dat mechanisme is. Zou het in het voorbeeld van de administratie van de universiteit niet mogelijk zijn dat één persoon zowel student als medewerker kan zijn, dan is overerving ook een goede oplossing.

2 De VormenApplicatie

2.1 PRODUCTOMSCHRIJVING

Grafische toepassingen, vooral in de sfeer van industrieel ontwerpen, bouwen een driedimensionale vorm soms op uit elementaire vormen als kubussen, blokken, bollen en cilinders. In deze paragraaf construeren we een applicatie die, zij het op uiterst eenvoudige wijze, ook met enkele van dit soort basisvormen werkt en wel met blokken, bollen en cilinders. Een blok moet hier worden opgevat als een verlengde kubus: twee zijden zijn even lang, de derde kan echter langer of korter zijn.

Productom-
schrijving

De applicatie houdt een verzameling blokken, bollen en cilinders bij en moet de volgende functionaliteit hebben:

- De gebruiker kan een van de basisvormen kiezen, de relevante afmetingen opgeven (zie tabel 3.1) en een nieuwe vorm met die afmetingen aan de verzameling vormen toevoegen.
- De gebruiker moet kunnen zien, welke vormen tot nu toe zijn toegevoegd; bij iedere vorm moet de inhoud worden getoond.
- De gebruiker kan vragen de totale inhoud te berekenen van alle tot nu toe toegevoegde vormen.

Bij het ontwerp moet rekening gehouden worden met mogelijke uitbreidingen, zoals het toevoegen van een nieuw soort vorm.

Tabel 3.1 toont de *formules voor de inhouden* van de gebruikte vormen.

TABEL 3.1 Inhoud van blok, bol en cilinder

<i>vorm</i>	<i>parameters</i>	<i>inhoud</i>
blok	zijde z , hoogte h	z^2h
bol	straal r	$4\pi r^3 / 3$
cilinder	straal r , hoogte h	$\pi r^2 h$

Formules voor
inhoud

OPGAVE 3.9

Splits de taak van deze applicatie in deeltaken (zie de ontwerpmethode uit paragraaf 4 van leereenheid 1).

2.2 TOEVOEGEN VAN NIEUWE VORMEN

Als eerste stap ontwerpt en implementeert u de eerste deeltaak: het toevoegen van een nieuwe vorm.

OPGAVE 3.10

Ontwerp voor deze deeltaak een klassendiagram. Kies eerst de meest voor de hand liggende klassen en breid het diagram dan uit.

We gaan dit deel van de applicatie implementeren (u hoeft dus alleen elementen op te nemen die nodig zijn voor de gebruiksmogelijkheid ‘toevoegen’).

OPDRACHT 3.11

- Open het project Le03Vormen. Dit project bevat al een deel van de implementatie van deze applicatie, namelijk de complete gebruikersinterface. Deze geeft foutmeldingen omdat de domeinklassen nog niet geïmplementeerd zijn.
- Voeg aan de package vormen een klasse Vorm toe. Hoe ziet de implementatie er uit?
- Voeg aan de package vormen klassen Blok, Bol en Cilinder toe en implementeer deze. Kies daarbij zelf een geschikte representatie voor de attributen van deze klassen.
- Implementeer de klasse Vormenlijst.

2.3 STRINGREPRESENTATIES TONEN

De volgende functionele eis is, dat de gebruiker moet kunnen zien welke vormen zijn toegevoegd. We willen de nieuwe vorm, inclusief zijn inhoud, in de interface tonen zodra de gebruiker deze heeft toegevoegd. We willen een algemene oplossing, die ook werkt als we de applicatie later willen uitbreiden met het verwijderen van vormen. Daarom wordt iedere keer als de vormenlijst verandert, een nieuwe stringrepresentatie van de hele lijst met vormen getoond.

OPDRACHT 3.12

- Welke signatuur heeft een methode die een stringrepresentatie van een object genereert? Van welke methode is dit een herdefinitie?
- Welke wijzigingen zijn er nodig in het ontwerp om deze gebruiksmogelijkheid toe te voegen?
- Implementeer deze wijzigingen.

Aanwijzingen

- Ook de inhoud maakt deel uit van de stringrepresentatie; zie tabel 3.1 voor de formules.
- Neem in de stringrepresentatie van de volledige vormenlijst nieuwe regels op. Gebruik daarbij de string “\n”.

2.4 TOTALE INHOUD BEREKENEN

Als laatste toevoeging moet de applicatie op verzoek ook de totale inhoud tonen van alle vormen uit de vormenlijst. De totale inhoud van alle vormen zullen we steeds opnieuw berekenen.

OPGAVE 3.13

Welke wijzigingen zijn nodig in het ontwerp om deze gebruiksmogelijkheid toe te voegen?

Voor we deze uitbreidingen gaan implementeren, staan we even stil bij de methode inhoud van de superklasse Vorm.

Kunnen we deze methode een lege romp geven, zoals neemOp in de klasse Rekening uit het bankvoorbeeld (zie leereenheid 2)?

Nee, dat kan niet: de methode moet immers een waarde opleveren. De klasse Vorm moet een methode inhoud hebben om die methode aan te kunnen roepen op objecten met gedeclareerd type Vorm. Omdat het actuele type van zo'n object altijd tot een van de subklassen behoort, zal deze versie van de methode echter nooit worden uitgevoerd. De implementatie ervan is dus feitelijk zonder betekenis.

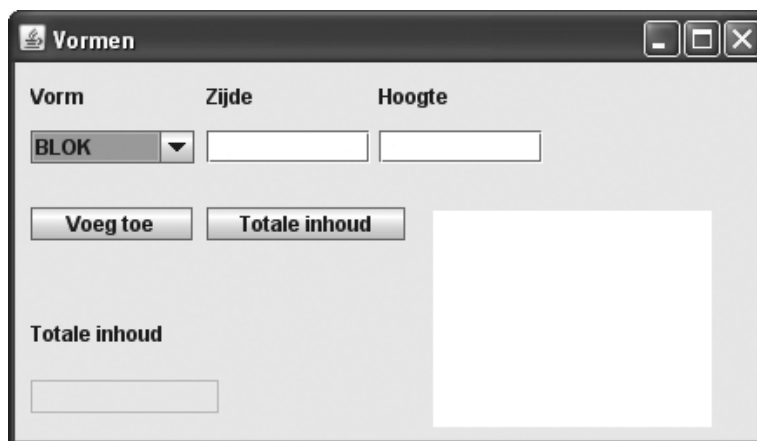
Bij gebrek aan een beter idee, laten we de methode de waarde -1 opleveren. Dit is nogal willekeurig en daarom niet erg bevredigend. Leereenheid 4 zal een betere oplossing presenteren.

OPDRACHT 3.14

- a Implementeer de nieuwe methoden.
- b Test de nu voltooide applicatie. Ook nu kunt u dat doen vanuit de gebruikersinterface die is bevat in de klasse VormenFrame in package vormengui (zie figuur 3.9). Verwerk deze applicatie en test of alles werkt zoals gewenst. Wat test u zoal?

Aanwijzing:

Als u de methoden in de klassen Vormenlijst, Vorm, Blok, Bol en Cilinder exact dezelfde namen heeft gegeven als in de terugkoppeling, zal de applicatie direct werken. Heeft u andere namen gebruikt, dan zult u in de klasse VormenFrame wat kleine wijzigingen moeten aanbrengen.

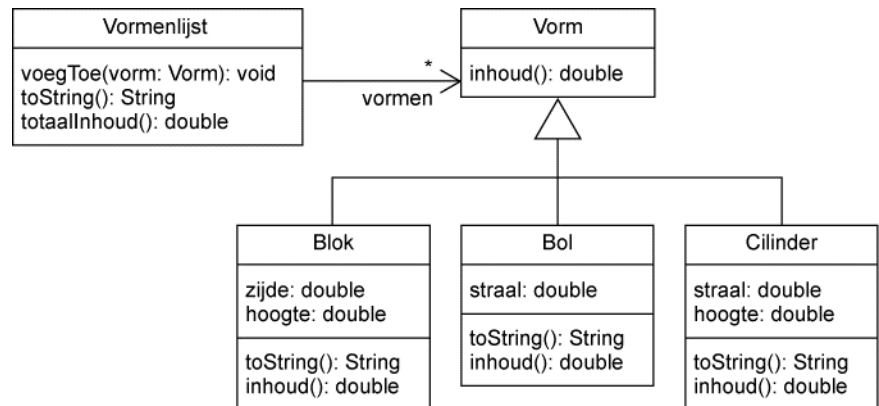


FIGUUR 3.9 De gebruikersinterface voor de VormenApplicatie

2.5 EVALUATIE VAN DE APPLICATIE

Laten we tot slot nog eenmaal goed stilstaan bij het ontwerp en met name de consequenties onderzoeken van andere keuzen.

<p>Alternatief 1</p> <p>Open/closed-principe</p>	<p>Stel dat we uitsluitend een klasse <code>Vorm</code> hadden opgenomen, met daarin een attribuut <code>vormtype</code> dat bijhoudt of de vorm een blok, bol of cilinder is. Die klasse zou een methode <code>inhoud</code> hebben met een switch, die – afhankelijk van de specifieke vorm – de goede formule toepast. Dit kan, maar het heeft een paar belangrijke nadelen ten opzichte van het huidige ontwerp.</p> <p>Ten eerste is het veel beter om code voor bollen, cilinders en blokken te scheiden en onder te brengen in verschillende klassendefinities. Deze toepassing is erg klein, maar in de praktijk zouden we te maken kunnen hebben met wel twintig vormen die elk ook veel meer methoden hebben. Eén grote klasse <code>Vorm</code>, met allemaal methoden die uit een lange switch bestaan, is dan veel onoverzichtelijker dan twintig kleine klassen.</p> <p>Ten tweede is in de huidige opzet uitbreiding met een nieuwe vorm vrij eenvoudig. Als we een kegel of een piramide toe willen voegen, hoeven we – afgezien van de benodigde wijzigingen in de gebruikersinterface en de toevoeging van een methode om de nieuwe vorm te maken in de klasse <code>Vormenlijst</code> – alleen een nieuwe subklasse <code>Kegel</code> of <code>Piramide</code> van <code>Vorm</code> te schrijven. Wijzigingen in de code van bestaande klassen is verder niet nodig. Dat is aantrekkelijk, want juist bij het wijzigen van bestaande code worden snel fouten gemaakt. Hiermee voldoen we aan het eerder genoemde <i>open/closed-principe</i>.</p>
<p>Alternatief 2</p>	<p>Een andere mogelijkheid is om alleen klassen <code>Blok</code>, <code>Bol</code> en <code>Cilinder</code> op te nemen, zonder de superklasse <code>Vorm</code>. De huidige implementatie van <code>Vormenlijst</code> illustreert hoe nadelig dit zou zijn. Juist omdat <code>Vorm</code> bestaat, kunnen we alle elementen in die lijst op dezelfde manier behandelen. Het is daarbij essentieel dat zowel <code>Vorm</code> als alle subklassen van <code>Vorm</code> de methoden <code>inhoud</code> en <code>toString</code> bevat, waarbij <code>toString</code> door klasse <code>Vorm</code> wordt geërfd van de klasse <code>Object</code>.</p>
<p>Alternatief 3</p>	<p>Zou het niet mogelijk zijn om de klasse <code>Vormenlijst</code> weg te laten? De klasse <code>VormenFrame</code> krijgt dan een attribuut <code>vormen</code> van type <code>ArrayList</code>. Het creëren van de vormen en het toevoegen hiervan aan de lijst komt terecht binnen de event handler <code>voegtoeAction</code> terwijl het bepalen van de totale inhoud in <code>totaalInhoudAction</code> wordt gedaan.</p> <p>Het beheren van vormen hoort niet onder het afhandelen van de interactie met de gebruiker. In een ontwerp met een aparte klasse <code>Vormenlijst</code> zijn de verantwoordelijkheden daarom beter gescheiden. De event handlers zelf blijven dan vrij eenvoudig.</p>
<p>Alternatief 4</p>	<p>Het vierde alternatief is verreweg het meest interessant. In het ontwerp uit figuur 3.13 heeft de klasse <code>Vormenlijst</code> drie methoden <code>maakBol</code>, <code>maakBlok</code> en <code>maakCilinder</code>. Voor het toevoegen van een nieuwe vorm is er dus een wijziging nodig in deze klasse. In plaats van de drie genoemde methoden zou <code>Vormenlijst</code> ook een methode <code>voegToe(Vorm)</code> kunnen krijgen die een elders gecreëerde <code>Vorm</code> aan de lijst toevoegt. We tonen dit ontwerp in figuur 3.10.</p>



FIGUUR 3.10 Alternatief ontwerp voor de domeinlaag

Klasse *Vormenlijst* is niet meer verantwoordelijk voor het creëren van *Vormen*, maar alleen voor het beheren van de lijst. Dit ontwerp heeft een belangrijk voordeel.

Het voordeel is, dat bij het toevoegen van een nieuw soort vorm er nu niets aan de klasse *Vormenlijst* hoeft te veranderen (denk weer aan het open/closed-principe). In deze klasse is nu geen enkele kennis meer aanwezig over de subklassen van *Vorm*. Dit maakt klasse *Vormenlijst* eenvoudiger en verlaagt de koppeling tussen klassen binnen de domeinlaag.

Wel moeten we nu nadenken over de vraag waar de diverse vormen dan wel gecreëerd worden. Creatie in de interfacelaag is sowieso geen optie: het creëren van domeinklassen is immers geen taak van deze laag. De oplossing is om creatie aan een aparte klasse in de domeinlaag toe te wijzen. Er is een ontwerppatroon (het Factory-patroon) dat daarvoor gebruikt kan worden; een uitleg daarvan voert nu echter te ver.

Betekent dit dat het oorspronkelijke ontwerp beter is? Dat hangt er een beetje van af. In het algemeen is het een goed idee om een beheersklasse van objecten ook verantwoordelijk te maken voor de creatie ervan. De rest van de applicatie heeft dan alleen met de beheersklasse te maken en niet ook nog met de beheerde objecten. Dat kan echter alleen wanneer deze objecten niet ook elders in de applicatie een rol spelen. We geven twee contrasterende voorbeelden.

In de Bankapplicatie uit leereenheid 1 treedt de klasse *Bank* op als beheerder van instanties van *Rekening*. *Bank* heeft als enige toegang tot de beheerde instanties zodat alle wijzigingen van rekeningen via de bank lopen. De klasse *Bank* moet de *Rekening*-instanties dan ook creëren.

De Swing-klasse *JPanel* treedt op als beheerder van componenten zoals knoppen en tekstvelden (een voorbeeld is de *ContentPane* van de GUI-klassen). Het zou echter erg onhandig zijn als iedere wijziging in een component gerealiseerd moest worden via een methodeaanroep op een *JPanel*: de klasse zou honderden methoden krijgen. Een component kan bovendien van de ene naar de andere container verhuizen. *JPanel* is dus geen exclusieve beheerder. Componenten worden daarom buiten de klasse *JPanel* gecreëerd en pas dan toegevoegd.

Bij de Vormenapplicatie zijn beide ontwerpen verdedigbaar. In deze beperkte context is VormenLijst de exclusieve beheerder. Dat heeft de keuze van het ontwerp bepaald, maar er zijn uitbreidingen denkbaar waarbij het alternatief beter zou zijn. In leereenheid 4 zullen we (om praktische redenen) het andere ontwerp in figuur 3.10 gebruiken.

SAMENVATTING

Paragraaf 1

In paragraaf 1 heeft u van een bestaande klasse Bag een specialisatie Verzameling ontworpen en geïmplementeerd. Herdefinitie van een methode uit de superklasse was daarbij noodzakelijk. Verder heeft u gezien dat overerving niet altijd de meest handige oplossing is om functionaliteit uit een klasse over te nemen; een alternatief is delegatie, waarbij een deel van de functionaliteit van een klasse via een associatie wordt uitbesteed aan een andere klasse.

Paragraaf 2

Bij de tweede toepassing lag de nadruk op dynamische binding. Dankzij dit mechanisme is het mogelijk om methoden te schrijven die instanties van alle subclasses van een superklasse op een uniforme manier behandelen.

In deze toepassing kan de methode totaalInhoud van de klasse Vormenlijst de som van de inhouden van een rij vormen berekenen, zonder na te hoeven gaan van welk type elke vorm is. Iets dergelijks geldt voor de methode toString die een stringrepresentatie van elke vorm maakt.

ZELFTOETS

- 1 De methode contains van de klasse ArrayList gaat na of haar parameter als element in de ArrayList voorkomt. Bekijk de volgende implementatie van deze methode (we hebben deze niet uit de klasse ArrayList overgenomen):

```
/**
 * @param elem the desired element
 * @return true if the specified object is a value of
 *         the collection
 */
public boolean contains(Object elem) {
    for (Object elem2 : elementData) {
        if (elem.equals(elem2)) {
            return true;
        }
    }
    return false;
}
```

De variabele elementData heeft als waarde een array met elementen van type Object, die de waarden bevat die in de ArrayList zijn opgeslagen.

a Gegeven zijn de klassen `Docent` en `Persoon` als gedefinieerd in paragraaf 2 van leereenheid 2. Bekijk nu de volgende reeks opdrachten:

```
ArrayList<Docent> docenten = new ArrayList<>();
Docent d = new Docent("Paauw");
docenten.add(d);
boolean bevat = docenten.contains(d);
```

Wat is de waarde van de variabele `bevat`, na verwerking van deze opdrachten? Leg *precies* uit wat er gebeurt!

NB: ga ervan uit, dat binnen de klassen `Persoon` en `Docent` nog *geen* methode `equals` gedefinieerd is.

b Bekijk nu de volgende reeks opdrachten:

```
ArrayList<Docent> docenten = new ArrayList<>();
Docent d = new Docent("Paauw");
docenten.add(d);
boolean bevat = docenten.contains(new Docent("Paauw"));
```

Wat is nu na verwerking de waarde van de variabele `bevat`?

c Stel nu dat we aan de klasse `Docent` de volgende methode toevoegen:

```
/**
 * @return true wanneer obj een docent is met dezelfde
 *         naam als deze docent
 */
public boolean equals(Object obj) {
    return obj instanceof Docent &&
           getNaam().equals(((Docent) obj).getNaam());
}
```

Wat is de waarde van de variabele `bevat` na verwerking van de reeks opdrachten uit a? En van de reeks opdrachten uit b?

- 2 Een kegel heeft een straal en een hoogte. De inhoud van een kegel met straal r en hoogte h , wordt gegeven door de formule $\pi r^2 h / 3$. Breid de `VormenApplicatie` (inclusief de gebruikersinterface) dusdanig uit, dat deze behalve met blokken, bollen en cilinders, nu ook met kegels kan werken.

TERUGKOPPELING

1 Uitwerking van de opgaven

- 3.1 Er is tweemaal een 1 toegevoegd en eenmaal een 5, vervolgens zijn één 1 en de 5 verwijderd. De bag bevat dus alleen nog een 1 en dus leveren de aanroepen de volgende waarden op:
- a true
 - b false

- 3.2 Een mogelijke implementatie is

```
/**
 * Vergelijkt deze bag met een andere bag.
 * Twee bags zijn gelijk als ze dezelfde elementen
 * bevatten.
 * @param obj een andere bag
 * @return true als de bags gelijk zijn, anders false
 */
public boolean equals(Object obj) {
    if (obj == null && || !(obj instanceof Bag)) {
        return false;
    }
    if (this.afmeting() != ((Bag) obj).afmeting()) {
        return false;
    }
    int[] elems1 = this.elementen();
    int[] elems2 = ((Bag) obj).elementen();
    Arrays.sort(elems1);
    Arrays.sort(elems2);
    return Arrays.equals(elems1, elems2);
}
```

Toelichting:

- Als het object waarmee vergeleken wordt geen Bag is, wordt false teruggegeven.
- Dan wordt gekeken of de twee bags hetzelfde aantal elementen bevatten. Als dat niet het geval is, dan weten we direct dat de bags niet gelijk zijn. Deze test is niet echt noodzakelijk, maar maakt de methode equals sneller voor bags met veel elementen.
- Deze implementatie maakt gebruik van methoden van de klasse Arrays. Eerst worden de arrayrepresentaties van de bags gesorteerd met de methode sort; deze worden vervolgens element voor element met elkaar vergeleken door de methode equals van Arrays.

Er zijn ook heel andere oplossingen mogelijk. Een voorbeeld is de volgende:

- Maak een kopie van de ArrayList met elementen uit de ‘andere’ bag. Daarvoor kan de constructor van ArrayList gebruikt worden met als parameter een reeds bestaande ArrayList (dit is een copy-constructor, vergelijkbaar met die voor Point uit de vorige leereenheid).

– Ga dan voor ieder element uit de bag na of deze ook in de gekopieerde lijst zit. Zo nee, dan zijn de bags verschillend. Zo ja, verwijder het element dan uit de kopielijs. Als alle elementen van de bag doorlopen zijn en ze zijn allemaal gevonden in de kopielijs, dan zijn de bags gelijk.

```
public boolean equals(Object obj) {
    if (obj == null || !(obj instanceof Bag)) {
        return false;
    }
    if (this.afmeting() != ((Bag)obj).afmeting()) {
        return false;
    }
    ArrayList<Integer> kopielijs =
        new ArrayList<>(((Bag)obj).elementen);
    for (int elem : elementen) {
        if (kopielijs.contains(elem)) {
            kopielijs.remove(new Integer(elem));
        } else {
            return false;
        }
    }
    return true;
}
```

Merk op dat bij de aanroep naar remove expliciet een Integer als parameter is meegegeven en geen int-waarde. Het lijkt voor de hand te liggen gebruik te maken van autoboxing, maar dat kan in dit geval niet: de klasse ArrayList heeft ook een methode remove met een int-parameter. Die verwijdert het element met de gegeven index en dat willen we in dit geval niet!

- 3.3 a De methode voegToe van Bag is niet bruikbaar voor een verzameling: we willen immers nu een element alleen toevoegen als dit nog niet voorkwam. We zullen deze methode dus moeten herdefiniëren. De nieuwe specificatie luidt:

```
/**
 * Voegt element toe aan een verzameling; heeft alleen
 * effect als het element nog niet tot de verzameling
 * behoort.
 * @param elem het toe te voegen element
 */
public boolean voegToe(int elem)
```

De methoden afmeting, bevat, elementen, equals, isLeeg, maakLeeg, toString en verwijder zijn zonder meer bruikbaar: de specificatie voor een verzameling zou niet anders luiden dan die voor een bag. Eventueel kan een efficiëntere versie van equals worden gemaakt; we weten nu dat ieder element maar één keer voorkomt. We laten dat echter achterwege. Behalve voegToe krijgt de klasse nog de twee nieuwe methoden vereniging en doorsnede, waarvan de specificatie al is getoond.

b De klasse Bag heeft alleen een (impliciete) parameterloze constructor. Volgens paragraaf 2.3 uit leereenheid 2 hoeft de klasse Verzameling dus niet verplicht een eigen constructor te hebben. Omdat een lege bag ook een lege verzameling is, voldoet de standaardconstructor.

3.4 a Het bestand moet er nu als volgt uitzien:

```
/**
 * De klasse Verzameling representeert een verzameling met
 * gehele getallen. In een verzameling komt ieder element
 * slechts een maal voor.
 */

public class Verzameling extends Bag {

    /**
     * Voegt een nieuw element toe aan de verzameling.
     * @param elem het toe te voegen element
     * @return true als een nieuw element is toegevoegd,
     * anders false
     */
    public boolean voegToe(int elem) {
    }

    /**
     * Berekent de vereniging van de verzameling met een
     * andere verzameling.
     * @param v de andere verzameling
     * @return de vereniging, een nieuwe verzameling
     */
    public Verzameling vereniging(Verzameling v) {
    }

    /**
     * Berekent de doorsnede van de verzameling met een
     * andere verzameling.
     * @param v de andere verzameling
     * @return de doorsnede, een nieuwe verzameling
     */
    public Verzameling doorsnede(Verzameling v) {
    }
}
```

b Het ontwerp van de methode voegToe luidt als volgt:

Als de verzameling elem nog niet bevat
Voeg elem aan de verzameling toe

Bij de implementatie maken we gebruik van de methoden bevat en voegToe van de superklasse. Omdat we voegToe net aan het herdefiniëren zijn, moeten we voor dit laatste het sleutelwoord super gebruiken. De implementatie wordt daarmee als volgt:

```
public boolean voegToe(int elem) {
    if (!bevat(elem)) {
        return super.voegToe(elem);
    }
    return false;
}
```

Om de vereniging te bepalen, gaan we als volgt te werk:

Maak een nieuw nieuwe instantie van Verzameling
Voeg daaraan alle elementen van de huidige verzameling toe
Voeg alle elementen van de tweede verzameling toe.

Alle elementen in een verzameling krijgen we via de methode `elementen`. We moeten de elementen uit de resulterende array een voor een aan de nieuwe verzameling toevoegen.

De implementatie ziet er als volgt uit:

```
public Verzameling vereniging(Verzameling v) {
    Verzameling vereniging = new Verzameling();
    for (int elem : this.elementen()) {
        vereniging.voegToe(elem);
    }
    for (int elem : v.elementen()) {
        vereniging.voegToe(elem);
    }
    return vereniging;
}
```

Voor de doorsnede doorlopen we de elementen van een verzameling en bekijken of die ook tot de tweede verzameling behoren. Alle elementen waarvoor dat geldt, vormen de doorsnede. Ga zelf na dat het niet uitmaakt welke verzameling doorlopen wordt.

NB Als we letten op efficiëntie, dan kunnen we beter de kleinste verzameling nemen.

Het ontwerp van deze methode is dus:

```
Maak een nieuwe instantie van Verzameling
Voor elk element in de huidige verzameling:
    als dit element ook in de tweede verzameling voorkomt,
        voeg het dan aan de nieuwe verzameling toe
```

De implementatie ziet er als volgt uit:

```
public Verzameling doorsnede(Verzameling v) {
    Verzameling doorsnede = new Verzameling();
    for (int elem : this.elementen()) {
        if (v.bevat(elem)) {
            doorsnede.voegToe(elem);
        }
    }
    return doorsnede;
}
```

c Deze implementatie voegt de elementen van het object waarop de methode `vereniging` is aangeroepen, toe aan de parameter `v` en wijzigt `v` daardoor. Dat is niet de bedoeling.

Stel bijvoorbeeld dat `v1` gelijk is aan `{1, 2, 3}` en `v2` aan `{3, 4, 5}`. Het resultaat van de aanroep `v1.vereniging(v2)` moet dan gelijk zijn aan `{1, 2, 3, 4, 5}`, terwijl nog steeds moet gelden dat `v1 = {1, 2, 3}` en `v2 = {3, 4, 5}`. Na de aanroep van `vereniging` wijzen de formele parameter `v` en de actuele parameter `v2` beide naar hetzelfde object (de verzameling `{3, 4, 5}`).

Omdat we daar de elementen van `{1, 2, 3}` aan toevoegen, is ook `v2` na afloop gelijk aan `{1, 2, 3, 4, 5}`.

- 3.5 a De methode om elementen uit een verzameling te verwijderen, is geërfd van de klasse `Bag`. Indien deze klasse goed functioneert, zou het verwijderen van een element uit een verzameling ook goed moeten gaan. We testen echter liever te veel dan te weinig en stellen dus ook voor deze gebruiksmogelijkheid testwaarden op. We proberen gevallen dat het element wel en dat het element niet in de verzameling zit.

Ook kijken we of het goed gaat wanneer de verzameling door het verwijderen van dit element leeg wordt, en wanneer de verzameling al leeg was:

- verwijder 4 uit {1, 2, 4}
- verwijder 3 uit {1, 2}
- verwijder 2 uit {2}
- verwijder 2 uit de lege verzameling.

b Ook deze methode is geërfd van Bag. We proberen het toch uit:

- maak {1, 2, 4} leeg
- maak {1} leeg
- maak de lege verzameling leeg.

Dat laatste klinkt triviaal, maar het moet wel goed gaan!

c Bij vereniging is het van belang of de verzamelingen elementen gemeen hebben en ook of een van de verzamelingen leeg is of zelfs allebei.

We proberen de volgende verenigingen uit:

- $\{-2, -1, 0\} \cup \{1, 2, 3, 4\}$
- $\{2, 3, 5, 7, 11\} \cup \{1, 3, 5, 7, 9\}$
- $\{2, 3, 5, 7, 11\} \cup \{2, 3, 5, 7, 11\}$
- $\{2, 3, 5, 7, 11\} \cup \text{LEEG}$
- $\text{LEEG} \cup \{-13\}$
- $\text{LEEG} \cup \text{LEEG}$.

d Voor de doorsnede tenslotte gelden ongeveer dezelfde overwegingen, we kunnen dezelfde testwaarden gebruiken:

- $\{-2, -1, 0\} \cap \{1, 2, 3, 4\}$
- $\{2, 3, 5, 7, 11\} \cap \{1, 3, 5, 7, 9\}$
- $\{2, 3, 5, 7, 11\} \cap \{2, 3, 5, 7, 11\}$
- $\{2, 3, 5, 7, 11\} \cap \text{LEEG}$
- $\text{LEEG} \cap \{-13\}$
- $\text{LEEG} \cap \text{LEEG}$.

e Voor de test op gelijkheid kunnen we weer dezelfde testwaarden gebruiken. Wat we extra moeten testen is of twee verzamelingen gelijk zijn waarin dezelfde waarden zitten die in verschillende volgorde zijn toegevoegd. We testen de volgende paren op gelijkheid:

- $\{-2, -1, 0\}$ en $\{1, 2, 3, 4\}$
- $\{2, 3, 5, 7, 11\}$ en $\{1, 3, 5, 7, 9\}$
- $\{2, 3, 5, 7, 11\}$ en $\{2, 3, 5, 7, 11\}$
- $\{2, 3, 5, 7, 11\}$ en LEEG
- LEEG en $\{-13\}$
- LEEG en LEEG
- $\{1, 2\}$ en $\{2, 1\}$
- $\{1, 2, 3, 4\}$ en $\{1, 4, 2, 3\}$.

3.6 Geen terugkoppeling.

- 3.7 a De grootste verschillen met de oorspronkelijke op delegatie gebaseerde implementatie zijn:
- Er is geen attribuut elementen nodig; de bag is nu immers zelf een ArrayList.
 - We roepen nu de methoden van ArrayList direct aan (we erven deze nu namelijk), en niet op het attribuut elementen.

De implementatie van de methoden kan gemakkelijk uit die van Bag worden afgeleid, door iedere referentie naar het attribuut elementen te vervangen door een referentie naar this.

We tonen een deel van de implementatie. De uitwerking van het project bevat een klasse Bag2 met de volledige implementatie.

```
public class Bag extends ArrayList<Integer> {

    // er zijn geen attributen
    public boolean voegToe(int elem) {
        return add(elem);
    }

    ...
    // ook de implementaties van verwijder, afmeting, bevat,
    // isLeeg en maakLeeg roepen direct een methode van de
    // superklasse ArrayList aan

    public String toString() {
        if (isEmpty()) {
            return "LEEG";
        }
        String s = "";
        int teller = 0;
        for (int elem : this) {
            teller++;
            ...
        }
        return s;
    }

    ...
}
```

De gemarkeerde regel is het meest interessant: in plaats van 'elem : elementen' staat er nu 'elem : this'. Dat is (voor een ArrayList of een andere structuur waarover geïtereerd kan worden) correct Java.

b Nee. Er hoeft niets veranderd te worden in de klasse Verzameling. De interface van Bag is niet gewijzigd. Hoe de interne implementatie van de klasse Bag is, is voor Verzameling niet interessant.

- 3.8 Naast de methoden die de klasse Verzameling erft van Bag, erft de klasse Verzameling ook alle protected en public methoden van de klasse ArrayList. Dat is in dit geval niet gewenst. Eén van de methoden die Verzameling erft is de methode add(E e) waarmee een object (in ons geval een integer) kan worden toegevoegd aan de verzameling. Deze methode houdt geen rekening met de definitie van een verzameling: dat een element in een verzameling maar één keer mag voorkomen. De volgende code is mogelijk:

```
Verzameling v = new Verzameling();
v.voegToe(3);
v.voegToe(1);
v.add(3);
```

Na afloop van dit stukje code bevat de verzameling twee maal het element 3. Dat is in strijd met de definitie van een verzameling. Het is daarom ongewenst dat verzameling de methode add van ArrayList erft.

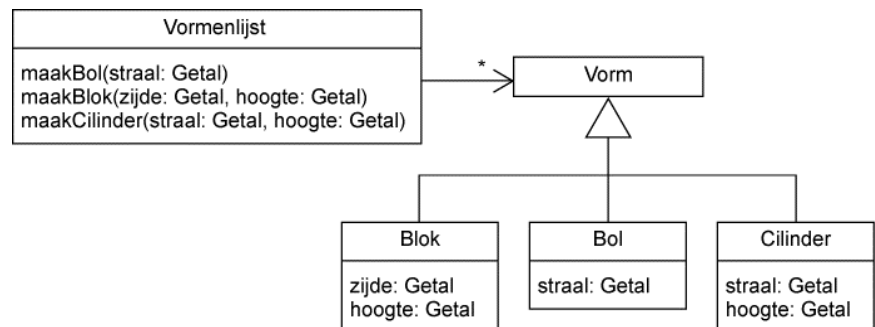
Het nadeel geldt ook al voor de klasse Bag: die erft bijvoorbeeld de methoden `indexOf` en `lastIndexOf` van `ArrayList`. Voor een bag hebben die geen betekenis.

- 3.9 In de productomschrijving staan drie gebruiksmogelijkheden genoemd:
- een nieuwe vorm met bepaalde afmetingen aan de verzameling vormen toevoegen
 - tonen welke vormen tot nu toe zijn toegevoegd, inclusief de inhoud van iedere vorm
 - de totale inhoud berekenen.

- 3.10 Er zijn in dit geval drie onmiddellijk voor de hand liggende klassen, namelijk voor de vormen Blok, Bol en Cilinder. De klassen Blok, Bol en Cilinder krijgen in elk geval hun afmetingen als attributen: voor een blok zijn dat een zijde en een hoogte, voor een bol alleen een straal en voor een cilinder een straal en een hoogte.

Hoewel de drie klassen geen overeenkomstige attributen hebben die naar een superklasse verhuisd kunnen worden, nemen we toch meteen ook een superklasse Vorm op in het ontwerp. Hiermee introduceren we namelijk de mogelijkheid om de verschillende vormen via dynamische binding op dezelfde manier te behandelen. Deze superklasse heeft dus dezelfde functie als de superklasse Rekening in het bankvoorbeeld uit leereenheid 2.

Vervolgens constateren we, dat er sprake is van een verzameling vormen waaraan we er één toe moeten kunnen voegen. Voor het beheer van die verzameling introduceren we een klasse Vormenlijst. Deze heeft dus een functie vergelijkbaar met Bank in het bankvoorbeeld. De klasse Vormenlijst is verantwoordelijk voor het maken en beheren van alle vormen. Figuur 3.11 toont het klassendiagram.



FIGUUR 3.11 Klassendiagram voor de VormenApplicatie

- 3.11 a Geen terugkoppeling.
b De klasse Vorm krijgt (vooralsnog) een lege implementatie. Op dit moment wordt deze klasse alleen nog gebruikt om de verschillende typen vormen in één lijst te kunnen stoppen.

```

package vormen;

public class Vorm {
}

```

c De afmetingen van een vorm hoeven niet noodzakelijk gehele getallen te zijn. We representeren de attributen dus als waarden van het type `double`. De implementaties zijn verder zeer eenvoudig. De klassen hebben voorlopig alleen een constructor nodig.

```
package vormen;

public class Blok extends Vorm {
    private double zijde = 0.0;
    private double hoogte = 0.0;

    public Blok(double zijde, double hoogte) {
        this.zijde = zijde;
        this.hoogte = hoogte;
    }
}

package vormen;

public class Bol extends Vorm {
    private double straal = 0.0;

    public Bol(double straal) {
        this.straal = straal;
    }
}

package vormen;

public class Cilinder extends Vorm {
    private double straal = 0.0;
    private double hoogte = 0.0;

    public Cilinder(double straal, double hoogte) {
        this.straal = straal;
        this.hoogte = hoogte;
    }
}
```

d Volgens het klassendiagram uit figuur 3.11 bevat een `Vormenlijst` 0 of meer instanties van `Vorm`. De klasse `Vormenlijst` moet dus een attribuut krijgen, dat nul of meer vormen kan bevatten. We zullen dit attribuut implementeren als een `ArrayList` van `Vorm`. We weten immers niet van te voren hoeveel vormen de lijst zal bevatten en dus is een array minder handig. Deze `ArrayList` zal straks instanties bevatten van `Blok`, van `Bol` en van `Cilinder`.

De implementatie van de klasse ziet er daarmee voorlopig als volgt uit:

```
import java.util.ArrayList;

/**
 * Representeert een lijst met vormen.
 */
public class Vormenlijst {

    private ArrayList<Vorm> vormen = new ArrayList<>();

    /**
     * Maakt een bol-instantie en voegt deze toe aan
     * de vormenlijst.
     * @param straal straal van de bol
     */
    public void maakBol(double straal) {
        Bol bol = new Bol(straal);
        vormen.add(bol);
    }

    /**
     * Maakt een blok-instantie en voegt deze toe aan
     * de vormenlijst.
     * @param zijde zijde van blok (lengte en breedte)
     * @param hoogte hoogte van blok
     */
    public void maakBlok(double zijde, double hoogte) {
        Blok blok = new Blok(zijde, hoogte);
        vormen.add(blok);
    }

    /**
     * Maakt een cilinder-instantie en voegt deze toe
     * aan de vormenlijst.
     * @param straal straal van de cilinder
     * @param hoogte hoogte van de cilinder
     */
    public void maakCilinder(double straal, double hoogte) {
        Cilinder cilinder = new Cilinder(straal, hoogte);
        vormen.add(cilinder);
    }
}
```

- 3.12 a Een methode die een stringrepresentatie van een object genereert, heeft altijd de volgende signatuur:

```
public String toString()
```

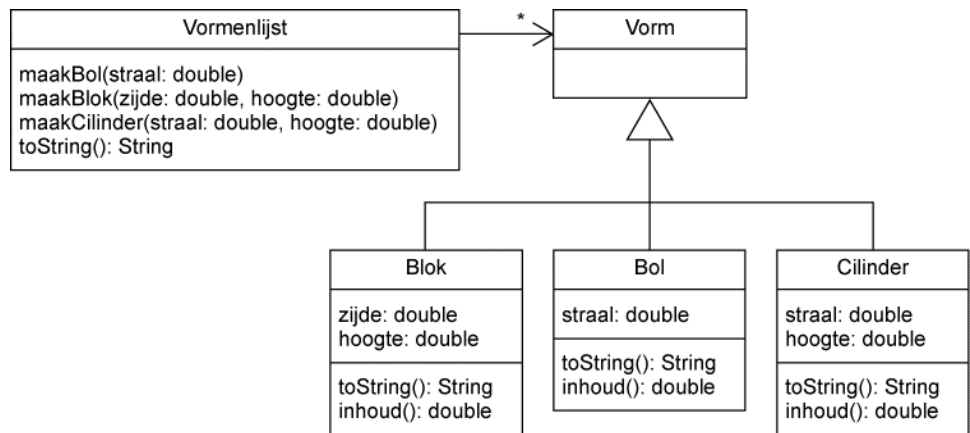
Dit is een herdefinitie van de gelijknamige methode van de klasse Object.

b De klasse `Vormenlijst` krijgt een methode `toString` die een string-representatie van alle vormen moet opleveren. Om de stringrepresentatie van een enkele vorm te bepalen, geven we iedere subklasse van `Vorm` ook een methode `toString`.

Het is in dit geval niet nodig om de klasse `Vorm` ook een methode `toString` te geven: deze wordt door `Vorm` al geërfd van `Object`. We kunnen de methode `toString` dus aanroepen op een instantie van `Vorm` zonder te hoeven weten om wat voor vorm het precies gaat.

Omdat de methode `toString` van iedere vorm ook de inhoud van de vorm moet tonen, geven we alle vormklassen ook een parameterloze methode `inhoud` die een waarde van het type `double` oplevert.

Figuur 3.12 toont het nieuwe klassendiagram. Merk op dat het op dit moment niet noodzakelijk is om klasse `Vorm` een (lege) implementatie van methode `inhoud` te geven. De methode `inhoud` wordt alleen nog maar binnen de verschillende vormklassen gebruikt en nog niet via dynamische binding via `Vorm`.



FIGUUR 3.12 Klassendiagram voor `VormenApplicatie`, versie 2

c Voor de implementatie van deze gebruiksmogelijkheid zijn de volgende wijzigingen nodig:

– Voeg aan de klasse `Blok` de volgende methoden toe:

```

/**
 * Levert een stringrepresentatie van het blok.
 * @return een stringrepresentatie van het blok
 */
public String toString() {
    return "Blok " + zijde + " " + hoogte +
        " (" + inhoud() + ")";
}

/**
 * Levert de inhoud van het blok.
 * @return de inhoud van het blok
 */
public double inhoud() {
    return zijde * zijde * hoogte;
}
  
```

– Voeg aan de klasse Bol de volgende methoden toe:

```
/**
 * Levert een stringrepresentatie van de bol.
 * @return een stringrepresentatie van de bol
 */
public String toString() {
    return "Bol " + straal +
        " (" + inhoud() + ")";
}

/**
 * Levert de inhoud van de bol.
 * @return de inhoud van de bol
 */
public double inhoud() {
    return (4 * Math.PI * straal * straal * straal) / 3;
}
```

– Voeg aan de klasse Cilinder de volgende methoden toe:

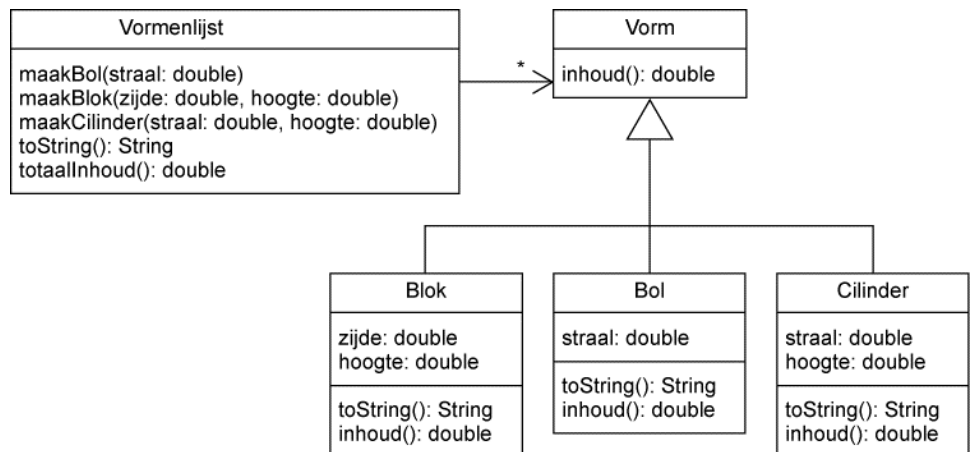
```
/**
 * Levert een stringrepresentatie van de cilinder.
 * @return een stringrepresentatie van de cilinder
 */
public String toString() {
    return "Cilinder " + straal + " " + hoogte +
        " (" + inhoud() + ")";
}

/**
 * Levert de inhoud van de cilinder.
 * @return de inhoud van de cilinder
 */
public double inhoud() {
    return Math.PI * straal * straal * hoogte;
}
```

– De meest interessante toevoeging is die aan de klasse Vormenlijst. Om de stringrepresentatie van de hele lijst te berekenen, hoeven we niet voor iedere vorm na te gaan van welke soort die is. We hoeven alleen op alle objecten uit de lijst de methode toString aan te roepen. Dynamische binding zorgt ervoor dat precies de juiste implementatie wordt verwerkt. De implementatie van toString in Vormenlijst is dus vrij eenvoudig.

```
/**
 * Levert een stringrepresentatie van de lijst met vormen.
 * @return een stringrepresentatie van de lijst met vormen
 */
public String toString() {
    String stringlijst = "";
    for (Vorm vorm : vormen) {
        stringlijst = stringlijst + vorm.toString() + "\n";
    }
    return stringlijst;
}
```

- 3.13 – De totale inhoud van alle vormen moet op verzoek berekend kunnen worden. De berekening is een verantwoordelijkheid voor de klasse `Vormenlijst`, deze krijgt daarvoor een methode `totaalInhoud`.
 – Bij het berekenen willen we gebruik maken van dynamische binding. We roepen steeds de methode `inhoud` aan op een vorm uit de lijst. Om dit mogelijk te maken moet de klasse `Vorm` een methode `inhoud` krijgen. Figuur 3.13 toont het nieuwe – en meteen ook definitieve – ontwerp.



FIGUUR 3.13 Het definitieve ontwerp van de VormenApplicatie

- 3.14 a De implementatie verloopt als volgt:

– Voeg aan de klasse `Vorm` de volgende methode toe:

```

/**
 * Deze methode moet in iedere subklasse geherdefinieerd
 * worden en dan de inhoud van de betreffende vorm
 * opleveren.
 * @return -1 (bij gebrek aan beter)
 */
public double inhoud() {
    return -1;
}

```

– Ook hier is weer de toevoeging aan de klasse `Vormenlijst` de meest interessante. De implementatie van de methode `totaalInhoud` volgt hetzelfde patroon als die van `toString`: op elk element van de lijst wordt de methode `inhoud` aangeroepen en dynamische binding zorgt ervoor dat de juiste berekening wordt uitgevoerd.

```

/**
 * Levert de som van de inhouden van alle vormen in de
 * lijst.
 * @return de som van de inhouden van alle vormen in de
 *         lijst
 */
public double totaalInhoud() {
    double inhoud = 0;
    for (Vorm vorm : vormen) {
        inhoud = inhoud + vorm.inhoud();
    }
    return inhoud;
}

```


b Begin met een grensgeval: wat gebeurt er als u de totaalinhoud wilt berekenen terwijl er nog geen vormen zijn gedefinieerd? Als het goed is, verschijnt er 0 in het tekstveld voor de Totale inhoud.

Voeg vervolgens van iedere vorm een instantie toe, om te beginnen met alle afmetingen 1. De waarde van de inhoud is dan gemakkelijk af te leiden: voor een blok is het 1, voor een bol $4\pi/3$ (ongeveer 4.19), voor een cilinder π (ongeveer 3.14). De totale inhoud van deze drie vormen is ongeveer 8.33.

Voeg dan voor iedere vorm een instantie toe met afmetingen 0 en controleer of de inhoud 0 zijn en de totale inhoud niet verandert.

Probeer tot slot nog wat andere afmetingen. Controleer de inhoud en de totale inhoud met behulp van een rekenmachine.

2 Uitwerking van de zelftoets

- 1 a Na verwerking van de eerste drie opdrachten, bevat de ArrayList docenten één element, met index 0: de instantie van Docent die ook de waarde is van de variabele d. Binnen de ArrayList docenten geldt dus dat `elementCount = 1`, `elementData = {d}`. De aanroep `docenten.contains(d)` leidt tot één aanroep naar `equals` en wel `d.equals(elementData[0])` ofwel `d.equals(d)`.

Omdat noch Docent noch de superklasse Persoon een methode `equals` heeft, wordt de methode `equals` van de superklasse Object aangeroepen. Deze constateert dat beide referenties gelijk zijn en levert dus `true` op, waardoor `contains` eveneens `true` oplevert.

Samenvattend: Na verwerking geldt dat `bevat = true`; de vergelijking binnen de methode `contains` wordt uitgevoerd door de methode `equals` van Object.

b Ook nu wordt de vergelijking uitgevoerd door de methode `equals` van Object. Het resultaat is nu echter `false`, omdat de betrokken referenties niet gelijk zijn.

c Nu wordt de vergelijking uitgevoerd door de methode `equals` van Docent. Deze vergelijkt met behulp van de methode `equals` van String de namen van de betrokken instanties. In beide gevallen zijn die gelijk en dus wordt nu in beide gevallen de waarde `true` opgeleverd.

- 2 Allereerst moeten we een klasse Kegel definiëren, als nieuwe subklasse van Vorm. De definitie lijkt op die van Blok, Bol en Cilinder:

```
public class Kegel extends Vorm {
    private double straal = 0.0;
    private double hoogte = 0.0;

    public Kegel(double straal, double hoogte) {
        this.straal = straal;
        this.hoogte = hoogte;
    }

    /**
     * Levert een stringrepresentatie van de kegel.
     * @return een stringrepresentatie van de kegel
     */
    public String toString() {
        return "Kegel " + straal + " " + hoogte
            + " (" + inhoud() + ")";
    }
}
```

```

/**
 * Levert de inhoud van de kegel.
 * @return de inhoud van de kegel
 */
public double inhoud() {
    return (Math.PI * straal * straal * hoogte) / 3;
}

```

Dan moet het mogelijk zijn om een kegel toe te voegen aan de vormenlijst. Daartoe dient de klasse `Vormenlijst` uitgebreid te worden met de volgende methode:

```

public void maakKegel(double straal, double hoogte) {
    Kegel kegel = new Kegel(straal, hoogte);
    vormen.add(kegel);
}

```

Verder zijn er een aantal wijzigingen nodig in de gebruikersinterface, de klasse `VormenFrame`:

- Er moet een constante `KEGEL` worden toegevoegd aan de enumeratie:

```

public enum Vormkeuze {BLOK, BOL, CILINDER, KEGEL};

```

- Om in de gebruikersinterface naast een blok, bol of cilinder ook een kegel te kunnen kiezen, moet de nieuwe vorm worden toegevoegd aan de combobox. Omdat echter in `mijnInit` in een `for-lus` alle mogelijke waarden uit de definitie van `Vormkeuze` worden gehaald en aan de combobox worden toegevoegd, hoeft u hiervoor zelf niets te doen.

- Bij keuze van kegel in `vormKeuze`, moet de interface invoervelden tonen met labels `straal` en `hoogte`. Hiertoe moet de event handler `selecteerVormAction` worden aangepast. Omdat de interface er voor kegels en cilinders hetzelfde uitziet, kan dit gemakkelijk door in de `switch-opdracht` de gevallen `CILINDER` en `KEGEL` op dezelfde wijze af te handelen. De code voor de event handler wordt dan als volgt:

```

void selecteerVormAction() {
    ...
    switch (keuze) {
        ...
        case CILINDER:
        case KEGEL:
            afmeting1Label.setText("Straal");
            afmeting1Label.setVisible(true);
            afmeting1TextField.setVisible(true);
            afmeting2Label.setText("Hoogte");
            afmeting2Label.setVisible(true);
            afmeting2TextField.setVisible(true);
            break;
        ...
    }
    ...
}

```

– Tot slot moet de event handler `voegtoeAction` worden aangepast. De mogelijkheid om een nieuwe kegel te maken, moet worden toegevoegd. Dit betekent het toevoegen van een geval aan de switch-opdracht. De code wordt als volgt:

```
void voegtoeAction() {
    ...
    switch (keuze) {
        ...
        case KEGEL:
            straal = Double.parseDouble(
                afmeting1TextField.getText());
            hoogte = Double.parseDouble(
                afmeting2TextField.getText());
            vormen.maakKegel(staal, hoogte);
            break;
        default:
            ...
    }
    ...
}
```

Aan de superklasse `Vorm` hoeft niets gewijzigd te worden. Deze klasse hoeft zelfs niet opnieuw gecompileerd te worden! Mogelijk had u moeite om uit te zoeken wat u nu precies waar moest wijzigen in de klasse `VormenFrame`. Dit onderstreept nog eens de wenselijkheid om switches die onderscheid maken tussen de verschillende typen zoveel mogelijk te vermijden.

Abstracte klassen en interfaces

Introductie 133

Leerkern 134

- 1 Abstracte klassen 134
 - 1.1 Abstracte klassen en methoden 134
 - 1.2 Syntaxis van klassen- en methodedefinities 135
 - 1.3 Ontwerppatronen 136
- 2 Interfaces 139
 - 2.1 Abstracte klasse als contract 139
 - 2.2 Het concept interface 142
 - 2.3 Waarom interfaces? 145
- 3 Interfaces in de praktijk 148
 - 3.1 Markeerinterfaces 148
 - 3.2 Eigenschappen afdwingen 149
 - 3.3 Kiezen tussen implementaties 149

Samenvatting 150

Zelftoets 151

Terugkoppeling 153

- 1 Uitwerking van de opgaven 153
- 2 Uitwerking van de zelftoets 155

Abstracte klassen en interfaces

INTRODUCTIE

In leereenheid 3 liepen we in de VormenApplicatie, bij implementatie van de klasse `Vorm`, tegen een probleem aan. Het is noodzakelijk dat `Vorm` een methode inhoud heeft, maar eigenlijk was er geen logische implementatie voor deze methode. Het is immers de bedoeling dat deze methode in *iedere* subklasse van `Vorm` geherdefinieerd wordt, dus willen we het liefst helemaal geen implementatie geven. Een dergelijke situatie komt in de praktijk zo vaak voor, dat Java een aparte constructie biedt die het mogelijk maakt een dergelijke methode niet te implementeren: de abstracte klasse. Een andere mogelijkheid die Java biedt bij het ontwerpen van programma's is het gebruik van zogeheten interfaces. Abstracte klassen en interfaces zijn te beschouwen als contracten. Iedere klasse die zich aan dat contract houdt, verwerft daardoor bepaalde rechten.

Andere hulpmiddelen die gebruikt kunnen worden bij het ontwerpen van een programma zijn ontwerp patronen. Ontwerppatronen bieden een standaardoplossing voor bekende ontwerp problemen.

In deze leereenheid worden de twee taalconcepten 'abstracte klasse' en 'interface' behandeld en maakt u kennis met enkele ontwerp patronen. In paragraaf 1 wordt aan de hand van het voorbeeld over vormen uit leereenheid 3 bekeken wat het nut is van abstracte klassen en abstracte methoden. Ook leert u wat ontwerp patronen zijn en maakt u kennis met het Composite-patroon. Paragraaf 2 gaat uitgebreid in op het begrip interface. Paragraaf 3 staat stil bij een aantal mogelijkheden die interfaces bieden. De meeste voorbeelden komen uit de Java API.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u

- de begrippen abstracte klasse en abstracte methode kunt omschrijven en kunt uitleggen wat het nut daarvan is
- zelf een abstracte klasse kunt maken
- het nut van een contract bij het programmeren kunt uitleggen
- het begrip interface kunt omschrijven en kunt uitleggen wat het nut daarvan is
- gebruik kunt maken van een bestaande interface en zelf een interface kunt schrijven
- een aantal toepassingen van interfaces kunt noemen
- kunt uitleggen wat een ontwerp patroon is
- het Composite-patroon kunt beschrijven en een voordeel kunt aangeven van het gebruik van het patroon
- de betekenis kent van de volgende kernbegrippen: enkelvoudige overerving, meervoudige overerving, markeerinterface.

Studeeraanwijzingen

De studielast van deze leereenheid bedraagt circa 6 uur.

L E E R K E R N

1 **Abstracte klassen**

1.1 ABSTRACTE KLASSEN EN METHODEN

De klasse *Vorm* zoals geïmplementeerd in leereenheid 3 met een willekeurige implementatie van methode *inhoud* kan aanleiding zijn tot fouten. We illustreren dit in de volgende opgave.

OPGAVE 4.1

Stel dat u, bij het intypen van de klasse *Blok*, een foutje had gemaakt en *Blok* daarmee een methode *imhoud* in plaats van *inhoud* had gegeven.

- a Leg uit dat de compiler deze fout niet opmerkt.
- b Wat gebeurt er als u in de applicatie een nieuw blok toevoegt?

In Java bestaan voor dergelijke gevallen abstracte klassen en abstracte methoden.

Abstracte klasse

Een *abstracte klasse* is een klasse die zelf *niet* geïntanceerd mag worden.

Abstracte methode

Een abstracte klasse kan *abstracte methoden* bevatten. Van zo'n methode wordt alleen de signatuur in de klasse opgenomen. De feitelijke implementatie wordt aan de subklassen overgelaten. Opdat de implementator van de subklasse weet wat de methode moet doen, is het wel aan te raden om de methoden goed te specificeren.

Voorbeeld

De klasse *Vorm* kan als volgt als abstracte klasse worden gedeclareerd:

```
public abstract class Vorm {

    /**
     * Berekent na herdefinitie de inhoud van
     * een specifiek soort vorm.
     * @return de inhoud van de vorm
     */
    public abstract double inhoud();
}
```

Vorm mag nu zelf nooit geïntanceerd worden. In iedere (niet abstracte) subklasse van *Vorm* moet een methode *inhoud* zijn opgenomen met dezelfde signatuur en overeenkomstig de hier gegeven specificatie.

OPDRACHT 4.2

Open het project *Le04Vormen*. Deze applicatie heeft dezelfde functionaliteit als de *Vormen*applicatie uit leereenheid 3.

Nu is echter niet het oorspronkelijke maar een alternatief ontwerp geïmplementeerd. We komen daar in paragraaf 2 op terug.

- a Maak de klasse *Vorm* op de hierboven aangegeven wijze tot een abstracte klasse (met *inhoud* als abstracte methode) en verwerk de applicatie.
- b Wijzig nu bovendien de naam van de methode *inhoud* van de klasse *Blok* in *imhoud*. Begrijpt u de foutmelding?

Het definiëren van *Vorm* als abstracte klasse biedt dus twee voordelen:

- Ten eerste is die vervelende ‘return –1’ niet meer nodig: in de klasse *Vorm* hoeft de methode inhoud nu helemaal niet meer geïmplementeerd te worden.
- Ten tweede worden we nu gedwongen om deze methoden in concrete subklassen te definiëren, waardoor een verkeerd werkend programma als in opgave 4.1, vermeden wordt.

Een klasse die een abstracte methode bevat, moet zelf ook abstract zijn, maar omgekeerd hoeft dat niet. Een abstracte klasse kan ook niet-abstracte methoden bevatten.

OPGAVE 4.3

- Is het zinvol om van de klasse *Rekening* uit de banksimulatie in leereenheid 2 een abstracte klasse te maken en van methode *neemOp* een abstracte methode?
- In opgave 2.17 uit leereenheid 2 stelden we voor om deze klasse methoden *eindeMaand* en *eindeJaar* te geven met een lege implementatie. Is het zinvol om deze methoden abstract te maken?

Als een methode een zinvolle standaard-implementatie heeft, dus een implementatie die in elk geval voor sommige subklassen voldoet, dan is het beter om die methode niet abstract te maken. Er is dan immers geen reden om de implementator van een subklasse te dwingen om voor die methode een eigen implementatie op te nemen.

1.2 SYNTAXIS VAN KLASSEN- EN METHODEDEFINITIES

Klassedefinitie

We geven de syntaxis van klassen- en methodedefinities, zoals we die tot nu toe hebben leren kennen.

Syntaxis

```
[public] [final] [abstract] class klassennaam
[extends klassennaam ] {
    [attribuutdeclaraties]
    [constructordefinities]
    [methodedefinities]
}
```

Abstract en final mogen niet samen voorkomen: een abstracte klasse is bedoeld als superklasse en een dergelijke klasse kan dus onmogelijk final zijn.

Methodedefinitie

De syntaxis van een *methodedefinitie* kunnen we als volgt weergeven:

Syntaxis

```
[toegang] [static] [final] terugkeertype
    methodenaam ( formele-parameterlijst )
    blok
```

of

```
[toegang] abstract terugkeertype
    methodenaam ( formele-parameterlijst ) ;
```

Het eerste geval betreft de definitie van een niet-abstracte methode. Deze definitie bestaat uit een signatuur plus een romp – het blok – waarin een implementatie van de methode is bevat. Het tweede geval betreft de definitie van een abstracte methode. Van deze methode wordt alleen de signatuur gegeven.

Een *private* methode kan niet gheredefinieerd worden en kan dus ook niet abstract zijn. Een abstracte methode kan alleen voorkomen in een abstracte klasse. Een abstracte methode kan niet static zijn.

In een UML-klassendiagram kan op twee manieren aangegeven worden dat een klasse of een methode abstract is: of door de klassennaam of methodesignatuur cursief te schrijven, of door achter de klassennaam of signatuur de aanduiding {abstract} te zetten.

1.3 ONTWERPPATRONEN

De introductie van abstracte klassen biedt een gelegenheid voor de introductie van een ander belangrijk OO-concept, namelijk het ontwerp-patroon. Als voorbeeld daarvan bekijken we het Composite-patroon.

In leereenheid 1 is een aantal wenselijke eigenschappen van programma's genoemd, met name scheiding van verantwoordelijkheden, lokaliteit en eenvoud. Een programmeur kan vaak op grond van zijn ervaring tot een goed ontworpen programma komen. Wie nog niet zoveel ervaring heeft kan gebruikmaken van de ervaring van anderen. Veel voorkomende typen software-ontwerpproblemen zijn namelijk als ontwerp-patroon in kaart gebracht, samen met een bijbehorend oplossingsmodel. Een beginnend programmeur kan met behulp van ontwerp-patronen bekende ontwerpproblemen leren herkennen en het bijbehorende model toepassen in zijn eigen ontwerp.

Ontwerppatroon

Engels: Design pattern

Een *ontwerppatroon* beschrijft een vaak optredend probleem bij het ontwerpen van OO-programma's en geeft een schematische oplossing voor dat probleem, in termen van een relatief klein aantal klassen en relaties tussen deze klassen. Deze oplossing is gebaseerd op bestaande OO-implementaties die in de praktijk succesvol zijn gebleken. Het is geen concreet ontwerp, maar een globaal model dat in verschillende situaties kan worden toegepast.

Composite-patroon

Het *Composite-patroon* is een ontwerp-patroon om elementen van een complexe structuur op hiërarchische wijze te ordenen.

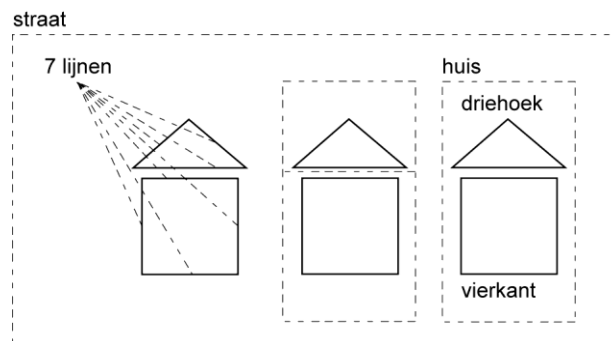
We introduceren het Composite-patroon met een voorbeeld.

OPGAVE 4.4

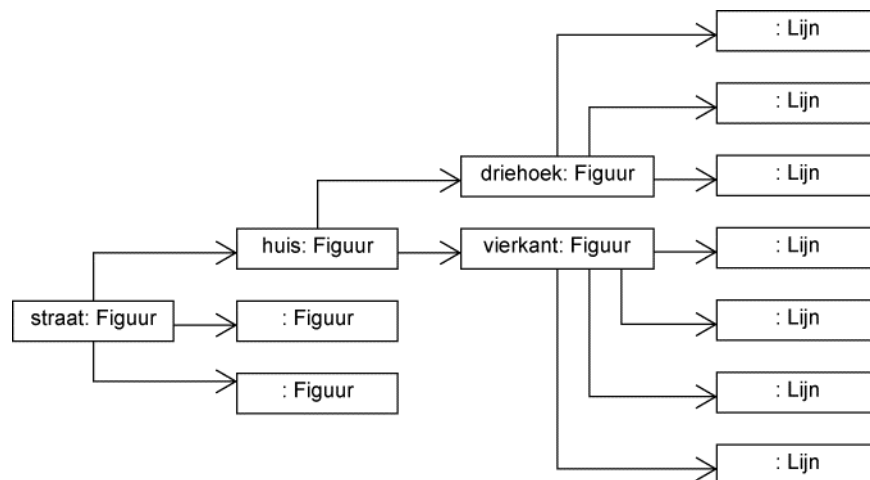
In veel tekenprogramma's kunnen figuren hiërarchisch worden opgebouwd. Uit lijnen kunnen bijvoorbeeld een driehoek en een vierkant worden opgebouwd, die twee kunnen worden samengevoegd tot een huisje en vervolgens kan een reeks huisjes een straat vormen: zie de tekening in figuur 4.1.

a Ga na dat het objectdiagram van figuur 4.2 de structuur van de tekening beschrijft. Dit objectdiagram is niet volledig; van slechts één deelfiguur (huis) zijn alle elementen getoond.

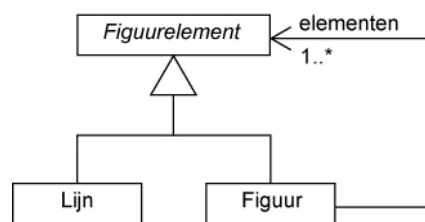
b Ga na dat dit objectdiagram overeenkomt met het klassendiagram van figuur 4.3.



FIGUUR 4.1 Tekening opgebouwd uit lijnen en deelfiguren

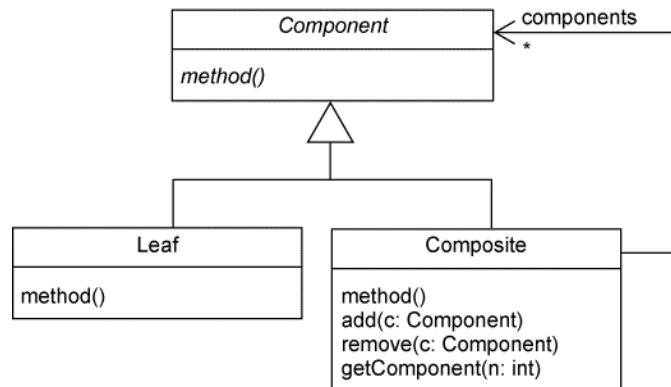


FIGUUR 4.2 (Onvolledig) objectdiagram bij figuur 4.1



FIGUUR 4.3 Klassendiagram voor de opbouw van figuren

De hiërarchie uit figuur 4.3 is ontworpen volgens het Composite-patroon. Figuur 4.4 toont de structuur van dit patroon.



FIGUUR 4.4 Composite-patroon

Met het Composite-patroon kunnen atomaire en samengestelde componenten op uniforme manier worden behandeld. De abstracte klasse *Component* representeert een component (atomair of samengesteld). De klasse heeft een of meer abstracte methoden – in de figuur aangeduid met ‘method’ – die in beide subclasses worden geherdefinieerd. Klasse *Composite* representeert een samengestelde component. Deze klasse heeft bovendien methoden voor het toevoegen, verwijderen en opvragen van componenten. Klasse *Component* definieert het standaardgedrag van alle componenten, klasse *Leaf* definieert het gedrag van de atomaire componenten en klasse *Composite* definieert het gedrag van samengestelde componenten.

OPGAVE 4.5

- Welke klasse uit figuur 4.3 komt overeen met *Component*, welke met *Composite* en welke met *Leaf*?
- Welke methoden zou u de klassen *Lijn* en *Figuur* in elk geval willen geven? Welke abstracte methode zou *Figuurelement* dan moeten bevatten?
- Schets hoe een aanroep van die methode op een *Figuur*-instantie verloopt.
- Stel u wilt een figuur van opgave 4.4 ook kunnen opbouwen met cirkels. Geef een gewijzigd klassendiagram voor de opbouw van een figuur.

Dit ontwerppatroon heeft zeer veel toepassingen. We noemen er twee afkomstig uit de Java API:

- In de *awt*-package komt de abstracte klasse *Component* overeen met de gelijknamige abstracte klasse in figuur 4.4, klasse *Container* komt overeen met klasse *Composite* en de klassen voor primitieve componenten (*Button*, *Label*, enzovoort) komen overeen met klasse *Leaf*. Containers zoals frames en panels kunnen daarom in een GUI willekeurig diep genest worden.
- De package *swing* bevat klassen voor de opbouw van een menubalk. Klasse *JMenuItem* komt overeen met de klasse *Component* in figuur 4.4, klasse *JMenu* komt overeen met klasse *Composite* en de klassen

JCheckBoxMenuItem en JRadioButtonMenuItem komen overeen met klasse Leaf. Menu's kunnen daardoor hiërarchisch worden opgebouwd, met submenu's die zelf weer submenu's kunnen hebben, enzovoort. Merk op dat klasse JMenuItem geen abstracte klasse is; ga dit na in de API. Een instantie van JMenuItem kan ook een primitief onderdeel van een menu zijn. Menu's komen nog aan de orde in leereenheid 11.

Deze cursus kan niet uitgebreid ingaan op dit belangrijke onderwerp maar zal volstaan met het hier en daar kort beschrijven van ontwerp Patronen. Er is tegenwoordig uitstekende literatuur te vinden in boekvorm en op internet.

2 Interfaces

Een abstracte klasse kan zowel abstracte als gewone, niet-abstracte methoden bevatten, waarvan de implementatie dus gegeven is. We kunnen een stap verder gaan door van een bepaald soort klasse te verlangen dat deze helemaal geen implementaties bevat. We komen dan bij het begrip interface. Een interface mag alleen abstracte methoden bevatten. Met het begrip interface bedoelen we hier nadrukkelijk niet de gebruikersinterface, maar het taalelement interface uit de taal Java.

Als van een methode geen implementatie wordt gegeven dan is alleen de signatuur bekend. De signatuur kan opgevat worden als een contract: deze vertelt wat de methode moet doen maar niet hoe dat moet gebeuren. Interfaces en abstracte klassen vertegenwoordigen contracten.

Voordat in deze paragraaf verder op het begrip interface wordt ingegaan, gaan we eerst in op hoe een contract afgedwongen kan worden door middel van een abstracte klasse. Vervolgens bespreken we het begrip interface waarmee ook een contract kan worden afgedwongen.

2.1 ABSTRACTE KLASSE ALS CONTRACT

In deze paragraaf gebruiken we de Vormenapplicatie uit leereenheid 3 als voorbeeld, maar met een ander ontwerp van de domeinlaag: in plaats van de drie methoden maakBol, maakBlok en maakCilinder heeft de klasse Vormenlijst nu een methode voegToe(Vorm) om een elders gecreëerde Vorm aan de lijst toe te voegen; zie figuur 4.5. Dit ontwerp is besproken in paragraaf 2.5 van leereenheid 3. We kiezen er hier voor om praktische redenen: met het oorspronkelijke ontwerp kunnen we het punt dat we hier willen maken niet illustreren.

Wat betekent een ontwerp als dat in figuur 4.5? Laten we ervan uitgaan dat er van de klassen in het diagram ook een specificatie beschikbaar is, die van alle methoden de signatuur geeft plus een uitleg wat de methode doet. We kunnen van twee kanten naar zo'n ontwerp kijken.

De ene kant is de kant van de *gebruiker*, niet de eindgebruiker, maar een programmeur die bij het implementeren van een andere klasse de klasse Vormenlijst wil gebruiken. Deze gebruiker kan zien welke methoden beschikbaar zijn. Bovendien is in de specificatie de signatuur van de methode vastgelegd, dus de gebruiker weet hoe hij de methode moet aanroepen: met hoeveel en met welke parameters.

De andere kant van het ontwerp is de kant van de *bouwer* van de klasse. Deze kan zien welke publieke methoden er geschreven moeten worden en wat die methoden moeten doen.

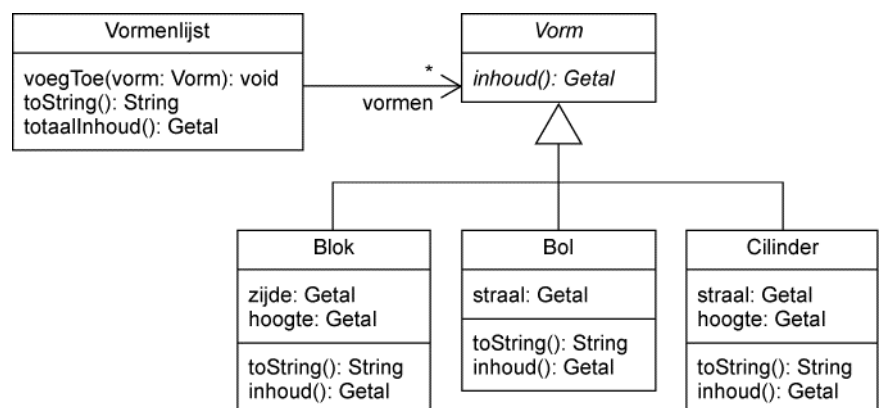
U hebt inmiddels met beide rollen kennisgemaakt: u hebt bestaande klassen uit de API gebruikt, maar u hebt ook zelf klassen geïmplementeerd. Het diagram en de specificaties samen vormen een afspraak tussen de twee betrokken partijen: de bouwer van een klasse en de gebruiker van die klasse.

De typeringsregels van Java spelen in beperkte mate een rol bij het controleren van die afspraak. Deze kunnen natuurlijk niet afdwingen dat in bijvoorbeeld de methode `voegToe` de opgegeven vorm inderdaad wordt toegevoegd. De taaldefinitie eist echter wel dat aan een aanroep van de methode `voegToe` een `Vorm` wordt meegegeven. De aanroep: `voegToe("dag allemaal")` voldoet niet aan de regels, want een `String` is geen `Vorm`.

Het aanroepen van methoden is één manier om gebruik te maken van bestaande klassen, overerving is een andere. In voorgaande leereenheden hebben we subklassen van allerlei klassen gemaakt. We voegden eigen methoden toe, specifiek voor ons doel, we herdefinieerden desgewenst methoden, maar we konden vooral van de bestaande implementatie van allerlei methoden gebruikmaken en onszelf daardoor veel werk besparen. Overerving is een handige manier om gebruik te kunnen maken van een bestaande implementatie.

Het hergebruiken van een bestaande implementatie is echter niet het enige waarvoor overerving gebruikt kan worden. Met overerving kan ook een afspraak, een contract, worden vastgelegd.

We bekijken de rol van overerving bij contracten aan de hand van de gewijzigde `VormenApplicatie`. Figuur 4.5 toont een gedeelte van het ontwerpmodel van die applicatie, met `Vorm` als abstracte klasse. We gaan nu uitgebreid onderzoeken wat het nut is van deze abstracte klasse `Vorm`.



FIGUUR 4.5 Domeinlaag van de gewijzigde `VORMenApplicatie`

Een `VORMenlijst` bevat een aantal `VORMen`; een `VORM` kan zijn: een `Blok`, een `Bol` of een `Cilinder`. Het probleem dat zich in leereenheid 3 voordeed toen klasse `VORM` niet abstract was, was dat de methode `inhoud` in de

klasse `Vorm` eigenlijk geen goede implementatie kon krijgen. De inhoud van een vorm kan niet worden berekend zonder te weten over welke vorm het gaat. Tegelijkertijd is het niet toegestaan om de methode inhoud helemaal weg te laten. Daar lijkt eigenlijk geen bezwaar tegen: van klasse `Vorm` zelf zal nooit een instantie worden gemaakt, alleen van de subklassen. Java eist echter dat de klasse `Vorm` wel een dergelijke methode heeft. In de `Vormenlijst` is een aantal `Vormen` opgeslagen, maar het is niet bij voorbaat bekend van welke subklasse. Methode `totaalInhoud` berekent het gezamenlijke inhoud van alle `Vormen`. Op de instanties van `Vorm` wordt daartoe de methode `inhoud` aangeroepen en dan moet het zeker zijn dat zo'n methode bestaat.

Het voornemen is om nooit een echte `Vorm` in de `Vormenlijst` te zetten, maar altijd een `Blok`, `Bol` of `Cilinder`. Dat voornemen is echter niet in de programmacode vastgelegd en de compiler kan het dus niet controleren. Een dergelijk voornemen is ook niet zoveel waard. Het voornemen kan – als we over een jaar nog eens wat aan de `VormenApplicatie` doen – al lang vergeten zijn. En ook een ander, die graag de klassen `Vorm` en `Vormenlijst` wil gebruiken en nieuwe `Vormen` (`Piramide`, `Kegel`) wil toevoegen, zal zich misschien niet aan dit voornemen houden.

Door de methode `inhoud` en de klasse `Vorm` abstract te maken, wordt het voornemen wél in de code vastgelegd. De compiler kan het voornemen nu controleren en zo het nakomen ervan afdwingen. Immers:

- Als de klasse `Vorm` een abstracte methode bevat, moet de klasse `Vorm` zelf ook abstract zijn.
- Een subklasse die de methode `inhoud` niet implementeert, moet ook abstract zijn.
- Een subklasse die niet abstract is, bevat dus een implementatie van de methode `inhoud`.
- Van een abstracte klasse kan geen instantie worden gemaakt.
- Als de methode `inhoud` wordt aangeroepen, bestaat er een object waar die aanroep heen wordt gezonden. Dat object is dan een instantie van een niet-abstracte klasse en bevat dus een implementatie van de methode `inhoud`.

Merk op dat de laatste redenering, waarbij de methode op een instantie wordt aangeroepen, niet opgaat voor klassenmethoden. Abstracte methoden mogen dan ook niet static zijn.

In dit geval is het erven van de abstracte methode `inhoud` niet nuttig uit een oogpunt van hergebruik van implementatie: er is helemaal geen implementatie. Het nut is dat er een afspraak wordt vastgelegd: op het moment van verwerken van een aanroep van de methode `inhoud` zal er een implementatie beschikbaar zijn. De compiler kan die afspraak ook controleren: als een niet abstracte klasse een abstracte methode erft, maar niet implementeert, zal de compiler protesteren.

De klasse `Vorm` bevat geen implementatie. Er zullen nooit objecten van de klasse `Vorm` worden gemaakt. Kan de klasse `Vorm` niet eenvoudig uit het ontwerp worden geschrappt, terwijl alle vormen toch in één lijst blijven staan?

Wat gebeurt er met de klasse `Vormenlijst` wanneer de klasse `Vorm` uit het ontwerp wordt geschrapt? De klasse `Vormenlijst` bewaart nu de vormen in een instantie van `ArrayList<Vorm>`. Dat zou een instantie van `ArrayList<Object>` moeten worden. De signatuur van methode `voegToe` zou dus als volgt aangepast kunnen worden:

```
public void voegToe(Object nieuweVorm)
```

Het gevolg van deze aanpassing zou zijn dat er in de vormenlijst niet alleen blokken, bollen en cilinders opgenomen kunnen worden, maar ook willekeurige andere objecten, als bijvoorbeeld `JButtons`. In de oorspronkelijke versie kunnen er in de vormenlijst geen verkeerde dingen worden gezet. We zouden die bescherming van de type-controle bij deze wijziging dus kwijt zijn.

Het echte probleem zit echter in de methode `totaalInhoud`:

```
for (Object vorm : vormen) {  
    inhoud = inhoud + vorm.inhoud();  
}
```

In de eerste regel binnen de `for`-each-opdracht moet een lijst van Objecten worden doorlopen in plaats van een lijst van `Vormen`. In de volgende regel wordt van elk element van de lijst de methode `inhoud` aangeroepen. De klasse `Object` heeft echter geen methode `inhoud`, dus dat is niet correct. Er zou een cast toegevoegd moeten worden, maar welke? Als we de klasse `Vorm` schrappen, kunnen we daar niet naar casten. Toch moet er een cast staan naar een klasse die een methode `inhoud` heeft. We kunnen naar `Bol` casten. Als het actuele type van `vorm` dan `Cilinder` blijkt te zijn, volgt een `ClassCastException`.

OPGAVE 4.6

Is het een goed idee om `Cilinder` en `Blok` subklassen van `Bol` te maken?

De klasse `Vorm` bevat van geen enkele methode een implementatie. Toch blijkt het een onmisbare klasse. De klasse `Vorm` representeert een afspraak, een contract. De klassen `Bol`, `Cilinder` en `Blok` erven niet van `Vorm` om gebruik te kunnen maken van aldaar aanwezige implementatie, want die is er niet.

Door `Bol`, `Cilinder` en `Blok` als subklassen van `Vorm` te definiëren, wordt vastgelegd dat deze klassen zullen voldoen aan het contract `Vorm`, dat wil zeggen dat zij implementaties van de methode `inhoud` zullen bevatten. De klasse `Vormenlijst` vertrouwt op dit contract. Zonder dit contract zou het niet mogelijk zijn om verschillende vormen naast elkaar in dezelfde vormenlijst op te nemen.

2.2 HET CONCEPT INTERFACE

Interface

Een *interface* kan worden opgevat als een zeer abstracte klasse: een klasse met uitsluitend constanten en abstracte methoden. Een interface is echter geen klasse, het is een apart concept naast het concept klasse.

We gaan eerst het begrip ‘interface’ introduceren. Zoals gebruikelijk bij de introductie van een nieuw taalconcept, moeten drie vragen worden beantwoord:

- 1 hoe wordt het geschreven (syntaxis)
- 2 wat betekent het (semantiek)
- 3 hoe wordt het gebruikt (pragmatiek).

Interfacedefinitie

Een *interfacedefinitie* kan onder meer de volgende vorm hebben:

Voorlopige syntaxis

```
[public] interface interfacenaam {
    [interfaceElementDefinities]
}
```

Een interface is impliciet abstract. Het is toegestaan om deze specificatie expliciet te vermelden; op dat punt is de hiervoor gegeven syntaxis onvolledig. Het wordt echter afgeraden.

We zullen in de cursus uitsluitend public interfaces gebruiken, hoewel de toegangsspecificatie ook package (dus zonder de aanduiding public) mag zijn. Dan kan de interface alleen in dezelfde package gebruikt worden.

De romp van een interface bevat nul of meer interfaceElementDefinities. Alle interface-elementen zijn impliciet public.

Een interfaceElementDefinitie kan zijn: een *constantendecclaratie* of een *methodespecificatie*. Binnen een interface kunnen dus alleen constanten worden gedeclareerd en methoden worden gespecificeerd.

De constanten in een interface hebben impliciet als toegangsspecificatie:

```
public static final
```

Het is toegestaan deze toegangsspecificaties expliciet op te nemen, maar het wordt – door de makers van de taal Java – afgeraden. Andere toegangsspecificaties voor constanten in een interface zijn niet toegestaan. We geven de volgende syntaxis, waarin de mogelijke toegangsspecificaties gemakshalve zijn weggelaten:

Constanten-declaratie

```
type variabelenaam = waarde;
```

Net als in klassen moet een constante altijd een waarde krijgen.

De methoden in een interface zijn impliciet altijd public en abstract. Het is wederom toegestaan om die specificaties expliciet op te nemen, maar het wordt afgeraden. Andere vormen van toegangsspecificatie zijn niet toegestaan. Wederom geven we de syntaxis waarin de expliciete toegangsspecificatie is weggelaten:

Methodespecificatie

```
terugkeertype methodenaam ([formele-parameterlijst]);
```

Voorbeeld

De abstracte klasse Vorm kan op de volgende wijze als interface worden geschreven:

```
public interface Vorm {
    double inhoud();
}
```

Bij het lezen van een interface moeten we in gedachten de impliciete specificaties invullen:

```
public abstract interface Vorm {
    public abstract double inhoud();
}
```

Dit heeft overigens geen gevolgen voor de klasse Vormenlijst. De methode voegToe kan gewoon de signatuur houden die hij had:

```
public void voegToe(Vorm vorm)
```

De volgende vraag is hoe een klasse (bijvoorbeeld Bol) van een interface (zoals Vorm) gebruik kan maken. Gebruikmaken is eigenlijk niet de juiste uitdrukking, we kunnen beter zeggen: de klasse Bol *verplicht zich tot* de interface Vorm. Dat wordt als volgt uitgedrukt:

```
public class Bol implements Vorm
```

De betekenis is dat de implementator van de klasse Bol toezegt de methode inhoud te implementeren. Het erven van een klasse geeft rechten: het recht om de methoden en attributen – voor zover niet private – in die klasse te gebruiken. Het erven van een abstracte klasse schept daarnaast nog verplichtingen, namelijk de verplichting om de abstracte methoden uit die klasse te implementeren.

Het *implementeren van een interface* geeft recht op toegang tot de constanten en schept de verplichting om alle methoden uit de interface te implementeren. Net als bij overerving verwerft de klasse Bol echter ook een recht, namelijk het recht om opgenomen te worden in een Vormenlijst. Algemener kan een Bol gebruikt worden op elke plaats waar een Vorm is toegezegd, bijvoorbeeld:

```
Vorm b = new Bol(7);
vormen.voegToe(b);
double inh = b.inhoud();
```

Bij de laatste opdracht is weer het actuele type van b bepalend voor de binding: de methode inhoud van Bol wordt verwerkt.

Implementeren van een interface

Bij een interface als gedeclareerd type past een implementatie daarvan als actueel type

Een interface kan dus net als een abstracte klasse als *typeaanduiding optreden in een declaratie*. Als *actueel type* voldoet iedere klasse die die *interface implementeert*. Binding van methoden is, zoals steeds, dynamisch.

Als de klasse Bol de methode inhoud niet implementeert, blijft die methode in de klasse Bol abstract. Dat kan, maar dan moet de klasse Bol als abstracte klasse worden gedeclareerd:

```
public abstract class Bol implements Vorm
```

Het voorgaande kan als volgt worden samengevat: als een abstracte klasse uitsluitend publieke constanten en abstracte methoden bevat (zoals de klasse Vorm), dan kan die klasse vervangen worden door een interface. Het nut van een interface is hetzelfde als het nut van een abstracte methode: het is geen implementatie, het representeert een afspraak, een contract.

OPDRACHT 4.7

Wijzig in Le04Vormen de abstracte klasse `Vorm` nu in een interface en breng ook de andere noodzakelijke wijzigingen aan. Ga na dat de applicatie nog steeds werkt.

OPGAVE 4.8

In een rijwiefabriek worden allerlei onderdelen gefabriceerd. Uit die onderdelen worden complete rijwielen gebouwd, die verkocht worden. Sommige onderdelen worden echter ook afzonderlijk verkocht. Een programma beheert alle producten: onderdelen en complete rijwielen. In dit programma worden verschillende producten als verschillende klassen gemodelleerd. Een product is verkoopbaar als het mogelijk is de verkoopprijs op te vragen.

- Geef de interface `Verkoopbaar` die de verkoopbaarheid van een product beschrijft.
- Spaken worden gemodelleerd met de klasse `Spaak`:

```
public class Spaak extends Product {
    // attributen
    private double lengte = 0.0; // in millimeter
    private double dikte = 0.0; // in millimeter
    private Materiaal materiaal = null;
    ...
}
```

Een `Spaak` is in deze implementatie niet verkoopbaar. Pas de klasse `Spaak` aan, zodat een `Spaak` wel verkoopbaar wordt.

Aanwijzing: als `I` een interface is en `K` een klasse, dan mogen de zinsneden ‘`extends K`’ en ‘`implements I`’ naast elkaar voorkomen.

- Bij het verwerken van bestellingen moet de totaalprijs voor een bestelling worden uitgerekend. De klasse `Bestelling` gebruikt daarbij de volgende hulpmethode, die de prijs uitrekent van het aantal bestelde exemplaren van één product.

```
/**
 * Berekent de prijs van een bestellingsregel.
 * @return de prijs van aantal exemplaren van product
 */
private double geefRegelPrijs(int aantal,
                               Verkoopbaar product)
```

Geef een implementatie van de methode `geefRegelPrijs`.

2.3 WAAROM INTERFACES?

Waarom is het concept interface nodig, als we precies hetzelfde kunnen bereiken met het al bestaande concept van abstracte klasse? In plaats van een interface te gebruiken kan een abstracte klasse gedeclareerd worden die uitsluitend constanten en abstracte methoden bevat. Aan de andere kant, als in plaats van een abstracte klasse een interface wordt gebruikt, dan hoeft niet meer bij elke methode te staan dat deze abstract is, want dat is hij vanzelf. Dat scheelt tikwerk. Toch is dat onvoldoende motivatie om naast abstracte klassen een geheel nieuw concept als interface in de taal Java te introduceren. Er moet een beter antwoord zijn.

Informatica en met name programmeren, is een voortdurende strijd om dingen die ingewikkeld dreigen te worden, eenvoudig te houden. Daarom moeten ook programmeertalen niet ingewikkelder worden gemaakt dan nodig is.

Enkelvoudige overerving

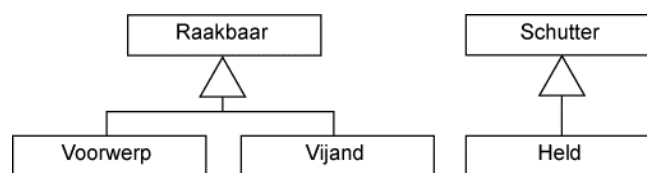
In paragraaf 2.2 van leereenheid 2 is vermeld dat er in Java sprake is van enkelvoudige overerving (single inheritance): in Java kan een klasse van slechts één directe superklasse erven. Dat de superklasse op haar beurt erft van een andere klasse doet er niet toe. Klasse JFrame van de package javax.swing bijvoorbeeld erft van klasse Frame van de package java.awt. Deze klasse erft op haar beurt van klasse Window en als we de klassenhiërarchie doorlopen komen we uiteindelijk bij klasse Object. Toch is hier sprake van enkelvoudige overerving want klasse JFrame heeft maar één directe superklasse. De andere subklassenrelaties zijn daar een gevolg van.

In opgave 4.7 heeft u gezien dat het soms wenselijk is dat een klasse eigenschappen meekrijgt van meer dan een superklasse: klasse Spaak is een Product en klasse Spaak is Verkoopbaar. Zou Spaak een subklasse kunnen zijn van zowel Product als Verkoopbaar? Nee, want meervoudige overerving is in Java niet toegestaan.

Meervoudige overerving

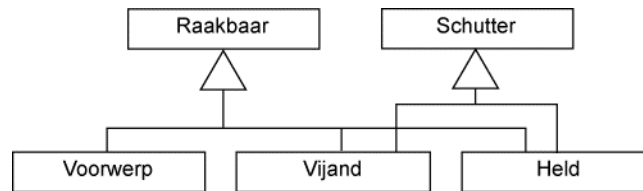
Meervoudige overerving (multiple inheritance) houdt in dat een klasse een (directe) subklasse is van meer dan een andere klasse.

Meervoudige overerving kan onder bepaalde omstandigheden heel nuttig zijn. Veronderstel dat we een klasse C willen maken die zowel de eigenschappen van klasse B als de eigenschappen van klasse A heeft. Dan zouden we graag C zowel van B als van A laten erven. We illustreren dit met het volgende voorbeeld. In een computerspelletje kan de held (van type Held) schieten op vijanden (van type Vijand) en op voorwerpen (van type Voorwerp). Op basis daarvan zou het ontwerp als in figuur 4.6 eruit kunnen zien.



FIGUUR 4.6 Klassendiagram van het spel, eerste versie

Veronderstel nu dat in een verdere ontwikkeling van het spel ook de vijanden op de held en op voorwerpen moeten kunnen schieten. Dat betekent dat de vijanden ook schutters zijn, en dat de held ook getroffen kan worden. De simpelste oplossing zou zijn: laat Held ook van Raakbaar en Vijand ook van Schutter erven. Dit wordt voorgesteld in figuur 4.7. Als klasse Raakbaar en klasse Schutter beide een methode bevatten met dezelfde signatuur, maar met een verschillende implementatie, welke implementatie moeten dan klasse Voorwerp, Vijand en Held overerven?



FIGUUR 4.7 Klassendiagram van het spel, tweede versie met meervoudige overerving

Java kent geen meervoudige overerving omdat dat problemen oplevert in combinatie met dynamische binding. Die problemen komen voort uit het erven van implementaties uit verschillende klassen. Bij interfaces spelen die echter niet: in interfaces komen immers helemaal geen implementaties voor.

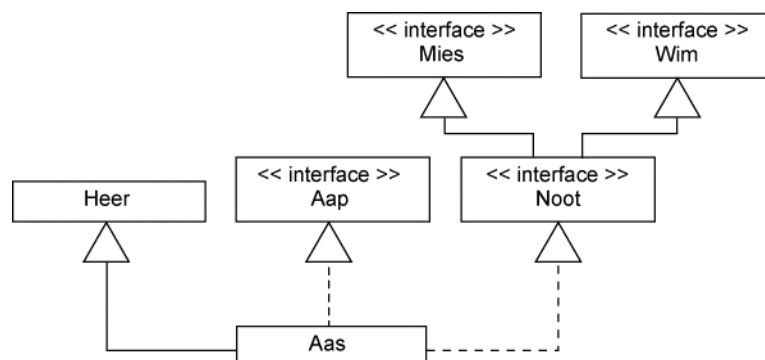
Er is dan ook geen enkel bezwaar tegen het implementeren van verschillende interfaces. Door naast het begrip klasse het begrip interface te introduceren, worden de mogelijkheden aanzienlijk verruimd.

Om te beginnen kan een klasse van een andere klasse erven en daarnaast een of meerdere interfaces implementeren. We hebben daarvan al een voorbeeld van gezien in de uitwerking van opgave 4.7:

```
public class Spaak extends Product implements Verkoopbaar
```

Voorts kan een klasse meer dan één interface tegelijk implementeren. In figuur 4.8 ziet u een klasse Aas die erft van de klasse Heer en de twee interfaces Aap en Noot implementeert:

```
public class Aas extends Heer implements Aap, Noot
```



FIGUUR 4.8 Klassendiagram met interfaces Aap, Noot, Mies en Wim

Dit komt ook in de API heel veel voor. De klasse ArrayList bijvoorbeeld erft van de (abstracte) klasse AbstractList en implementeert in totaal zes interfaces. De klasse JFrame erft van Frame en implementeert eveneens zes interfaces.

Tenslotte kan een interface van verscheidene andere interfaces erven. In figuur 4.8 erft de interface Noot van de twee interfaces Mies en Wim. De syntaxis is hier: `extends`, en niet: `implements`. Dat is logisch, want een interface implementeert niet. Deze relatie wordt daarom in het klassediagram met een volle pijl (extends-relatie) weergegeven terwijl de pijlen voor implementatie gestippeld zijn.

```
public interface Noot extends Mies, Wim
```

De betekenis is, dat het contract Noot bestaat uit de contracten Mies en Wim, plus de eventuele verplichtingen die in Noot zelf staan beschreven. Een klasse die Noot wil implementeren moet dus ook de gespecificeerde methoden in Mies en Wim implementeren. Dit soort stapeling van interfaces zullen we in deze cursus overigens niet ontwerpen.

Interfacedefinitie

De syntaxis van *interfacedefinities* wordt dus als volgt – waarbij een `interfaceElementDefinitie` weer een constantedeclaratie of een methodespecificatie is, als behandeld in paragraaf 2.2:

Syntaxis

```
[public] interface interfacenaam [extends interfacenamen] {
    [interfaceElementDefinities]
}
```

In een UML klassendiagram wordt een interface aangegeven door boven de naam van de interface de markering «interface» op te nemen.

3 Interfaces in de praktijk

In deze paragraaf bekijken we verschillende manieren waarop interfaces gebruikt kunnen worden.

3.1 MARKEERINTERFACES

Een interface bevat constanten en abstracte methoden. Interface `Cloneable` uit de package `java.lang` is echter leeg. Deze is als volgt gedefinieerd:

```
public interface Cloneable {
}
```

Markeerinterface

Een lege interface is een *markeerinterface*. Een markeerinterface heeft een speciale betekenis. Het is een soort vlaggetje bedoeld als een waarschuwing voor Java. Java verwacht van een klasse die een markeerinterface implementeert speciale eigenschappen.

Als een klasse de interface `Cloneable` implementeert, geeft de programmeur daarmee aan dat van instanties van deze klasse met behulp van de methode `clone` kopieën gemaakt mogen worden. De klasse erft de protected methode `clone` van `Object`, maar wordt ook geacht zelf een publieke herdefinitie van deze methode te bevatten. Omdat de interface de methode `clone` niet bevat, wordt dit laatste echter niet afgedwongen. In leereenheid 7 gaan we verder in op de interface `Cloneable`.

3.2 EIGENSCHAPPEN AFDWINGEN

In de VormenApplicatie hebben we al een voorbeeld gezien van het afdwingen van eigenschappen. De (gewijzigde) klasse Vormenlijst heeft een methode voegToe die instanties van type Vorm verwacht. Daardoor kan klasse Vormenlijst erop rekenen dat alle vormen die worden toegevoegd methode inhoud implementeren.

Ook de Java API dwingt de programmeur soms om klassen te gebruiken met bepaalde eigenschappen. Als voorbeeld noemen we de klasse Arrays van package java.util. Deze klasse bevat een aantal overloaded methoden sort voor het sorteren van de elementen van een array. Zo bevat de klasse een methode sort met de volgende signatuur:

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface.

In de specificatie van de methode wordt vermeld dat de elementen van de array interface Comparable moeten implementeren. Interface Comparable specificeert de methode compareTo; methode sort maakt gebruik van een implementatie van deze methode.

Zie ook
leereenheid 9

Third-party
implementatie

Als derde voorbeeld kijken we naar de package java.sql. Deze package verstrekt een API voor de toegang en de verwerking van data die opgeslagen zijn in een relationele database. Als we de package nader bekijken blijkt dat er een twintigtal interfaces wordt verstrekt, maar voor de meeste daarvan geen implementatie. Waarom is dat? De implementatie wordt overgelaten aan zogenaamd *third-parties*, externe bedrijven. In dit geval gaat het om bedrijven die gespecialiseerd zijn in DBMS-en (database management systemen). Zij kunnen een set klassen ontwikkelen die de verstrekte interfaces implementeren op een wijze die aansluiting biedt op hun eigen producten. Door gebruik te maken van de interfaces in java.sql kunnen programmeurs hun Java-programma schrijven zonder kennis te hebben van de details van DBMS-specifieke implementaties, eventueel zelfs voordat hun bedrijf een keus heeft gemaakt voor een bepaald DBMS.

Door het verstrekken van een set interfaces wordt dus een standaard neergezet waaraan third-parties zich moeten houden.

3.3 KIEZEN TUSSEN IMPLEMENTATIES

Zoals uit de voorgaande paragraaf blijkt, biedt de package java.sql programmeurs – of liever gezegd hun bedrijf – de keus uit verschillende implementaties van de package. Een programma dat gebruik maakt van een database kan, door gebruik te maken van de interfaces, geschreven worden voor een op dat moment nog onbekend DBMS. Als de keus gemaakt is, kan met weinig inspanning het programma geschikt worden gemaakt voor de specifieke implementatie van de package.

Ook uit de Java API kunnen we een voorbeeld geven waar de programmeur de keus heeft tussen verschillende implementaties. Voor een lijst van waarden heeft u tot nu toe meestal een `ArrayList` gebruikt. Naast de `ArrayList` bevat de API een andere implementatie van een list, klasse `LinkedList`. Beide klassen implementeren de interface `List`. Door hun verschil in implementatie – een `ArrayList` maakt intern gebruik van een array van elementen en een `LinkedList` maakt gebruik van schakels die met elkaar verbonden zijn – zijn er grote prestatieverschillen. Een `ArrayList` is bijvoorbeeld sneller bij het toevoegen van een element aan het einde van de lijst en bij het zoeken naar een bepaald element. Een `LinkedList` is sneller bij het toevoegen van elementen aan het begin of in het midden van de lijst en bij het verwijderen van elementen.

OPDRACHT 4.9

Project Le04Lijst bevat in de package `lijstdemo` de klasse `LijstDemo` die het verschil in prestatie tussen een `ArrayList` en een `LinkedList` laat zien. De klasse heeft een constante `N` voor het aantal elementen van de lijst en als lijst van getallen een attribuut van type `List<Integer>`. In de constructor wordt de lijst, die meegegeven is als parameter, gevuld met random-getallen. De getallen worden telkens aan het begin van de lijst gezet. Verder heeft klasse `LijstDemo` een methode `zoek` waarin met behulp van een methode uit de API alle elementen van de lijst een keer worden gezocht. De tijd voor het doorlopen van de constructor en van methode `zoek` worden op de standaarduitvoer getoond. Om de test uit te voeren gebruikt methode `main` eerst een lijst van type `ArrayList<Integer>` en daarna van type `LinkedList<Integer>`. Verwerk de klasse en bekijk het resultaat.

Door in het programma overal gebruik te maken van het type `List` – de interface – is één opdracht voldoende, namelijk de creatie van een instantie van type `List`, om naar behoefte te switchen van implementatie voor de lijst.

SAMENVATTING

Paragraaf 1

Superklassen die zelf niet geïnstantieerd hoeven te worden, kunnen gedefinieerd worden als abstracte klassen. Abstracte klassen kunnen abstracte methoden bevatten. Van een dergelijke methode wordt binnen de abstracte klasse alleen de signatuur opgenomen, de methode moet in iedere niet-abstracte subklasse worden geherdefinieerd. Een ontwerppatroon beschrijft een vaak optredend probleem bij het ontwerpen van OO-programma's en geeft een schematische oplossing voor dat probleem, in termen van een relatief klein aantal klassen en relaties tussen deze klassen. Het Composite-patroon is een voorbeeld van een ontwerppatroon. Elementen van een complexe structuur kunnen hiermee op hiërarchische wijze worden geordend.

Paragraaf 2

Overerving is niet alleen nuttig om bestaande code te hergebruiken, maar kan ook een manier zijn om een afspraak vast te leggen. Een interface is te beschouwen als een contract. Methoden in een interface zijn per definitie `public` en `abstract`. Een interface mag ook constanten bevatten.

Deze hebben per definitie de toegangsspecificaties: public, static en final. Een klasse die een interface implementeert, verplicht zich alle daarin genoemde methoden te implementeren. De klasse verwerft daarmee het recht overal te verschijnen waar deze interface wordt verwacht. Een interface kan dus gebruikt worden als gedeclareerd type van een variabele of parameter. Het actuele type is dan een klasse die de interface implementeert. Bij methodeaanroepen vindt dynamische binding plaats. Een klasse die een interface implementeert maar niet alle methoden daaruit, bevat abstracte methoden en moet dus zelf ook als abstract worden gedeclareerd.

Een klasse mag maar van één andere klasse (direct) erven. Onafhankelijk daarvan mag een klasse nul of meer interfaces implementeren. Interfaces kunnen ook van andere interfaces erven.

Paragraaf 3

Een markeerinterface is een lege interface bedoeld als een waarschuwing voor Java. Cloneable is zo een interface. Op instanties van een klasse die de interface implementeert kan methode clone worden aangeroepen. Een interface kan gebruikt worden om eigenschappen af te dwingen. Een klasse die de interface implementeert moet alle abstracte methoden van de interface implementeren.

Een interface kan als type fungeren in een programma. Een programma kan daarmee generiek worden geschreven. Hierdoor is het heel makkelijk om verschillende implementaties van een bepaalde klasse te gebruiken.

ZELFTOETS

- 1 Welke taalelementen kunnen in een abstracte klasse en in een interface voorkomen?
- 2 Noem een aantal verschillen en overeenkomsten tussen abstracte klassen en interfaces.
- 3 De harde schijf van een computer bevat mappen en bestanden. De inhoud van een map bestaat uit submappen en bestanden. Modelleer dit domeinmodel in een klassendiagram.
Aanwijzingen
– Maak gebruik van het Composite-patroon.
– Neem in het diagram geen attributen en methoden op.
- 4 Voor het beoordelen van een uitwerking kan een docent verschillende schalen hanteren. Op een schaal van twee is de score een voldoende of een onvoldoende. Op een schaal van tien kan een score 0 t/m 10 zijn en op een schaal van honderd kan een score 0 t/m 100 zijn. In alle gevallen kan van de score bepaald worden of deze voldoende of onvoldoende is.
a Project Le04Beoordeling bevat in de package beoordeling de drie klassen Beoordeling2, Beoordeling10 en Beoordeling100 die beoordelingsscores representeren op de schaal van respectievelijk twee, tien en honderd.
Op instanties van de drie klassen moet methode isVoldoende kunnen worden aangeroepen om te bepalen of de score wel of niet voldoende is. Schrijf voor dit doel een abstracte klasse Beoordeling en pas ook de drie Beoordeling-klassen aan.

b Bij schalen van tien en van honderd kunnen ook verschillende scores opgeteld worden om een gemiddelde te kunnen berekenen. De waarde van de scores moet dan wel eerst gestandaardiseerd worden; als standaard kiezen we de schaal 0 .. 10.

Op instanties van de klassen `Beoordeling10` en `Beoordeling100` moet een methode `standaardWaarde` kunnen worden aangeroepen die de gestandaardiseerde score geeft (een getal van 0 t/m 10). Schrijf voor dit doel een interface `Optelbaar` en pas de twee klassen aan.

c Klasse `BeoordelingTest` uit de package `test` dient voor het testen van de domeinklassen. In deze klasse worden twee cijferlijsten gemaakt en worden de methoden `isVoldoende` en `isStandaardWaarde` op de beoordelingen aangeroepen. Test het programma met klasse `BeoordelingTest`.

TERUGKOPPELING

1 Uitwerking van de opgaven

- 4.1 a Het is niet verplicht een subklasse van `Vorm` een methode `inhoud` te geven en dus bevat de klasse `Blok` geen (taal)fouten. In de rest van de code wordt de methode `inhoud` altijd aangeroepen op een object met gedeclareerd type `Vorm` en omdat `Vorm` een methode `inhoud` heeft, is dit in orde.
- b Op de betreffende instantie van `Blok` wordt de methode `inhoud` aangeroepen. Omdat die klasse geen methode `inhoud` heeft, wordt de methode van de superklasse gebruikt. Deze levert `-1` op, een onjuiste waarde.
- Een foutje in de naam van een methode (bijvoorbeeld `imhoud` in plaats van `inhoud`) kan dus tot het onbedoelde gebruik van een methode uit de superklasse leiden.

- 4.2 a Klasse `Vorm` wordt hiermee

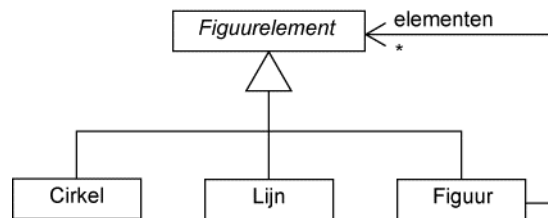
```
public abstract class Vorm {
    public abstract double inhoud();
}
```

De applicatie werkt verder hetzelfde.

- b In het editor-venster verschijnt bij de kop van de klasse `Blok` een foutmelding die aangeeft dat klasse `Blok` de abstracte methode `inhoud` van `Vorm` moet implementeren. Er wordt geen bezwaar gemaakt tegen de aanwezigheid van methode `imhoud`, wel tegen de afwezigheid van methode `inhoud`.
- 4.3 a Ja, het is zinvol om klasse `Rekening` abstract te maken: het is immers de bedoeling dat de klasse `Rekening` zelf nooit geïnstantieerd wordt. Het is ook zinvol de methode `neemOp` abstract te maken. Elke subklasse van `Rekening` heeft een eigen implementatie van die methode.
- b Nee, dat is niet zinvol. In verband met de lokaliteit willen we dat iedere rekening – dus iedere instantie van een subklasse van `Rekening` – methoden `eindeMaand` en `eindeJaar` heeft. Voor sommige soorten rekeningen hoeft er aan het einde van de maand of van het jaar niets te gebeuren. Het is handig wanneer die de lege implementatie van beide methoden uit `Rekening` erven.
- 4.4 a De straat bestaat uit drie huizen. Elk huis is opgebouwd uit een driehoek voor het dak en een vierkant voor de woning. Een driehoek is opgebouwd uit drie lijnen en een vierkant is opgebouwd uit vier lijnen.
- b De klasse `Figuurelement` heeft twee subklassen, namelijk `Lijn` en `Figuur`. Elk figuurelement is dus of een lijn of een andere (deel)figuur. Klasse `Lijn` heeft geen associaties. Een instantie van `Lijn` heeft dus geen links naar andere objecten. Klasse `Figuur` heeft een associatie met `Figuurelement`; een instantie van `Figuur` heeft dus altijd minstens een link met een `Figuurelement` (In figuur 4.2 zijn niet al deze links getekend, maar ze zijn er wel, ook vanuit de onderste twee anonieme instanties van `Figuur`).

Het klassendiagram geeft dus precies aanleiding tot een hiërarchisch objectdiagram als dat uit figuur 4.2: links een figuur, rechts lijnen en daartussen deelfiguren.

- 4.5
- a De klasse *Figuurelement* komt overeen met *Component*, de klasse *Lijn* komt overeen met *Leaf* en de klasse *Figuur* met *Composite*.
 - b Om de figuren te kunnen tekenen ligt het voor de hand om *Lijn* en *Figuur* een methode *teken* te geven. Deze methode speelt dan de rol van de methode die in figuur 4.4 is aangeduid als 'method'. De abstracte klasse *Figuurelement* krijgt daarom een abstracte methode *teken*.
 - c De methode *teken* van *Lijn* tekent de lijn. De methode *teken* van *Figuur* tekent zelf niets maar roept op alle elementen hun methode *teken* aan. Dat mag omdat *Figuurelement* een methode heeft met die naam. De tekenopdracht wordt door een figuur dus doorgegeven aan de deelfiguren; alleen de lijnen onderin de hiërarchie worden uiteindelijk echt getekend.
 - d Zie figuur 4.9.



FIGUUR 4.9 Cirkels zijn ook figuurelementen

- 4.6 We kunnen het programma – zonder de klasse *Vorm* – inderdaad weer aan de praat krijgen door *Cilinder* en *Blok* subklassen van *Bol* te maken. Daar zijn echter principiële bezwaren tegen. Als we de klasse *K1* laten erven van de klasse *K*, dan bedoelen we meestal dat *K1* een bijzonder soort *K* is. Het is in dit geval echter moeilijk staande te houden dat een *Blok* of een *Cilinder* een bijzonder soort *Bol* is. *Cilinder* en *Bol* opvatten als subklassen van *Blok* stuit op hetzelfde bezwaar. Er is onder deze drie begrippen niet één algemeen begrip – en twee bijzondere – maar het zijn alle drie bijzondere gevallen van het algemene begrip *Vorm*.
- 4.7 De volledige implementatie van *Vorm* luidt:

```

package vormen;

public interface Vorm {
    /**
     * Berekent na herdefinitie de inhoud van de
     * een specifiek soort vorm.
     * @returns de inhoud van het vorm
     */
    double inhoud();
}

```

Daarnaast moet in de kop van de klassen *Blok*, *Bol* en *Cilinder* 'extends' vervangen worden door 'implements':

```

public class Blok implements Vorm {
public class Bol implements Vorm {
public class Cilinder implements Vorm {

```

- 4.8 a De interface Verkoopbaar ziet er als volgt uit:

```
public interface Verkoopbaar {
    double getVerkoopprijs();
}
```

- b Om instanties van de klasse Spaak verkoopbaar te maken, moeten we de volgende aanpassing maken:

```
public class Spaak extends Product
    implements Verkoopbaar {
    // attributen
    private double lengte = 0.0; // in millimeter
    private double dikte = 0.0; // in millimeter
    private Materiaal materiaal = null;
    private double prijs = 0.0;
    ...
    public double getVerkoopprijs() {
        return prijs;
    }
    ...
}
```

- c Implementatie van de methode regelPrijs:

```
private double geefRegelPrijs(int aantal,
                               Verkoopbaar product) {
    double stuksprijs = product.getVerkoopprijs();
    return aantal * stuksprijs;
}
```

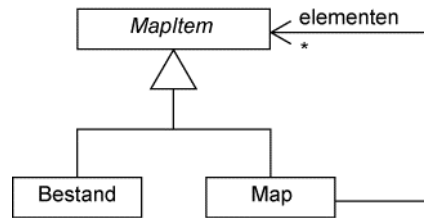
Merk op, dat Spaak voldoet als actueel type voor de parameter product, omdat Spaak Verkoopbaar implementeert.

- 4.9 U ziet dat de creatie van een ArrayList langer duurt dan die van een LinkedList, maar dat het zoeken in een LinkedList veel trager is.

1 Uitwerking van de zelftoets

- 1 Een abstracte klasse kan constantendeclaraties, attribuutdeclaraties, implementaties van methoden en specificaties van abstracte methoden bevatten.
Een interface kan constantendeclaraties en specificaties van abstracte methoden bevatten.
- 2 Abstracte klassen en interfaces mogen beide niet geïnstantieerd worden. Ze worden gebruikt om een contract vast te leggen. Ze kunnen wel beide als gedeclareerd type voorkomen in een programma.
In tegenstelling tot interfaces kan een abstracte klasse attribuutdeclaraties en implementaties van methoden bevatten. Een klasse mag slechts van één abstracte klasse erven maar verschillende interfaces implementeren.

- 3 Behalve de klassen Map en Bestand bevat het diagram nog een abstracte klasse MapItem die de inhoud van een map modelleert; zie figuur 4.10.



FIGUUR 4.10 klassendiagram met Composite-patroon

Merk op dat de multiplicititeit van de associatie hier * is en niet 1..*, zoals bij de klasse Figuur (zie figuur 4.3). Een Figuur bestaat altijd uit minstens een Figurelement, maar een map kan leeg zijn en dus geen MapItems hebben.

- 4 a Klasse Beoordeling ziet er als volgt uit:

```

package beoordeling;

/**
 * Representeert een beoordeling
 * @author Open Universiteit Nederland
 */
public abstract class Beoordeling {

    /**
     * Bepaalt of de score voldoende is.
     * @return true als voldoende, false anders
     */
    public abstract boolean isVoldoende();
}
  
```

Bij de klassen Beoordeling2, Beoordeling10 en Beoordeling100 moet de kop van de klasse gewijzigd worden zodat geërfd wordt van klasse Beoordeling en moet methode isVoldoende geïmplementeerd worden. We geven hier de code van klasse Beoordeling2 met de aanpassingen in grijs gearceerd:

```

package beoordeling;

/**
 * Beoordeling op een schaal van twee.
 * Een beoordeling kan voldoende of onvoldoende zijn.
 * Bij creatie is de score altijd voldoende.
 * @author Open Universiteit Nederland
 */
public class Beoordeling2 extends Beoordeling{

    public enum Waarde {V, O}

    private Waarde waarde = Waarde.V; // score
  
```

```

/**
 * Bepaalt of de score voldoende is.
 * @return true als voldoende, false anders
 */
public boolean isVoldoende(){
    return waarde == Waarde.V;
}

/**
 * Maakt de score voldoende.
 */
public void setVoldoende() {
    waarde = Waarde.V;
}

/**
 * Maakt de score onvoldoende.
 */
public void setOnvoldoende() {
    waarde = Waarde.O;
}

/**
 * String-representatie van de score.
 */
public String toString() {
    if(waarde == Waarde.V) {
        return "voldoende";
    }
    else {
        return "onvoldoende";
    }
}
}

```

De wijzigingen van de andere twee klassen zijn vergelijkbaar.

b Interface Optelbaar ziet er als volgt uit:

```

package beoordeling;

/**
 * Representeert een optelbare waarde.
 * Een waarde is optelbaar als deze gestandaardiseerd
 * kan worden op een schaal van tien, van 0 t/m 10.
 * @author Open Universiteit Nederland
 */
public interface Optelbaar {
    /**
     * Geeft de tegenwaarde op een schaal van tien,
     * van 0 t/m 10.
     * @return de waarde
     */
    int standaardWaarde();
}

```

Bij de klassen `Beoordeling10` en `Beoordeling100` moet de kop van de klasse gewijzigd worden om interface `Optelbaar` te implementeren en moet methode `standaardWaarde` geïmplementeerd worden. We tonen de delen van de klasse `Beoordeling100` die zijn gewijzigd:

```
public class Beoordeling100 extends Beoordeling
                                implements Optelbaar{
    ...

    /**
     * Geeft de standaardwaarde op een schaal van tien,
     * dus een waarde van 0 t/m 10.
     * @return de hoogte (0 t/m 10) van de score
     */
    public int standaardWaarde(){
        return waarde / 10;
    }

    ...
}
```

c Geen uitwerking.

Inleveropdracht

De inleveropdracht vindt u op de cursussite in yOulearn onder de tab Bronnen. Onder de tab Cursus/Cursusstructuur ziet u wanneer u uw uitwerking uiterlijk kunt inleveren. Het maken van deze inleveropdracht is verplicht.

Generieke typen en enumeratietypen

Introductie 161

Leerkern 162

- 1 Generieke klassen en methoden 162
 - 1.1 Een niet generieke klasse 162
 - 1.2 Generieke klassen 163
 - 1.3 Naamgeving 166
 - 1.4 Ruwe typen en subtypen 166
 - 1.5 Begrensde typeparameters 168
- 2 Toepassing: een generieke stack 170
- 3 Enumeratietypen 172

Samenvatting 176

Zelftoets 177

Terugkoppeling 178

- 1 Uitwerking van de opgaven 178
- 2 Uitwerking van de zelftoets 185

Generieke typen en enumeratietypen

INTRODUCTIE

Voor het bijhouden van een lijst kent u de klasse `ArrayList<E>`. Een lijst strings is van type `ArrayList<String>`, een lijst gehele getallen van type `ArrayList<Integer>` en een lijst objecten van type `ArrayList<Object>`. Bij de klasse `ArrayList<E>` is het dus mogelijk om aan te geven wat het type moet zijn van de elementen in de lijst, op voorwaarde dat het type een referentietype is. Dit wordt aangegeven door tussen de vishaakjes het gewenste type op te geven.

Normaal gesproken heeft een klasse in Java geen toevoeging in de vorm van een type tussen vishaken. De klasse `ArrayList<E>` is daarom een bijzonder soort klasse, een generieke klasse ofwel een generiek type. De API bevat nog meer generieke klassen, met name in het zogeheten Collections Framework, dat bestaat uit een stel klassen voor het representeren van allerlei soorten verzamelingen. De meeste verzamelingen bestaan immers uit elementen van hetzelfde type; het is handig om dat type (net als bij `ArrayList`) expliciet aan te kunnen geven.

In paragraaf 1 en 2 van deze leereenheid bekijken we generieke klassen. Dat helpt u bij het gebruik van de generieke klassen uit de API. Ook leert u (beperkt) om zelf zulke klassen te ontwerpen.

In de eerste cursus heeft u ook geleerd om voor een groep constante waarden een enumeratietype te definiëren, gebruikmakend van het sleutelwoord `enum`. Nu u over voldoende gereedschap beschikt om de theorie hierover beter te begrijpen, zetten we in paragraaf 3 van deze leereenheid de puntjes op de i.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u

- kunt uitleggen wat een generieke klasse is en kunt aangeven waarvoor generieke klassen worden gebruikt
- een generieke klasse kunt gebruiken
- zelf een eenvoudige generieke klasse kunt ontwerpen en implementeren
- kunt aangeven wanneer wel en niet sprake is van een subtype bij generieke typen
- een grens aan een typeparameter kunt geven
- een enumeratietype als aparte klasse kunt definiëren
- de belangrijkste methoden kunt noemen die enumeratietypen tot hun beschikking hebben
- de betekenis kunt geven van de volgende kernbegrippen: formele typeparameter, actuele typeparameter, geparametriseerd type, ruw type, bovengrens.

Studeeraanwijzingen

Het is niet mogelijk om binnen deze cursus alle aspecten te behandelen die te maken hebben met generieke klassen (dat heeft zowel te maken met de studielast als met de complexiteit van het onderwerp). Wij hebben er daarom voor gekozen om alleen basistheorie te behandelen en problemen te signaleren die verband houden met wat wel en wat niet mag bij het gebruik van generieke klassen.

De studielast van deze leereenheid bedraagt circa 8 uur.

LEERKERN

1 Generieke klassen en methoden

Generiek

Vanaf JDK 1.5 zijn generieke klassen en generieke methoden geïntroduceerd in de taal Java. De term *generiek* verwijst hier naar het feit dat de definitie van een klasse of van een methode zo kan worden opgesteld dat deze bruikbaar is voor verschillende typen. Bij het gebruik van de klasse moet dan het gewenste type worden opgegeven.

Klasse `ArrayList` is een voorbeeld van een generieke klasse. Met behulp van de volgende twee opdrachten kunt u bijvoorbeeld een lijst creëren voor elementen van type `String` en een andere voor elementen van type `Integer`:

```
ArrayList<String> lijst1 = new ArrayList<>();
ArrayList<Integer> lijst2 = new ArrayList<>();
```

We beginnen met het nut van generieke klassen te illustreren aan de hand van een voorbeeld dat ze nu juist niet gebruikt.

1.1 EEN NIET GENERIEKE KLASSE

De volgende (gewone, niet generieke) klasse `Lade` representeert een lade waarin één object van type `Object` bewaard kan worden:

```
public class Lade {
    private Object object = null;

    public void bewaar(Object object) {
        this.object = object;
    }

    public Object getObject() {
        return object;
    }
}
```

Een instantie van type `Lade` bevat bij creatie geen object. Met methode `bewaar` kan een object in de lade worden geplaatst en met methode `getObject` kan bekeken worden welk object de lade bevat.

Stel u wilt de klasse `Lade` gebruiken voor het bewaren van een getal van type `Integer`. U schrijft de volgende code:

```

1 Lade lade = new Lade();
2 lade.bewaar(10);
3 Integer i = (Integer)lade.getObject();
4 System.out.println(i);

```

In regel 3 wordt een cast naar Integer gebruikt voor de toekenning aan de Integer i; methode getObject heeft immers een terugkeerwaarde van type Object. Omdat lade een Integer bevat, is de cast correct. Het programmafragment kan probleemloos verwerkt worden.

OPGAVE 5.1

Stel dat regel 2 van het programmafragment nu als volgt wordt veranderd:

```
lade.bewaar("10");
```

Kan het programmafragment worden gecompileerd en verwerkt?

Het programmafragment is geschreven om te gebruiken met objecten van type Integer. In opgave 5.1 werd daartegen gezondigd, met een verwerkingsfout als gevolg. De programmeur had natuurlijk commentaar kunnen gebruiken om zijn voornemen kenbaar te maken alleen Integers in de lade te bewaren, maar hiermee wordt dat voornemen niet afgedwongen. Bij een simpel voorbeeld als dit zal de vergissing niet snel worden gemaakt. In een echt programma met vele regels code en vele methoden en waar meerdere programmeurs aan werken, zal dit veel sneller het geval zijn.

Het gebruik van een cast impliceert een belofte. Overtreding van deze belofte wordt pas tijdens verwerking geconstateerd. Een betere typecontrole tijdens compilatie kan een verwerkingsfout voorkomen. Hier bieden generieke klassen en methoden een oplossing voor.

1.2 GENERIEKE KLASSEN

Het probleem van de klasse Lade was dat een instantie van die klasse elk subtype van Object kon bevatten. We willen nu een klasse Doos definiëren waarvan elke instantie alleen een object van een gegeven type kan bevatten, zonder dat type bij de definitie van de klasse al vast te leggen. Pas bij het gebruik van de klasse in een declaratie wordt het type opgegeven. Klasse Doos wordt een generieke klasse waarin het type van het object dat bewaard wordt nog niet vastligt.

De definitie van de generieke klasse Doos is als volgt:

```

public class Doos<T> {
    private T object;

    public void bewaar(T object) {
        this.object = object;
    }

    public T getObject() {
        return object;
    }

    public String toString() {
        return "Doos: " + object;
    }
}

```

Formele
typeparameter
Generieke klasse

Vergelijken we deze definitie met die van klasse Lade, dan zien we dat het type Object overal is vervangen door type T. De naam van de klasse is Doos<T> om aan te geven dat een instantie van Doos een object van type T zal bevatten.

T is een *formele typeparameter*, dat wil zeggen een formele parameter die staat voor een willekeurig type. Een *generieke klasse* – of generiek type – is een klasse die een formele typeparameter heeft. De formele typeparameter wordt gedeclareerd door een naam tussen vishaakjes te plaatsen achter de naam van de klasse. Merk op dat bij een declaratie van een gewone variabele of parameter het type altijd samen met de naam moet worden opgegeven. Bij een typeparameter kan dat niet. Daarom staat alleen de naam van de typeparameter tussen de vishaakjes.

De syntaxis van de kop van een klasse wordt (voorlopig) als volgt uitgebreid:

Syntaxis

```
[toegang] class klassennaam [<formele-typeparameterlijst>]
                                [extends klassennaam]
                                [implements interfacenamen]
```

Tussen de vishaakjes <> kan meer dan één typeparameter worden gedeclareerd. De namen van de typeparameters moeten verschillend zijn. De namen worden gescheiden door komma's.

In de generieke klasse Doos wordt de typeparameter T gedeclareerd in de kop van de klasse. Alle andere voorkomens van T binnen de romp van de klasse verwijzen naar dezelfde typeparameter.

Bij het gebruik van de klasse Doos moet worden aangegeven voor welk type de klasse gebruikt wordt. De volgende declaratie geeft aan dat het object in de doos van type Integer zal zijn:

```
Doos<Integer> integerDoos;
```

De variabele integerDoos wordt als volgt geïnstantieerd (u kent dit al van ArrayList):

```
integerDoos = new Doos<>();
```

De declaratie en de creatie kunnen uiteraard gecombineerd worden tot één opdracht:

```
Doos<Integer> integerDoos = new Doos<>();
```

Elk voorkomen van T in de romp van de klasse Doos<Integer> staat dan voor Integer. Het type van het attribuut object is Integer, de formele parameter object van methode bewaar heeft als type Integer en het terugkeertype van methode getObject is ook Integer.

Een klasse zoals Doos<Integer> die een instantie is van het generieke type Doos<T> definieert een *geparametriseerd type*.

Een type zoals Integer dat gebruikt wordt om een geparametriseerd type te creëren heet een *actuele typeparameter*. Een actuele typeparameter moet altijd een referentietype zijn. Een primitief type als int kan dus niet als actuele typeparameter fungeren.

Geparametriseerd
type
Actuele
typeparameter

Met het volgende codefragment wordt een doos gebruikt voor het bewaren van een getal van type Integer:

```
1 Doos<Integer> integerDoos = new Doos<>();
2 integerDoos.bewaar(10);
3 Integer i = integerDoos.getObject();
4 System.out.println(i);
```

Merk op dat in regel 3 geen cast meer nodig is. Methode getObject heeft een terugkeerwaarde van type T, dus in het geval van Doos<Integer> van de actuele typeparameter Integer.

OPDRACHT 5.2

a Open het project Le05Doos. Klasse DoosDemo bevat in methode main het gegeven codefragment. Voeg na de eerste aanroep naar bewaar de volgende regel toe:

```
integerDoos.bewaar("10");
```

Wat gebeurt er?

b Herstel de code en voeg aan het einde van methode main de volgende code toe:

```
Doos<String> stringDoos = new Doos<>();
stringDoos.bewaar("10");
stringDoos = integerDoos;
```

Welke fout signaleert de compiler?

c Vervang in de code van opdracht b de actuele typeparameter String door Object. Wat gebeurt er en wat maakt u daaruit op?

In opdracht 5.2.a is getracht een object van type String te plaatsen in een instantie van type Doos<Integer>. Deze fout werd, in tegenstelling tot de fout van opgave 5.1, tijdens compilatie ontdekt. Het gebruik van een generieke klasse is een goede manier om typecontrole tijdens compilatie af te dwingen.

Belangrijk

Zoals blijkt uit opdracht 5.2, staat Java geen casting toe tussen twee geparametriseerde typen die verschillende actuele typeparameters hebben, ook al is een cast tussen die typeparameters zelf wel mogelijk. Als T een directe of indirecte subklasse is van S, dan is Doos<T> toch geen subklasse van Doos<S>.

OPGAVE 5.3

Geef een codefragment dat een instantie van ArrayList creëert voor elementen van type Doos<String> en dat vervolgens een element aan die lijst toevoegt.

Elk referentietype kan als actuele typeparameter gebruikt worden; primitieve typen zijn niet toegestaan.
Een generiek type kan klassenmethoden bevatten. Een formele typeparameter mag daarin echter *niet* gebruikt worden. De volgende code is dus niet toegestaan:

Fout!

```

public class Klasse<E> {
    private static E elem = null; // mag niet!

    public static E getElem() { // mag niet!
        return elem;
    }

    public static void setElem(E elem) { // mag niet!
        this.elem = elem;
    }
}

```

1.3 NAAMGEVING

Afspraak

Voor het gebruik van typeparameters geldt de volgende naamconventie: De naam van een typeparameter is een enkele hoofdletter. Vaak worden daarvoor T en E gebruikt, als afkorting van Type en Element. Voor een tweede, derde en vierde typeparameter worden meestal S, U en V gebruikt.

Er is voor gekozen om een enkele letter te gebruiken om het verschil met de naam van een klasse en van een interface te benadrukken.

OPDRACHT 5.4

a Start een nieuw project Le05Paar met een package paren en definieer daarbinnen een generieke klasse Paar die twee elementen kan bevatten van verschillende typen. Geef de klasse in plaats van een methode bewaar een constructor die de elementen een waarde geeft. Geef de klasse ook een get-methode voor elk van de elementen en een methode toString.

Aanwijzing: in de constructor moet de naam van de klasse zonder typeparameters worden gebruikt. De typeparameters worden immers al in de kop van de klasse gedeclareerd.

b Schrijf ook een eenvoudig testprogramma en test de klasse (u hoeft maar één paar te creëren).

1.4 RUWE TYPEN EN SUBTYPEN

De klassen `ArrayList<E>` van de package `java.util`, `Doos<T>` en `Paar<T, S>` zijn drie voorbeelden van generieke klassen. Nemen we als actuele typeparameter bijvoorbeeld `String` dan kunnen we instanties declareren van type `ArrayList<String>`, `Doos<String>` en `Paar<String, String>`. Een instantie creëren doen we door de constructor aan te roepen en daarbij eveneens de actuele typeparameter te specificeren:

```

ArrayList<String> lijst = new ArrayList<>();
Doos<String> doos = new Doos<>();
Paar<String, String> paar = new Paar<>("aap", "noot");

```

In plaats van `String` kan ook klasse `Object` – of elk ander referentietype – als actuele typeparameter worden gebruikt:

```

Doos<Object> objDoos = new Doos<>();

```

Ruwe type

In `objDoos` kunnen objecten van type `Object` bewaard worden en dus ook instanties van elke subklasse van `Object`. De aanroepen `objDoos.bewaar("aap")` en `objDoos.bewaar(10)` zijn beide correct (de tweede gebruikt autoboxing). De aanroep van methode `getObject` van klasse `Doos` op instantie `objDoos` zal echter altijd een terugkeerwaarde hebben van (gedeclareerd) type `Object` (ongeacht het actuele type).

Het is ook mogelijk om een instantie te declareren en te creëren van type `Doos`, zonder daarbij een actuele typeparameter te specificeren. In dat geval zal de instantie van het *ruwe type* (raw type) `Doos` zijn. Een instantie is van het ruwe type als het generieke type wordt gebruikt zonder een vermelding van de actuele typeparameters. In dat geval geldt dat het type `Object` wordt gebruikt als actuele typeparameter. Met de opdracht

```
Doos ruweDoos = new Doos();
```

Waarschuwing

wordt een instantie gemaakt van het ruwe type `Doos` waarin een object van type `Object` bewaard kan worden. De compiler zal wel een *waarschuwing* (warning, unchecked warning) geven dat `Doos` een generiek type is dat bij voorkeur geparametriseerd moet worden. Een waarschuwing is geen foutmelding; het programma kan gecompileerd en verwerkt worden. De programmeur wordt alleen geattendeerd op een bijzondere situatie die aanleiding tot een fout kan zijn.

Ruwe typen zijn nodig om de werking te garanderen van programma's die geschreven zijn in eerdere versies van Java, dus zonder generieke typen. We geven als voorbeeld het Java Collections Framework, een verzameling klassen waarmee datastructuren in Java gedefinieerd en bewerkt kunnen worden. Klasse `ArrayList` is een klasse uit dit framework. Het framework bestond al in versies van Java ouder dan JDK 1.5, dus voordat generieke klassen bestonden. Programma's van voor JDK 1.5 maakten dus instanties aan van `ArrayList` en van de andere klassen uit het framework zonder een actuele typeparameter te specificeren; die mogelijkheid bestond immers niet. Om deze programma's nu nog te kunnen gebruiken zijn de ruwe typen in het leven geroepen.

Niet doen!

Gebruik zelf nooit ruwe typen in uw programma, ook als `Object` de actuele typeparameter zou zijn. Dit leidt alleen maar tot waarschuwingen en mogelijk tot verwerkingsfouten. Het gebruik van verkeerde typen binnen generieke klassen leidt tot een compilatiefout. Het gebruik van ruwe typen leidt meestal tot een waarschuwing.

In de package `java.util` is klasse `ArrayList<E>` gedefinieerd als een subklasse van de abstracte klasse `AbstractList<E>`. Er bestaat daarom een ordeningsrelatie tussen de twee klassen zolang de actuele typeparameters identiek zijn. `ArrayList<String>` is een subklasse van `AbstractList<String>`, `ArrayList<Integer>` is een subklasse van `AbstractList<Integer>` en `ArrayList<Object>` is een subklasse van `AbstractList<Object>`. Maar tussen `ArrayList<String>`, `ArrayList<Integer>` en `ArrayList<Object>` bestaat *geen* ordeningsrelatie.

Subtypering bij generieke typen is in Java ontworpen volgens het substitutieprincipe van Liskov. Het substitutieprincipe van Liskov zegt dat het mogelijk moet zijn om in een klassenhierarchie objecten van een subklasse te behandelen als waren het objecten van een superklasse.

We illustreren dit met een voorbeeld om inzichtelijk te maken waarom `ArrayList<String>` geen subtype is van `ArrayList<Object>`. Een banaan is een vrucht. Wie een banaan krijgt heeft een vrucht ontvangen. Banaan is dus een subtype van Vrucht. In een vruchtendoos kunnen alle soorten vruchten worden opgeborgen. We kunnen ons voorstellen dat de doos een deksel met uitsparingen heeft, waar elk type vrucht doorheen kan. In een bananendoos zijn de uitsparingen langwerpiger, zodat er wel bananen in passen maar geen andere soorten vruchten. We kunnen een bananendoos dus niet behandelen als een vruchtendoos. Bananendoos is dus geen subtype van Vruchtendoos.

OPGAVE 5.5

Welke van de opdrachten uit het volgende programmafragment zijn correct, welke leiden tot een waarschuwing en welke tot een compilatiefout?

```
1 ArrayList<Integer> lijst1 = new ArrayList<>();
2 ArrayList lijst2 = new ArrayList<>();
3 AbstractList<Integer> lijst3 = lijst1;
4 AbstractList<Integer> lijst4 = lijst2;
5 AbstractList<String> lijst5 = lijst3;
6 AbstractList lijst6 = lijst2;
```

1.5 BEGRENSENDE TYPEPARAMETERS

Het is soms wenselijk om voorwaarden te koppelen aan een formele typeparameter. Stel bijvoorbeeld dat we een doos willen maken waarin alleen instanties bewaard kunnen worden van klasse `Vorm` zoals gedefinieerd in paragraaf 2 van leereenheid 3. Als gegarandeerd is dat in de doos altijd een waarde van type `Vorm` (of van een subklasse van `Vorm`) zit, dan is binnen Doos de methode inhoud uit de klasse `Vorm` beschikbaar om op instanties van type `T` aan te roepen. We moeten dan kunnen afdwingen dat de formele typeparameter `T` van `Doos<T>` als waarde `Vorm` of een subklasse van `Vorm` krijgt.

Dat kan door als voorwaarde aan een formele typeparameter een *bovengrens* (upper bound) op te geven. Een bovengrens is een van de volgende twee:

- de naam van een klasse waarvan de actuele typeparameter een subklasse moet zijn (de actuele typeparameter mag ook gelijk zijn aan de bovengrens)
- de naam van een interface die de actuele typeparameter moet implementeren.

Een formele typeparameter met een bovengrens is een *begrensde typeparameter*.

Om een bovengrens aan een formele typeparameter te stellen wordt bij de specificatie van de typeparameter tussen vishaakjes de naam van de parameter opgegeven, gevolgd door het sleutelwoord `extends`, gevolgd door de bovengrens. Bijvoorbeeld:

```
public class MijnKlasse<T extends AndereKlasse>
```

Bovengrens

*Begrensde
typeparameter*

Ook als de bovengrens een interface is wordt het sleutelwoord `extends` gebruikt:

```
public class MijnKlasse<T extends MijnInterface>
```

Bij meerdere bovengrenzen worden de bovengrenzen gescheiden door het karakter `&`:

```
<T extends AndereKlasse & MijnInterface>
```

De betekenis van het sleutelwoord `extends` heeft in het geval van een bovengrens een ruimere betekenis dan u gewend bent. Als er staat `<T extends AndereKlasse>` dan kan de actuele parameter de klasse `AndereKlasse` zelf zijn of een subklasse van `AndereKlasse`. We zeggen in de rest van deze leereenheid dan dat de actuele parameter een *extensie* van `AndereKlasse` is.

Extensie

Kop van een klasse

De volledige syntaxis van de *kop van een klasse* luidt dus:

Syntaxis

```
[toegang] [final] [abstract] class klassennaam
    [< formele-typeparameterlijst >]
    [extends klassennaam]
    [implements interfacenamen]
```

Formele-
typeparameter

Een formele-typeparameterlijst bestaat uit een of meer *formele-type-parameters*, gescheiden door komma's.

De syntaxis van een *formele-typeparameter* is:

```
parameternaam [extends klassenlijst]
```

waarbij een klassenlijst bestaat uit één klassennaam en/of één of meer interfaces, gescheiden door `&`, oftewel:

```
TypeParameter extends Class & Interface1 & ... & InterfaceN
```

Voorbeeld

Als voorbeeld tonen we de klasse `VormenDoos` waarvan de formele typeparameter de abstracte klasse `Vorm` als bovengrens heeft. In de klasse is een methode `inhoudObject` opgenomen die de inhoud van het object teruggeeft. Deze methode is toegestaan omdat het type `T` gegarandeerd een implementatie van methode `inhoud` bevat.

```
import vormen.Vorm;

public class VormenDoos<T extends Vorm> {
    private T object;

    public void bewaar(T object) {
        this.object = object;
    }

    public T getObject() {
        return object;
    }

    public double inhoudObject() {
        return object.inhoud();
    }
}
```

Bij het gebruik van deze klasse moet de actuele typeparameter een extensie zijn van `Vorm`, dus `Blok`, `Bol`, `Cilinder` of `Vorm` zelf.

Mogelijke geparametriseerde typen zijn dus `VormenDoos<Blok>`, `VormenDoos<Bol>`, `VormenDoos<Cilinder>` of `VormenDoos<Vorm>`.

Wat is nu eigenlijk het nut hiervan? Kunnen we niet net zo goed een niet-generieke versie `VormenDoosNG` definiëren met een attribuut van type `Vorm`? De methode `bewaar` krijgt dan ook een `Vorm` als parameter en de methode `getObject` geeft een `Vorm` terug.

Er is wel degelijk een verschil tussen `VormenDoos<T>` en `VormenDoosNG`. Bij de klasse `VormenDoos<T>` kan worden afgedwongen dat een instantie alleen cilinders kan bevatten:

```
VormenDoos<Cilinder> cilinderDoos = new VormenDoos<>();
```

Het is nu niet toegestaan om in `cilinderDoos` een blok te stoppen. Bij `VormendoosNG` is dat niet mogelijk.

OPDRACHT 5.6

In de package `java.lang` is de interface `Comparable<T>` gedefinieerd. Klassen die deze interface implementeren maken het mogelijk om hun instanties te vergelijken met een instantie van type `T`. Daartoe dient de methode `compareTo` te worden geïmplementeerd. Deze methode is als volgt gespecificeerd:

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

a Maak een kopie van de klasse `Doos<T>` uit het project `Le05Doos` en wijzig die zo, dat de typeparameter `T` de interface `Comparable<T>` als bovengrens krijgt. Noem de gewijzigde klasse `VergelijkbareDoos`. Geef de klasse een methode `vergelijkMet` die de inhoud van de doos vergelijkt met de inhoud van een andere doos.

Aanwijzing

Een methode kan een generiek type als parameter hebben.

b Geef de klasse een methode `max` die een instantie van `VergelijkbareDoos` vergelijkt met een andere instantie en die als terugkeerwaarde de grootste van de twee doos geeft.

Aanwijzingen

- Een methode kan een generiek type als terugkeerwaarde hebben.
- De grootste doos is de doos met de grootste inhoud.

2 Toepassing: een generieke stack

Generieke klassen en methoden zijn geïntroduceerd in Java vanwege de behoefte aan generieke datastructuren in het Java Collections Framework. Programmeurs willen vaak een homogene verzameling specificeren, dat wil zeggen een verzameling voor een bepaald type element. Het van oorsprong niet generieke framework bood die mogelijk niet; de elementen waren altijd van type `Object` (zoals nu bij het ruwe type).

Voorbeeld

De klasse `ArrayList<E>` maakt deel uit van het Collections Framework. In deze en de vorige cursus is dankbaar gebruik gemaakt van het generieke karakter van deze klasse. In de VormenApplicatie bijvoorbeeld was de lijst met vormen van type `ArrayList<Vorm>`. Zonder de typeparameter, die vooraf al garandeert dat alle elementen in de lijst van type `Vorm` zijn, zou de code van de klasse `Vormenlijst` veel meer casts bevatten.

Toepassingen waarin generieke klassen en methoden gedefinieerd worden zullen meestal gevonden worden in de sfeer van verzamelingen en datastructuren, juist vanwege die wens van homogeniteit in het type van de elementen.

In deze paragraaf gaan we als voorbeeld zelf een generieke stack (stapel) implementeren. Een stack is te vergelijken met een stapel borden: het laatste bord dat op de stapel is gelegd wordt er ook het eerst weer van af gepakt. Een stack is een datastructuur die in zijn meest elementaire vorm alleen twee operaties beschikbaar stelt. Er kan een element op de stack worden geplaatst (dit wordt een push-operatie genoemd) en het element boven op de stack kan verwijderd worden (dit wordt een pop-operatie genoemd).

Stacks worden veel gebruikt voor het bewaren van een historie. Denk bijvoorbeeld aan methodeaanroepen waarbij het terugkeeradres bewaard moet worden of tekstverwerking waarbij correcties met een 'undo' ongedaan gemaakt moeten kunnen worden. Andere toepassingen van stacks zijn het berekenen van een expressie of het controleren of in een expressie haakjesparen correct zijn genest.

OPDRACHT 5.7

In deze opdracht gaat u een generieke klasse `Stack<E>` definiëren. De klasse krijgt methoden `push` en `pop` om een element op de stack te plaatsen respectievelijk het bovenste element te verwijderen. De klasse krijgt ook een methode `toString` voor het maken van een stringrepresentatie van de stack en een methode `isEmpty` om te controleren of de stack leeg is.

- Waarvoor staat de formele typeparameter `E` in dit geval?
- Welke attributen krijgt de klasse? Toon hun declaratie.
- Geef specificaties van alle methoden.
- Maak een nieuw project `Le05Stack` met een package `stack` en implementeer daarbinnen de volledige klasse `Stack<E>`.
- Schrijf een (eenvoudige) testklasse en test de klasse daarmee.

We gaan nu een stack gebruiken om te controleren of in een expressie de haakjesparen correct zijn genest. Een expressie kan ronde- en rechte haakjesparen bevatten, dus de haakjes `() []`.

Voorbeelden

Een voorbeeld van een (in dit opzicht) correcte expressie is

$$[x] + ((5*3) - y[x+2, 5])$$

Een voorbeeld van een incorrecte expressie is

$$[x + (5*3)] - y(x+2, 5)$$

In deze laatste expressie staat een rond sluithaakje te veel en bovendien klopt de nesting niet: na de 3 wordt het vierkante haakje al gesloten terwijl het ronde haakje nog open is.

Bij het controleren van de nesting van een dergelijke expressie komt een stack goed van pas. We lopen karakter voor karakter door de expressie. Als we een openingshaakje tegen komen, zetten we het op de stack. Komen we een sluithaakje tegen, dan halen we het bovenste element van de stack af. Dit moet het corresponderende openingshaakje zijn: rond voor een rond sluithaakje, recht voor een recht sluithaakje. Lig er niets op de stack of ligt er bovenop een ander openingshaakje, dan is de expressie niet correct. Als alle karakters bekeken zijn, moet de stack leeg zijn (anders zijn er openingshaakjes die niet worden gesloten). Merk op dat de expressie ook allerlei andere karakters bevat. Daar doen we niets mee.

OPDRACHT 5.8

- Wat wordt in dit geval het type van de elementen op de stack?
- Voeg aan Le05Stack een klasse Expressie toe en geef deze een methode haakjesGoed die controleert of de haakjes in de expressie goed staan.
- Test de klasse Expressie.

3 Enumeratietypen

Een enumeratietype (enumerated type, enum) definieert een vast aantal constante waarden.

Gegeven is de volgende klasse Speler:

```
public class Speler {
    private enum Kleur {WIT, ZWART}

    private Kleur kleur;

    public Speler(Kleur kleur) {
        this.kleur = kleur;
    }

    public void printKleur() {
        if (kleur == Kleur.WIT) {
            System.out.println("Deze speler is wit");
        }
        else {
            System.out.println("Deze speler is zwart");
        }
    }
}
```

Kleur is een enumeratietype met de twee constante waarden WIT en ZWART, die zelf ook van type Kleur zijn. Binnen de klasse Speler kunnen de waarden Kleur.WIT en Kleur.ZWART worden gebruikt. De naam is in dit geval gekwalificeerd met de naam van het enumeratietype. Omdat bij de definitie van het enumeratietype de toegangsspecificatie public is gebruikt, kunnen de waarden ook buiten de klasse Speler worden gebruikt. In dat geval moet de naam tevens gekwalificeerd worden met de naam van de klasse waarin het enumeratietype is gedefinieerd: Speler.Kleur.WIT en Speler.Kleur.ZWART.

Een enumeratietype wordt gedefinieerd met behulp van het sleutelwoord `enum`. Een enumeratietype hoeft niet binnen een klasse te staan maar kan ook apart gedefinieerd worden.

Voorbeeld

Wordt het enumeratietype `Kleur` buiten de klasse `Speler` gedefinieerd, dan krijgen we de volgende twee definities:

```
public enum Kleur {
    WIT,
    ZWART
}

public class Speler {

    private Kleur kleur;

    public Speler(Kleur kleur) {
        this.kleur = kleur;
    }

    public void printKleur() {
        if (kleur == Kleur.WIT) {
            System.out.println("Deze speler is wit");
        }
        else {
            System.out.println("Deze speler is zwart");
        }
    }
}
```

Elke klasse die gebruik maakt van de constante waarden van de klasse `Kleur` moet de naam van de constanten kwalificeren met de naam van het enumeratietype, dus in het geval van `Kleur`: `Kleur.WIT` en `Kleur.ZWART`. Dat geldt voor de klasse `Speler` maar ook voor alle andere klassen die gebruik maken van `Kleur`. In de cursus Objectgeoriënteerd programmeren zagen we dat enumeratietypen, indien ze in een andere package gebruikt worden, geïmporteerd moeten worden. Nu begrijpt u vermoedelijk ook beter waarom dat is: het zijn gewoon klassen. De definitie van een enumeratietype is in feite niets anders dan de definitie van een speciaal soort klasse, met een opsomming van constanten (static en final) die allemaal van hetzelfde type zijn als de klasse zelf.

Als een enumeratietype binnen een andere klasse wordt gedefinieerd, dan heeft de definitie bij voorkeur de toegang `private` of `protected`. De constante waarden worden dan uitsluitend binnen de omhullende klasse gebruikt. Wil men buiten de klasse toegang hebben tot het enumeratietype, dan kan het enumeratietype beter als aparte klasse met toegang `public` worden gedefinieerd of kan de toegang beperkt worden tot de package waarin de klasse zich bevindt.

De volgorde van declaratie van de constanten bepaalt hun onderlinge rangorde. De eerste constante krijgt index 0, de tweede index 1, enzovoort.

Naamconventie

Zoals elke klassennaam begint de naam van een enumeratietype bij afspraak met een hoofdletter en worden de constanten met uitsluitend hoofdletters geschreven.

Methoden `toString`,
`compareTo` en
`ordinal`

Elk enumeratietype is een subklasse van de klasse `Enum` uit de package `java.lang`. Daardoor erft het enumeratietype alle methoden van klasse `Enum`, waaronder de methoden `toString` voor een stringrepresentatie, `compareTo` om constanten met elkaar te vergelijken en `ordinal` waarmee de index van een constante verkregen kan worden; raadpleeg hiervoor desgewenst de API Specification. Verder zijn de operatoren `==` en `!=` beschikbaar om twee waarden van een enumeratietype met elkaar te vergelijken.

Daarnaast krijgt elk enumeratietype automatisch de beschikking over de methode `values` met de volgende signatuur:

Methode `values`

```
static naamEnumeratietype[] values()
```

Deze methode levert een array op van alle constanten die binnen het enumeratietype zijn gedeclareerd, in volgorde van declaratie. De aanroep van bijvoorbeeld `Kleur.values()` levert de array `{Kleur.WIT, Kleur.ZWART}` op van type `Kleur[]`. De methode `values()` ontbreekt in de API van klasse `enum`. Deze statische methode wordt voor elk enum-type door de compiler gegenereerd.

OPDRACHT 5.9

- Bekijk de definitie van de genoemde methoden van `Enum` in de API Specification.
- Start een nieuw project `Le05Dag` en definieer in een aparte klasse het enumeratietype `Dag` dat als constanten de dagen van de week heeft.
- Schrijf een testklasse waarmee de dagen van de week naar de standaarduitvoer worden geschreven. Gebruik hiervoor methode `values`.

Omdat een enumeratietype een klasse is, kunnen er attributen, methoden en constructoren aan worden toegevoegd. De opsomming van constanten, die altijd als eerste in de klassendefinitie moet worden opgenomen, moet dan eindigen met een puntkomma achter de laatste constante. Daarna kunnen dan andere elementen volgen.

OPDRACHT 5.10

Voeg aan de klasse `Dag` een klassenmethode `isWeekend` toe waarmee getest kan worden of een gegeven dag in het weekend valt.

In opdracht 5.10 definieerde u `isWeekend` als klassenmethode. Eigenlijk ligt dat niet voor de hand. De enumklasse `Dag` heeft zeven constanten, die zelf ook allemaal van type `Dag` zijn. Het ligt dan in de lijn van het OO-paradigma om de methode `isWeekend` niet aan te roepen met een `Dag` als parameter, maar op een waarde van `Dag` zelf, dus

```
Dag.MAANDAG.isWeekend();  
// in plaats van Dag.isWeekend(Dag.MAANDAG)
```

Dat betekent dat we de methode `isWeekend` moeten definiëren als objectmethode. Dat kan ook. De code ziet er dan als volgt uit:

```
public boolean isWeekend() {  
    return this == ZATERDAG || this == ZONDAG;  
}
```

Bij de aanroep die we zojuist toonden heeft this de waarde MAANDAG en is de terugkeerwaarde dus false. Merk op dat de kwalificatie van de constanten met Dag nu overbodig is: de code staat immer binnen de definitie van Dag.

OPDRACHT 5.11

Wijzig de klasse Dag dusdanig dat isWeekend een objectmethode wordt. Pas ook de testklasse aan.

Door toevoeging van methoden aan een enumeratietype wordt het mogelijk om gedrag en eigenschappen aan de constanten te koppelen. In opdracht 5.11 heeft u daarvan een voorbeeld gezien: door de methode isWeekend is het mogelijk om te bepalen of een dag in het weekend valt. Attributen worden gebruikt om de afzonderlijke constanten met extra informatie te verbinden.

Voorbeeld

Het enumeratietype Land definieert constanten voor vijf landen. Elk land is gekoppeld met de domeinaanduiding die in webadressen voor dat land wordt gebruikt, bijvoorbeeld '.nl' voor Nederland. De code van de klasse Land ziet er als volgt uit:

```
public enum Land {
    NEDERLAND("nl"),
    BELGIE("be"),
    LUXEMBURG("lu"),
    DUITSLAND("de"),
    FRANKRIJK("fr");

    private String code = null;

    private Land (String code) {
        this.code = code;
    }

    public String getCode() {
        return code;
    }
}
```

Bij elke constante is de waarde van het attribuut code opgenomen als parameter achter de naam van de constante. Zouden er meerdere attributen zijn, dan worden de parameters gescheiden door komma's. De declaratie van het attribuut code en de definitie van de constructor volgen op de opsomming van de constanten. Merk op dat de toegangsspecificatie van de constructor private is. De constructor wordt alleen door de Java Virtual Machine gebruikt en mag nooit expliciet worden aangeroepen. De JVM creëert met een aanroep van de constructor voor elke constante een instantie van de klasse Land, waarbij het bijbehorende attribuut wordt geïnitieerd. De methode getCode, die hier ter illustratie is opgenomen, kan op een instantie van type Land worden aangeroepen om de bijbehorende code te krijgen.

NB: Het gegeven voorbeeld beperkt zich hier tot vijf landen. Dat is veel minder dan alle landen waarvoor een landencode bestaat. Dit enumeratietype Land kan alleen gebruikt worden in een toepassing waarvoor

geen uitbreiding van het aantal landen is voorzien. Een enumeratietype is namelijk bedoeld voor een verzameling waarden die niet aan verandering onderhevig is en waar de extra informatie die opgeslagen is in attributen ook niet verandert. Gebruik een gewone klasse zodra de waardeverzameling niet vastligt.

OPDRACHT 5.12

Breid de klasse `Dag` uit opdracht 5.11 zo uit dat bij elke dag ook de Engelse naam wordt bijgehouden. Neem in de klasse ook een methode op voor het opvragen van deze vertaling.

Klasse `EnumSet`

Enumeratietypen zijn bedoeld om een verzameling constanten te definiëren en om deze verzameling of een deel daarvan te kunnen doorlopen. In dit verband noemen we de klasse `EnumSet` en de daarin gedefinieerde methode `range`. Klasse `EnumSet` representeert een deelverzameling van de verzameling constanten van een enumeratietype. Deze methode krijgt als parameters twee waarden van een enumeratietype en levert de deelverzameling van dat type op die begint met de eerste en eindigt met de tweede waarde.

Voorbeeld

Het volgende codefragment drukt de namen van alle werkdagen af.

```
System.out.println("Werkdagen:");
for (Dag dag: EnumSet.range(Dag.MAANDAG, Dag.VRIJDAG)) {
    System.out.println(dag);
}
```

Klasse `EnumSet` bevat nog meer handige methoden. Raadpleeg desgewenst de API Specification.

In deze paragraaf is voor de enumeratietypen telkens een aparte klasse gedefinieerd. Wordt een enumeratietype binnen een andere klasse gedefinieerd, dan gelden dezelfde mogelijkheden. Ook dan kunnen attributen, methoden en constructoren worden toegevoegd.

SAMENVATTING

Paragraaf 1

Een klasse kan generiek worden opgesteld voor allerlei typen. Een bekend voorbeeld is de klasse `ArrayList<E>`. Tussen de vishaakjes wordt de formele typeparameter gedeclareerd. Een typeparameter is altijd een referentietype. Door het gebruik van een typeparameter kan typecontrole tijdens compilatie plaatsvinden. Typecasts zijn dan niet nodig. Een instantie van een generiek type is een geparametriseerd type. Voor de formele typeparameter wordt een waarde opgegeven. Dit is de actuele typeparameter. Als een generiek type wordt gebruikt zonder een actuele typeparameter is er sprake van een ruw type. De compiler gaat er dan vanuit dat type `Object` de actuele typeparameter is. Een typeparameter kan begrensd zijn met een klasse en/of een interface. De actuele typeparameter moet dan de grens zelf zijn of een subklasse zijn van de grens en/of de opgegeven interface implementeren.

- Paragraaf 2 In deze paragraaf is een generieke klasse gedefinieerd die een stack representeert. Met een push-operatie kan een element op de stack worden geplaatst en met een pop-operatie kan het bovenste element van de stack worden gehaald. We hebben de generieke stack gebruikt om te controleren of de haakjesparen van een gegeven expressie correct zijn geplaatst.
- Paragraaf 3 Een enumeratietype is een opsomming van constante waarden. Een enumeratietype kan binnen een andere klasse of als aparte klasse worden gedefinieerd. De klasse kan attributen, methoden en private constructoren hebben. De attributen definiëren eigenschappen van de constanten. Ze worden geïnitieerd door de constructor. De constructor van de klasse heeft altijd de toegang private. Hij mag alleen door Java zelf worden aangeroepen. Elk enumeratietype krijgt automatisch de beschikking over de methode values die een array van de constanten oplevert.

ZELFTOETS

- 1
 - a In opdracht 5.4 hebben we een klasse Paar gedefinieerd die twee elementen van verschillende typen kan bevatten. Definieer nu een klasse Duo die twee elementen van hetzelfde type kan bevatten. Geef de klasse een constructor die de twee elementen een waarde geeft. Geef de klasse ook een get-methode voor elk van de elementen en een methode equals waarmee een instantie vergeleken kan worden met een andere instantie.
 - b Geef de code voor de declaratie en initialisatie van de variabele duoLijst die een ArrayList is waarvan de elementen duo's van strings zijn.
- 2 De abstracte klasse Beeld representeert allerlei beeldmateriaal. Subklassen daarvan zijn de klassen Grafiek voor getekend materiaal als diagrammen, Foto voor alle soorten foto's en Kaart voor stadsplattegronden en landkaarten. Deze drie klassen hebben zelf ook subklassen. Schrijf een generieke klasse voor het beheren van beeldmateriaal. De lijst van beeldmateriaal wordt opgeslagen in een ArrayList. Geef de klasse een methode voor het toevoegen van een beeld aan de lijst.
- 3 Kinderen kunnen ingedeeld worden in leeftijdsfasen naar gelang hun leeftijd. Van 0 tot 2 jaar is een kind een baby, van 2 tot 4 jaar een peuter, van 4 tot 6 jaar een kleuter, van 6 tot 12 jaar een kind en van 12 tot 18 jaar een puber. Definieer als aparte klasse een enumeratietype voor de leeftijdsfasen van kinderen. Geef de klasse een methode geefFase voor het opvragen van de fase die correspondeert met een gegeven leeftijd.

TERUGKOPPELING

1 **Uitwerking van de opgaven**

- 5.1 De code is syntactisch correct en dus kan het programma gecompileerd worden. De cast belooft dat de terugkeerwaarde van methode getObject van type Integer is.
Tijdens verwerking wordt een ClassCastException opgegooid omdat het actuele type van de terugkeerwaarde String is in plaats van Integer.
- 5.2 a De compiler geeft een foutmelding: methode bewaar(Integer) van klasse Doos<Integer> mag niet gebruikt worden met een String als actuele parameter.
b Bij de toekenning van een waarde van type Doos<Integer> aan een variabele van type Doos<String> meldt de compiler dat het type Doos<Integer> niet geconverteerd kan worden naar het type Doos<String>.
c De aanroep van bewaar is correct. Een string is een object. De string mag in een Doos<Object> worden geplaatst.
De compiler geeft dezelfde foutmelding als in b bij de toekenning van een waarde van type Doos<Integer> aan een variabele van type Doos<Object>. Integer is een subklasse van Object maar de klassen Doos<Integer> en Doos<Object> zijn kennelijk niet aan elkaar gerelateerd: Doos<Integer> is geen subklasse van Doos<Object>.

- 5.3 Het codefragment kan als volgt luiden:

```
ArrayList<Doos<String>> lijst = new ArrayList<>();
Doos<String> doos = new Doos<>();
doos.bewaar("10");
lijst.add(doos);
```

Klasse ArrayList is een generiek type. Hier wordt een instantie gecreëerd met Doos<String> als actuele typeparameter.

- 5.4 a Klasse Paar<T, S> heeft twee formele typeparameters. De code luidt:

```
package paren;

public class Paar<T, S> {
    private T elem1 = null;
    private S elem2 = null;

    public Paar (T elem1, S elem2) {
        this.elem1 = elem1;
        this.elem2 = elem2;
    }

    public String toString() {
        return "paar: " + elem1 + ", " + elem2;
    }

    public T getElem1() {
        return elem1;
    }

    public S getElem2() {
        return elem2;
    }
}
```

- b We kunnen de klasse Paar testen met de volgende code in de methode main van een testklasse PaarTest:

```
String s = "abc";
Integer i = new Integer(10);
Paar<String, Integer> paar = new Paar<>(s, i);
System.out.println("elem1: " + paar.getElem1());
System.out.println("elem2: " + paar.getElem2());
System.out.println(paar.toString());
```

5.5 Opdracht 1 is correct.

Opdracht 2 leidt tot een waarschuwing omdat bij voorkeur een actuele typeparameter gebruikt moet worden.

Opdracht 3 is correct omdat ArrayList<Integer> een subklasse is van AbstractList<Integer>.

Opdracht 4 leidt tot een waarschuwing omdat typeveiligheid niet gegarandeerd is. Instantie lijst2 is van een ruw type.

Opdracht 5 levert een compilatiefout op. Er is geen ordeningsrelatie tussen AbstractList<String> en AbstractList<Integer>.

Opdracht 6 leidt tot een waarschuwing omdat bij voorkeur een actuele typeparameter gebruikt moet worden.

5.6 a De klasse wordt:

```
public class VergelijkbareDoos<T extends Comparable<T>> {
    private T object;

    public void bewaar(T object) {
        this.object = object;
    }

    public T getObject() {
        return object;
    }

    public String toString() {
        return "Doos: " + object;
    }

    public int vergelijkMet(
        VergelijkbareDoos<T> andereDoos) {
        return object.compareTo(antereDoos.getObject());
    }
}
```

Merk op dat de klassen die als actuele typeparameter fungeren voor T de interface Comparable<T> moeten implementeren en dus methode compareTo moeten implementeren. Daarom kan in methode vergelijkMet methode compareTo op het attribuut object worden aangeroepen.

Merk ook op dat de parameter van methode vergelijkMet type VergelijkbareDoos<T> heeft. Hier wordt de bovengrens niet meer genoemd. De voorwaarde in de kop van de klasse geldt in de hele klasse.

Merk tenslotte op dat de klasse `VergelijkbareDoos` zelf ook de interface `Comparable` zou kunnen implementeren. Methode `vergelijkMet` zou in dat geval omgedoopt moeten worden tot `compareTo`. De kop van de klasse zou er dan als volgt uitzien:

```
public class VergelijkbareDoos<T extends Comparable<T>>
    implements Comparable<VergelijkbareDoos<T>>
```

b De implementatie van methode `max` luidt:

```
public VergelijkbareDoos<T> max(
    VergelijkbareDoos<T> andereDoos) {
    if(vergelijkMet(antereDoos) > 0) {
        return this;
    }
    else {
        return andereDoos;
    }
}
```

- 5.7 a De formele typeparameter `E` staat voor het type van de elementen op de stack.
 b Klasse `Stack<E>` krijgt een attribuut `stack` van type `ArrayList<E>` om de elementen van de stack te bewaren:

```
private ArrayList<E> stack = new ArrayList<>();
```

c De specificaties van de methoden `push`, `isEmpty` en `toString` liggen voor de hand, maar bij `pop` moeten we beslissen wat er wordt teruggegeven als de stack leeg is. We besluiten `null` terug te geven.

```
/**
 * Plaatst een element boven op de stack.
 * @param elem het element
 */
public void push(E elem)

/**
 * Haalt een element van de top van de stack.
 * @return het element bovenop de stack
 *         of null als de stack leeg is
 */
public E pop()

/**
 * Test of de stack leeg is.
 * @return true als de stack leeg is,
 *         false anders
 */
public boolean isEmpty()

/**
 * Levert een String-representatie van de stack.
 * @return een String-representatie van de stack.
 */
public String toString()
```

d De implementatie ziet er bijvoorbeeld als volgt uit (het commentaar is weggelaten):

```
package stack;

import java.util.ArrayList;

public class Stack<E> {

    private ArrayList<E> stack = new ArrayList<>();

    public void push(E elem) {
        stack.add(elem);
    }

    public E pop () {
        if (!stack.isEmpty()) {
            return stack.remove(stack.size() - 1);
        }
        else {
            return null;
        }
    }

    public boolean isEmpty() {
        return stack.isEmpty();
    }

    public String toString() {
        String res = "";
        int i = 0;
        while (i < stack.size()) {
            res = res + stack.get(i);
            i++;
            if (i < stack.size()) {
                res = res + ", ";
            }
        }
        return res;
    }
}
```

De klasse heeft geen constructor nodig.

e Een voorbeeld van eenvoudige testcode:

```
// Maak een stack van Strings en push er drie strings op
// Toon dan de stack.
// Maak de stack vervolgens leeg en controleer of pop
// dan null oplevert
Stack<String> stack1 = new Stack<>();
stack1.push("aap");
stack1.push("noot");
stack1.push("mies");
System.out.println(stack1);
while (!stack1.isEmpty()) {
    System.out.println(stack1.pop());
}
System.out.println(stack1.pop());

// Maak een stack van Integers en push er de getallen
// 1..10 op.
// Vervang het bovenste element en toon de stack.
// Haal er tien elementen af en controleer de
// waarde van isEmpty.
```

```
Stack<Integer> stack2 = new Stack<>();
for (int i = 1; i <= 10; i++) {
    stack2.push(i);
}
stack2.pop();
stack2.push(100);
System.out.println(stack2);
for (int i = 1; i <= 10; i++) {
    stack2.pop();
}
System.out.println(stack2.isEmpty());
```

- 5.8 a We kunnen geen `Stack<char>` maken, want alleen referentietypen kunnen als actuele typeparameter gebruikt worden. We hebben dus de bijbehorende verpakingsklasse nodig; dat is de klasse `Character`.
- b De klasse krijgt een attribuut `expressie` voor de expressie en een constructor die deze een waarde geeft. Voor de conversie van `char` naar `Character` en omgekeerd vertrouwen we op `auto(un)boxing`. De klasse kan als volgt worden geïmplementeerd:

```
package stack;

public class Expressie {

    private String expressie = null;

    public Expressie(String expressie) {
        this.expressie = expressie;
    }

    public boolean haakjesGoed() {
        Stack<Character> stack = new Stack<>();
        for (int i = 0; i < expressie.length(); i++) {
            char c = expressie.charAt(i);
            if (c == '(' || c == '[') {
                stack.push(c);
            }
            else if (c == ')' &&
                    (stack.isEmpty() || stack.pop() != '(')) {
                return false;
            }
            else if (c == ']' &&
                    (stack.isEmpty() || stack.pop() != '[')) {
                return false;
            }
        }
        return stack.isEmpty();
    }
}
```

- c Een minimale testset is:
- De lege string (daarin zijn de haakjes correct genest!)
 - Een expressie zonder haakjes
 - Een enigszins complexe expressie met verschillende soorten haakjes waarin de haakjes correct staan
 - Een expressie met te veel openingshaakjes
 - Een expressie met te veel sluihaakjes
 - Een expressie waarin het aantal van elk type haakje klopt, maar de nesting niet.

De testklasse luidt dan bijvoorbeeld (we gebruiken een hulpmethode om de expressie te creëren en true of false af te drukken):

```
package stack;

public class ExpressieTest {

    public static void main(String[] args) {
        testExpressie(""); // true
        testExpressie("x + y*2 - 17"); // true
        testExpressie("[x] + ((5*3) - y[x+2, 5])"); // true
        testExpressie("(x + ((5*3) - y[x+2, 5]))"); // false
        testExpressie("(x + ((5*3) - y[x+2, 5]))"); // false
        testExpressie("[x + ([5*3] - y[x+2, 5])]"); // false
    }

    private static void testExpressie(String teststring) {
        Expressie expr = new Expressie(teststring);
        System.out.println(expr.haakjesGoed());
    }
}
```

- 5.9 a Geen terugkoppeling.
b Klasse Dag kan als volgt worden geïmplementeerd:

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

- c Om de klasse te testen schrijven we de applicatie DagTest met de volgende code als romp voor de methode main.

```
Dag[] dagen = Dag.values();
System.out.println("De dagen van de week: ");
for (Dag dag: dagen) {
    System.out.println(dag);
}
```

- 5.10 We plaatsen eerst een puntkomma achter de lijst constanten en definiëren de static methode isWeekend als volgt:

```
public enum Dag {
    MAANDAG,
    DINSDAG,
    WOENSDAG,
    DONDERDAG,
    VRIJDAG,
    ZATERDAG,
    ZONDAG;

    public static boolean isWeekend(Dag dag) {
        return (dag == ZATERDAG || dag == ZONDAG);
    }
}
```

Een andere versie van de return-opdracht maakt gebruik van de methode `ordinal` en luidt:

```
return dag.ordinal() >= ZATERDAG.ordinal();
```

Voor een test voegen we aan methode `main` uit de klasse `DagenTest` de volgende code toe:

```
System.out.println("Maandag weekend? " +
    Dag.isWeekend(Dag.MAANDAG));
System.out.println("Zaterdag weekend? " +
    Dag.isWeekend(Dag.ZATERDAG));
```

- 5.11 De code van de methode `isWeekend` is al getoond in de leerkern. De gewijzigde versie van de testopdrachten luidt:

```
System.out.println("Maandag weekend? " +
    Dag.MAANDAG.isWeekend());
System.out.println("Zaterdag weekend? " +
    Dag.ZATERDAG.isWeekend());
```

- 5.12 De gewijzigde klasse `Dag` kan als volgt worden geïmplementeerd:

```
public enum Dag {
    MAANDAG("monday"),
    DINSDAG("tuesday"),
    WOENSDAG("wednesday"),
    DONDERDAG("thursday"),
    VRIJDAG("friday"),
    ZATERDAG("saturday"),
    ZONDAG("sunday");

    private String naamEngels = null;

    private Dag(String naamEngels) {
        this.naamEngels = naamEngels;
    }

    public String getNaamEngels () {
        return naamEngels;
    }

    public boolean isWeekend() {
        return this == ZATERDAG || this == ZONDAG;
    }
}
```

De klasse heeft een attribuut `naamEngels` en een private constructor gekregen. Bij elke dag is als parameter de Engelse naam opgenomen. Methode `getNaamEngels` geeft de Engelse naam van de dag.

Met het volgende codefragment kan de methode `getNaamEngels` worden getest:

```
Dag[] dagen = Dag.values();
System.out.println("De dagen van de week: ");
for (Dag dag: dagen)
    System.out.println(dag + ", " + dag.getNaamEngels ());
```

1 Uitwerking van de zelftoets

- 1 a Omdat beide elementen hetzelfde type hebben heeft de generieke klasse `Duo` maar één formele typeparameter. De implementatie van klasse `Duo<T>` luidt:

```
public class Duo<T> {
    private T elem1 = null;
    private T elem2 = null;

    public Duo(T elem1, T elem2) {
        this.elem1 = elem1;
        this.elem2 = elem2;
    }

    public T getElem1() {
        return elem1;
    }

    public T getElem2() {
        return elem2;
    }

    public boolean equals(Object obj) {
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Duo<T> otherDuo = (Duo<T>) obj;
        return elem1.equals(otherDuo.getElem1())
            && elem2.equals(otherDuo.getElem2());
    }
}
```

- b De variabele `duoLijst` kan als volgt worden gedeclareerd:

```
ArrayList<Duo<String>> duoLijst = new ArrayList<>();
```

- 2 De formele typeparameter `T` van de klasse `BeeldenVerzameling` moet klasse `Beeld` als bovengrens hebben:

```
public class BeeldenVerzameling<T extends Beeld> {
    private ArrayList<T> beelden = new ArrayList<>();

    public void voegToe(T beeld) {
        beelden.add(beeld);
    }
}
```

3 De klasse kan als volgt worden geïmplementeerd:

```
public enum Fase {  
    BABY(0, 2),  
    PEUTER(2, 4),  
    KLEUTER(4, 6),  
    KIND(6, 12),  
    PUBER(12, 18);  
  
    private int minLeeftijd = 0;  
    private int maxLeeftijd = 0;  
  
    private Fase(int minLeeftijd, int maxLeeftijd) {  
        this.minLeeftijd = minLeeftijd;  
        this.maxLeeftijd = maxLeeftijd;  
    }  
  
    public static Fase geefFase(int leeftijd) {  
        for(Fase fase: Fase.values()) {  
            if (leeftijd >= fase.minLeeftijd &&  
                leeftijd < fase.maxLeeftijd) {  
                return fase;  
            }  
        }  
        return null; // geen kind  
    }  
}
```

Er worden twee attributen gedeclareerd, een voor de minimumleeftijd en een voor de maximumleeftijd. De constructor heeft de toegang private. Methode geefFase is een static methode omdat de vraag aan klasse Fase wordt gesteld, niet aan een constante. Er is hier voor gekozen om de methode geefFase null te laten retourneren als de opgegeven leeftijd te klein of te groot is.

- @After II: 20
- @Before II: 19
- Aanroep superklasse-constructor I: 63
- Abstracte klasse I: 134
- Abstracte methode I: 134; III: 138
- ActionEvent III: 62
- Actueel type I: 68
- Actuele typeparameter I: 164
- Actuele-parameterlijst III: 151
- Adapterklasse III: 32
- Alternatieve sleutel II: 125
- Anonieme instantie I: 33
- Anonieme klasse III: 37
- Arithmetic-Exception II: 13
- Array-constante III: 150
- ArrayIndexOutOfBoundsException II: 16
- ArrayList II: 105
- ArrayStoreException II: 16
- assert III: 154
- Associatie I: 30
- atomair II: 101
- Attribuutdeclaratie III: 137
- AWT III: 10, 39
- Bag I: 101
- Basiscomponenten III: 12
- Beëindigd II: 100
- Begrensde typeparameter I: 168
- Begrijpelijke code I: 22
- Bestaande klasse als luisteraarklasse III: 37
- Bezig II: 100
- Binding I: 75
- Binnenklasse III: 34
- Blok III: 143
- BorderLayout III: 22, 23
- Bovengrens I: 168
- Break-opdracht III: 142
- Breakpoint II: 25
- Bron III: 28
- Call stack II: 22, 90
- Case-specificatie III: 144
- Cast III: 153
- Cast in expressie I: 71
- Catch-clausule II: 54, 55; III: 147
- Checked exception II: 46
- ClassCast-Exception I: 69
- ClassNotFoundException II: 50
- CloneNotSupportedException II: 51
- Comboboxmodel III: 58
- Compilatiefout II: 10
- Component III: 11
- Compositepatroon I: 136
- Concretemethode-definitie III: 138
- Concurrency II: 81
- ConcurrentModificationException II: 16
- Connection II: 132
- Constanten I: 25
- Constructie I: 19
- Constructievolgorde I: 64
- Constructor-definitie III: 138
- Container III: 12, 18
- ContentPane III: 13
- continue III: 154
- Control flow II: 86
- Correctheid I: 22
- Creatie-expressie III: 152
- Database Management Systeem II: 129
- Database-server II: 129
- Datalaag II: 139
- DBMS II: 129
- deadlock II: 101
- Debugger II: 21
- Declaratieopdracht III: 140
- Decrement-opdracht III: 141
- Default exception handler II: 45
- Default-specificatie III: 144
- Delegatie I: 107
- Delegatiemodel III: 28
- Diepe kopie II: 51
- Dimensie III: 152
- Dimension III: 15
- Dispatching II: 100
- Domeinmodel I: 18
- Doorzichtigheid III: 69
- Do-while-opdracht III: 146
- Downcast I: 69
- Drag and drop III: 67
- Driver II: 131
- DriverManager II: 131
- Dynamische binding I: 77
- Eenvoudige control flow I: 23
- Eenzijdige associatie I: 31
- Efficiëntie I: 23
- Enumeratiedefinitie III: 136
- Enumeratiewaarde III: 150
- Event III: 28
- Event handling III: 27
- Event-dispatching-thread III: 39
- EventObject III: 61
- Events voor menu-items III: 77
- Event-wachtrij III: 39
- Exception handler II: 45
- Exception opgooien II: 45
- Exception vangen en afhandelen II: 45
- Exceptionpropagatie II: 45
- Expliciete cast I: 66
- Expressie III: 148
- Expressielijst III: 150
- Extensie I: 169
- final I: 80
- Finally-clausule II: 55
- Floating-point overflow II: 13
- FlowLayout III: 19, 20
- For-each-opdracht III: 147
- Formele typeparameter I: 164, 169, III: 135
- Formele-parameterlijst III: 138
- Formules voor inhoud I: 110
- For-opdracht III: 146
- Geblokkeerd II: 100
- Gebroken-getal III: 149
- Gebruiksmogelijkheid I: 18
- Gecreëerd II: 100
- Gedeclareerd type I: 68
- Geheel getal III: 149
- Gekwalificeerd this III: 35
- Generalisatie I: 30, 58
- Generiek I: 162
- Generieke klasse I: 164
- Geparametriseerd type I: 164
- Gescheiden verantwoordelijkheden I: 24
- Gescheiden-model architectuur III: 59
- Grammatica III: 133
- GridLayout III: 21
- Herbruikbaarheid I: 23
- Herdefinitie van een methode I: 72
- Hiërarchie exceptionklassen II: 47
- Icon III: 70
- If-opdracht III: 144
- IllegalAccessException II: 50
- IllegalArgumentException II: 17
- ImageIcon III: 70
- Implementatie I: 19
- Implementeren van een interface I: 144
- Impliciete cast I: 66
- Importopdracht III: 134

Incrementopdracht III: 141
 IndexOutOfBoundsException II: 15
 Infix-operator III: 153
 Init-opdracht III: 146
 INSERT II: 128
 InstantiationException II: 50
 Integer overflow II: 12
 Interface I: 142
 Interface DatabaseMetaData II: 138
 Interfacedefinitie I: 143, 148; III: 136
 InterfaceElement-Definitie III: 136
 Interleaving II: 91
 Invoerfocus III: 64
 ItemEvent III: 63

Java Database Connectivity II: 129
 java.awt.Color III: 26
 java.awt.event III: 30
 java.awt.Font III: 26
 Java-bestand III: 134
 JButton III: 57
 JCheckBox III: 57
 JColorChooser III: 74
 JComboBox III: 58
 JDBC II: 129
 JDBC-Driver II: 129
 JDialog III: 72
 JFileChooser III: 73
 JFrame III: 14
 JList III: 59
 JOptionPane III: 74
 JRadioButton III: 57
 JScrollPane III: 71
 JTabbedPane III: 72
 JToggleButton III: 57

Karakter III: 150
 Klasse Class II: 49
 Klasse EnumSet I: 176
 Klasse Error II: 48
 Klasse Exception II: 48
 Klasse JOptionPane II: 62
 Klasse Object I: 61
 Klasse ondersteunt event III: 62
 Klasse RuntimeException II: 48
 Klasse Thread implementeert zelf
 Runnable II: 98
 Klasse Throwable II: 46
 Klassendefinitie I: 135; III: 134
 Kleine klassen en methoden I: 23
 Kleur III: 26
 Kleur en lettertype III: 69
 Kolom II: 125
 Kop van een klasse I: 169

Lage koppeling I: 26
 Layout manager III: 19
 LegeOpdracht III: 141
 Letterlijke-waarde III: 149
 Lettertype III: 26
 Lijst I: 101
 Link I: 33
 ListSelectionEvent III: 64
 Lokale klasse III: 36
 Lokaliteit I: 25
 Long overflow II: 12
 Luisteraar III: 28

Markeerinterface I: 148
 Methode Thread.sleep II: 95
 Methodeaanroep III: 151
 Methodedefinitie I: 135; III: 138
 Methodeopdracht III: 141
 Modaal venster III: 73
 Modale dialogen III: 72
 Model III: 107
 Model-View-Controller III: 120
 MouseListener III: 28
 Multipliciteit I: 32
 MVC III: 120

NaN II: 13
 native III: 154
 NegativeArraySizeException II: 16
 Null layout III: 19
 NullPointerException II: 17
 NumberFormatException II: 11

Object relational mapping II: 140
 Observable III: 111
 Observer III: 111, 112
 Onderhoud I: 21
 Ondiepe kopie II: 51
 Ontwerp I: 19
 Ontwerpmodel I: 19
 Ontwerppatroon I: 136; III: 112
 Onveilige conversie I: 66
 Onveilige toekenning I: 69
 Opdracht III: 140
 Open/closed-principe I: 83, 113
 Operator instanceof I: 71
 ORM II: 140
 OutOfMemoryError II: 13
 Overloading I: 74

package I: 60
 Packagedeclaratie III: 134
 Packagenaam III: 134
 Polymorfisme I: 76

Positie en afmetingen III: 69
 Prefix-operator III: 153
 PreparedStatement II: 133
 Primaire sleutel II: 125
 PrimitiefType III: 139
 private I: 60
 protected I: 60
 Protocolnaam II: 130
 Prototype I: 20
 public I: 60

Recursie II: 14
 Referentietype III: 139
 Resultaattabel II: 127
 ResultSet II: 135
 ResultSetMetaData II: 137
 Resume II: 25
 Return-opdracht III: 142
 Robuust II: 12
 Robuustheid I: 22
 Rol I: 31
 run II: 88
 Runnable II: 88
 Ruwe type I: 167

Samengestelde-expressie III: 153
 Scheduler II: 100
 Schuifbalken III: 71
 SELECT II: 127
 Sequentiediagram II: 22
 setIcon III: 70
 Single-threadregel III: 81
 Software-ontwikkeling I: 16
 Specialisatie I: 57
 SQL II: 126
 StackOverflowError II: 14
 Startklaar II: 100
 Step Into II: 25
 Step Over II: 25
 Step Return II: 25
 Stored procedures II: 139
 strictfp III: 154
 StringIndexOutOfBoundsException
 II: 16
 Strokendiagram II: 126
 Subklassendefinitie I: 80
 super I: 79
 Swing III: 11
 Switch-opdracht III: 144
 Synchronized (klasse) II: 105
 Synchronized (methode) II: 106

Tabbladen III: 71
 Tabel II: 124
 Target II: 89
 Technisch ontwerp I: 19
 Terminate II: 25
 Terugkeertype III: 138
 Terugkoppeling I: 11
 Testen I: 20
 Third-party implementatie I: 149
 this I: 79
 this(..) I: 65
 Thread II: 81, 87, 89
 Thread 'AWT-EventQueue' III: 39
 Thread 'PostEventQueue' III: 39
 Thread.yield II: 100
 Thread-safe III: 81
 Throw-opdracht II: 61; III: 143
 Throws-clausule II: 53; III: 136
 Timer III: 82
 Toegang III: 135
 Toekenning III: 141
 Top-levelcontainer III: 12, 13
 Transactie II: 138
 transient III: 154
 Try-opdracht II: 54, 55; III: 147
 Tweezijdige associatie I: 31
 Type III: 138
 Type casting I: 66

 Uitbreidbaarheid I: 22
 Unchecked exception II: 46
 Unified Modeling Language (UML)
 I: 27
 Unified Process (UP) I: 20
 Unitdefinitie III: 134
 Upcast I: 69
 UPDATE II: 128
 URL II: 130

 Variabele III: 151
 Variabelendeclaratie III: 137
 VariabelType III: 140
 Vector is wel synchronized II: 105
 Veilige conversie I: 66
 Veilige toekenning I: 69
 Verbieden herdefinitie methoden
 I: 80
 Verbieden subklassen I: 80
 Verwerkingsfout II: 11
 Verwerkingspunt II: 26
 Verwijssleutels II: 126
 Verzameling I: 101
 View III: 107
 Virtuele toetscode III: 65
 volatile II: 92; III: 154

 Volgorde bij meer luisteraars III: 31
 Volgorde is onvoorspelbaar II: 91
 Voorkeursafmeting III: 20
 Voortraject I: 17
 Vraagtaal II: 126

 Waardelijst III: 137
 Waardenaam III: 150
 While-opdracht III: 145

 Zichtbaarheid III: 69