Inhoud leereenheid 2

Overerving (1)

Introductie *59*

Leerkern *60*

1 Specialisatie en generalisatie *60*
2 Functionaliteit aan een klasse toevoegen *62*
2.1 Toegangsspecificaties *63*
2.2 Definitie van subklassen *65*
2.3 Constructie van instanties van een subklasse *66*
2.4 Overerving en het typesysteem *70*
3 Herdefinitie *76*
3.1 Herdefinitie en overloading van methoden *76*
3.2 Dynamische binding *80*
3.3 Herdefinitie van attributen *82*
3.4 Het sleutelwoord super *83*
3.5 Verbieden van subklassen en herdefinitie methoden *84*
4 Een toepassing: uitbreiding van de bank *85*

Samenvatting *88*

Zelftoets *90*

Terugkoppeling *94*

1 Uitwerking van de opgaven *94*
2 Uitwerking van de zelftoets *100*

Leereenheid 2

Overerving (1)

INTRODUCTIE

In de cursus Objectgeoriënteerd programmeren in Java 1 hebt u kennis gemaakt 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 in Java 1 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 geïnitialiseerd 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 subklassendefinitie 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

In leereenheid 5 van Objectgeoriënteerd programmeren in Java 1 zijn de eerste beginselen van overerving behandeld. Lees deze leereenheid door als u zich die stof niet meer goed herinnert.

De studielast van deze leereenheid bedraagt circa 8 uur.

LEERKERN

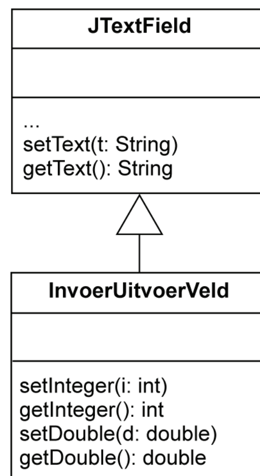
1 Specialisatie en generalisatie

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.



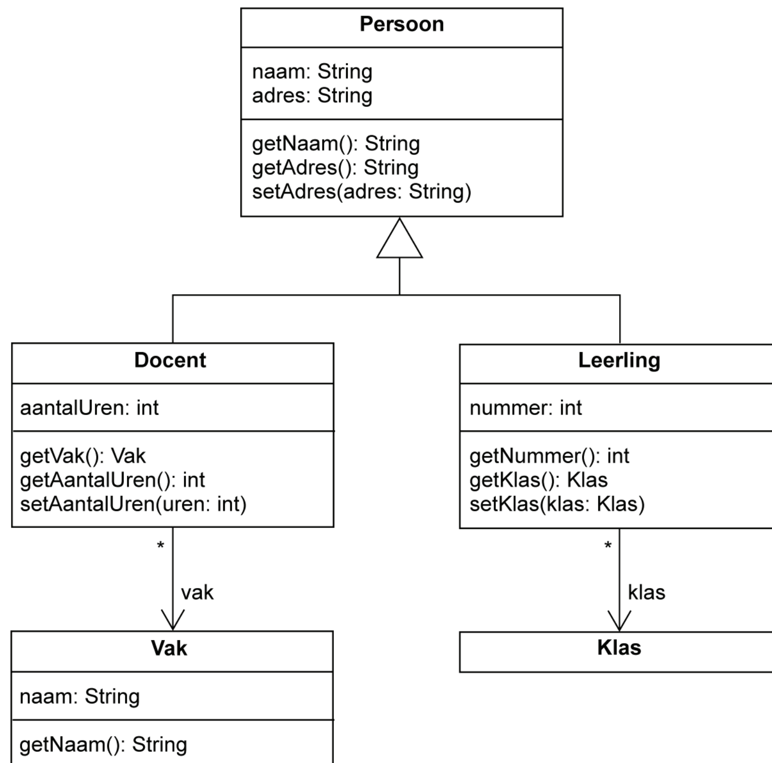
FIGUUR 2.1 Voorbeeld van specialisatie

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.

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

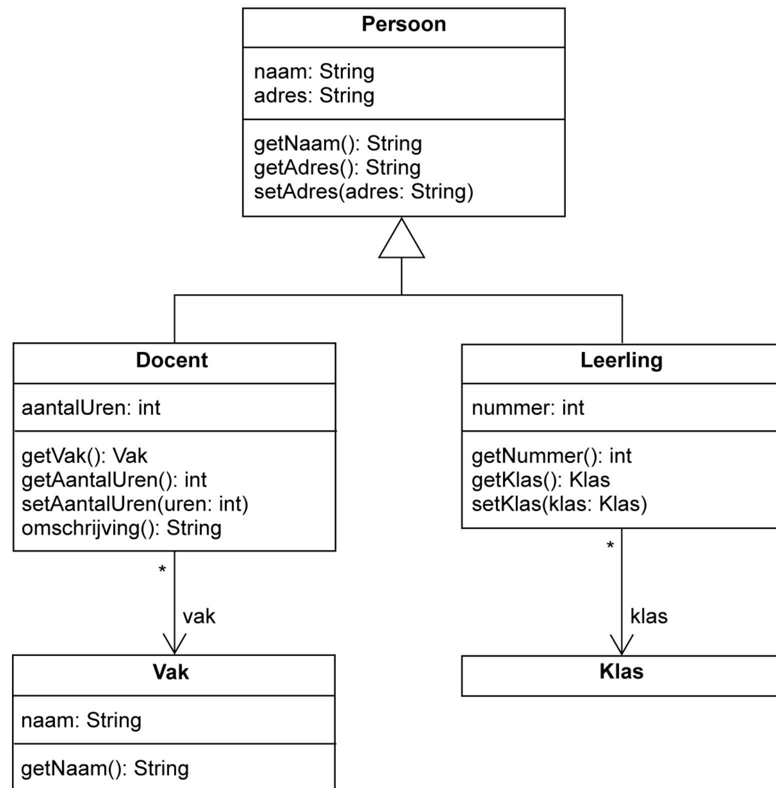
In dit voorbeeld worden eerst twee klassen ontworpen, vervolgens worden de gemeenschappelijke delen daaruit gelicht en ondergebracht in een superklasse. We spreken wel van *generalisatie*: de programmeur definieert een superklasse bij twee eerder ontworpen (sub)klassen.

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).

Opgave 2.1 illustreert een uiterst belangrijk punt. De overervingrelatie is een relatie tussen *klassen* en niet tussen *objecten*. Als een subklasse zoals Docent geïnstantieerd 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 subklassen (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 subklassen ontoegankelijk. Dit geldt bijvoorbeeld voor de attributen van de klasse JFrame, die vanuit de subklassen 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 subklassendefinitie 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 subklassen van de klasse, ongeacht in welke package die staan. Een attribuut of methode krijgt dus de toegangsspecificatie protected, wanneer deze toegankelijk moet zijn voor subklassen binnen dezelfde of een andere package.

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 subklassen, 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
a 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?
b 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 stringrepresesentatie 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 geherdefinieerd, zie paragraaf 3.1.

methode equals

De signatuur van de methode equals van Object luidt:

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

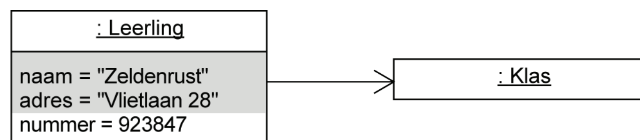
Deze methode vergelijkt het object met een ander object, en geeft true terug wanneer de objecten gelijk zijn, anders false. De implementatie in de klasse Object, die standaard door alle klassen geërfd wordt, is:

```
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 of als beide stringobjecten dezelfde reeks karakters bevatten; het mogen daarbij best verschillende stringinstanties zijn. De methode equals is daartoe in de klasse String geherdefinieerd. 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 geïnitialiseerd.
– 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 geïnitialiseerd 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. In dit voorbeeld is het nut misschien niet zo heel duidelijk; wat is er bijvoorbeeld tegen om te schrijven x = 0; y = 0; in de pararameterloze constructor? In dit geval niet veel. Echter vaak wordt in een constructor nog veel meer gedaan; dat willen we op slechts één plaats implementeren (denk daarbij aan de eis van lokaliteit).

2.4 OVERERVING EN HET TYPESYSTEEM

Het typesysteem van Java vereist dat bij een toekenning de variabele aan de linkerkant en de waarde 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.

Type casting

Een voorbeeld is het volgende programmafragment:

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

Veilige conversie

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*.

Impliciete cast

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

Onveilige conversie

Expliciete cast

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;
```

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 in Java 1) 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 *gedeclareerde* 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 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

betaalrekening

spaarrekening

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.

ClassCast-Exception

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

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*
Tussen A en B bestaat geen superklasse-subklasse relatie A B	*Toekenning nooit toegestaan* Deze fout wordt door de compiler ontdekt.

A is een directe of indirecte superklasse van B
A
B
	Veilige toekenning
	
	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 *upcast* genoemd.
	
	Voorbeeld:
	
	`B b = new B();`
	`A a = b;`

A is een directe of indirecte subklasse van B
B
A
	Onveilige toekenning
	
	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 ClassCastException.
	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 *downcast* genoemd.
	
	Voorbeeld:
	
	`B b = new A();`
	`A a = (A)b;`

Veilige toekenning Upcast

Onveilige toekenning Downcast

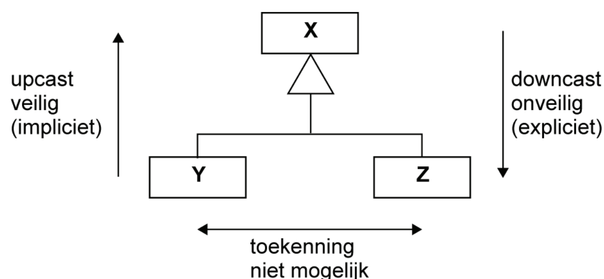
Merk op dat het bij deze regels om statische eigenschappen gaat van de code. Daarom spreken we over de toekenning van een object met *gedeclareerd* type B aan een variabele van (gedeclareerd) type A. Het actuele type van B is uit de code immers niet altijd te achterhalen.

Deze regels zijn niet alleen van toepassing op toekenningsopdrachten 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 een bepaalde variabele uit de superklasse 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 + "]";
}
```

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 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);
```

Voorbeeld 3

Sommige klassendefinities zijn opgesteld met het oog op toekomstige specialisatie, dus met de uitdrukkelijke bedoeling dat er subklassen van worden gedefinieerd. Een voorbeeld is de klasse JApplet: voor iedere concrete applet moet een subklasse van JApplet worden gedefinieerd. Bij het opstellen van de definitie van JApplet (eigenlijk: bij de definitie van Applet, de superklasse van JApplet) is hiermee rekening gehouden. De definitie bevat vier methoden init, start, stop en destroy, die tijdens de levensloop van iedere applet op bepaalde momenten automatisch worden aangeroepen: init wanneer de applet voor het eerst gestart wordt, stop wanneer de applet onderbroken wordt (bijvoorbeeld omdat de gebruiker de pagina verlaat waarin deze is opgenomen), start na init en wanneer de applet na onderbreking weer verder gaat (de gebruiker keert weer terug op de pagina) en destroy wanneer de applet definitief wordt afgesloten. Al deze methoden hebben, binnen de definitie van (J)Applet, een *lege* romp: ze doen helemaal niets! Ze staan er alleen maar om geherdefinieerd te worden.

De init-methode wordt altijd geherdefinieerd omdat daarin de gebruikersinterface wordt opgebouwd. Herdefinities van start, stop of destroy zijn niet altijd nodig.

Ter illustratie geven we de volgende toepassing. Een applet die een complexe animatie uitvoert, legt daarmee een soms fors beslag op de processor en het geheugen van het systeem waarop die draait. Als de gebruiker van dat systeem niet langer naar de applet kijkt, kan de animatie beter stopgezet worden en pas weer herstart worden als de gebruiker er weer wel naar kijkt. Om dit gedrag te verwezenlijken, zullen de methoden start en stop worden geherdefinieerd.

Toelichting

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.

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 even 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 eerder genoemde 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 in Java 1 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 overloaden 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. De specificatie van equals bij Set (een interface; zie leereenheid 4) eist zelfs 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.

Binding

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*.

Als een methode wordt geherdefinieerd, 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.

Polymorfisme

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). 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.

Dynamische binding

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 klassenhiërarchie 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.

super

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.

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 geherdefinieerd, 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 Klassenaam
```

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

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

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

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 verpakkingsklassen? 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

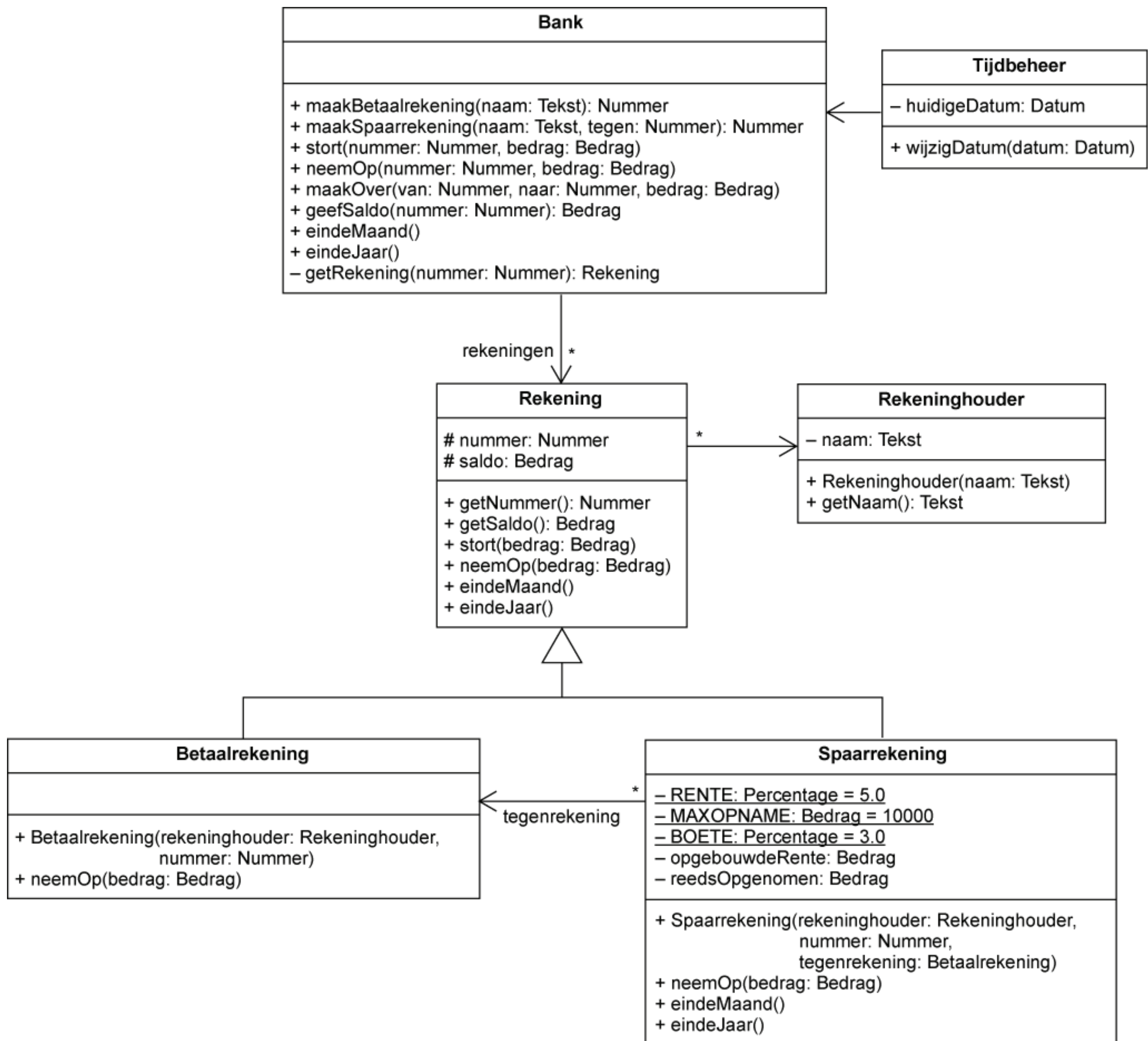
Als we het maken van subklassen van een klasse wel willen toestaan, maar herdefinitie van een bepaalde methode van die klasse willen verbieden, kunnen we in de signatuur van die methode het woord final opnemen:

```
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 niet erg 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.

SAMENVATTING

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 gedeclareerd.

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 *gedeclareerde 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 toegekend 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)d 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) begint het zoeken naar een binding voor methodenaam in de superklasse 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

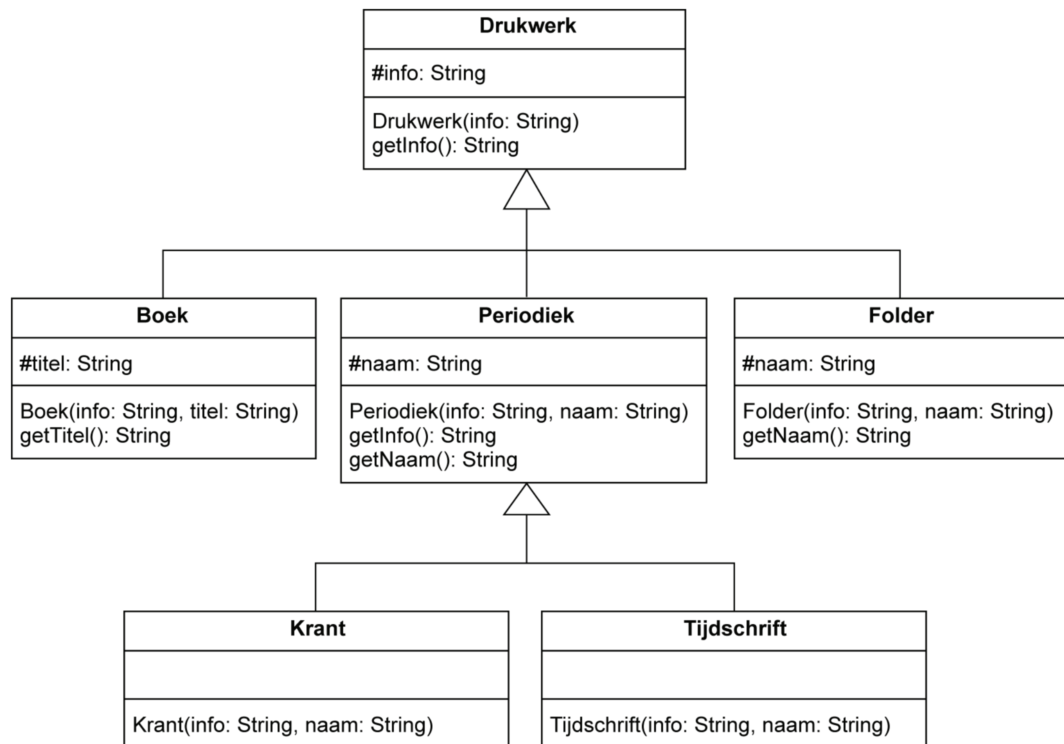
1 a Stel we willen een klasse OmkeerLabel definiëren als subklasse van JLabel, dusdanig dat in een instantie van OmkeerLabel de opgegeven tekst omgekeerd verschijnt. Waarom is de constructor in de volgende definitie van deze klasse onjuist?

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 programmafragment:

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

e Geef de waarde van s2 na verwerking van het volgende programmafragment:

```
Drukwerk d1 = new Krant("Dagblad", "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

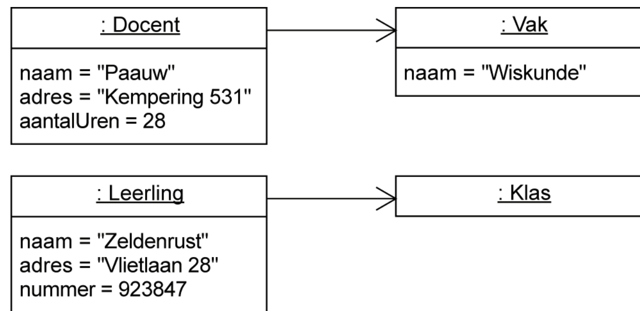
```
public void sluitRekening(int nummer)
```

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 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;
  }
```

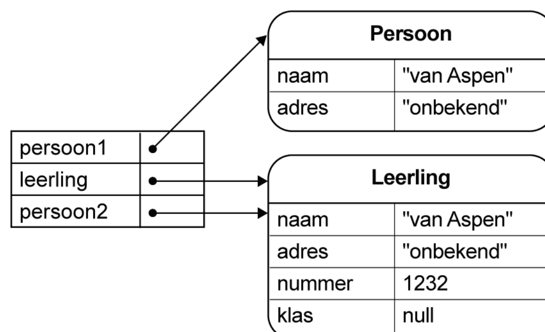
```
    public int getAantalUren() {
      return aantalUren;
    }

    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 uigevoerd; 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. Zowel het gedeclareerde als het actuele type van lwaarde 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.

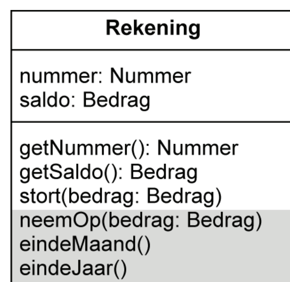
2.11 In t1.plus(2) wordt de implementatie van plus uit Som1 gebruikt; omdat de waarde van term1 gelijk is aan 5, levert deze aanroep de waarde 7 op. In t2.plus(2) wordt de implementatie van plus uit Som2 gebruikt; omdat de waarden van term1 en term2 gelijk zijn aan 3 respectievelijk 4, levert deze aanroep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



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 beleggings-rekeningen.

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();
```

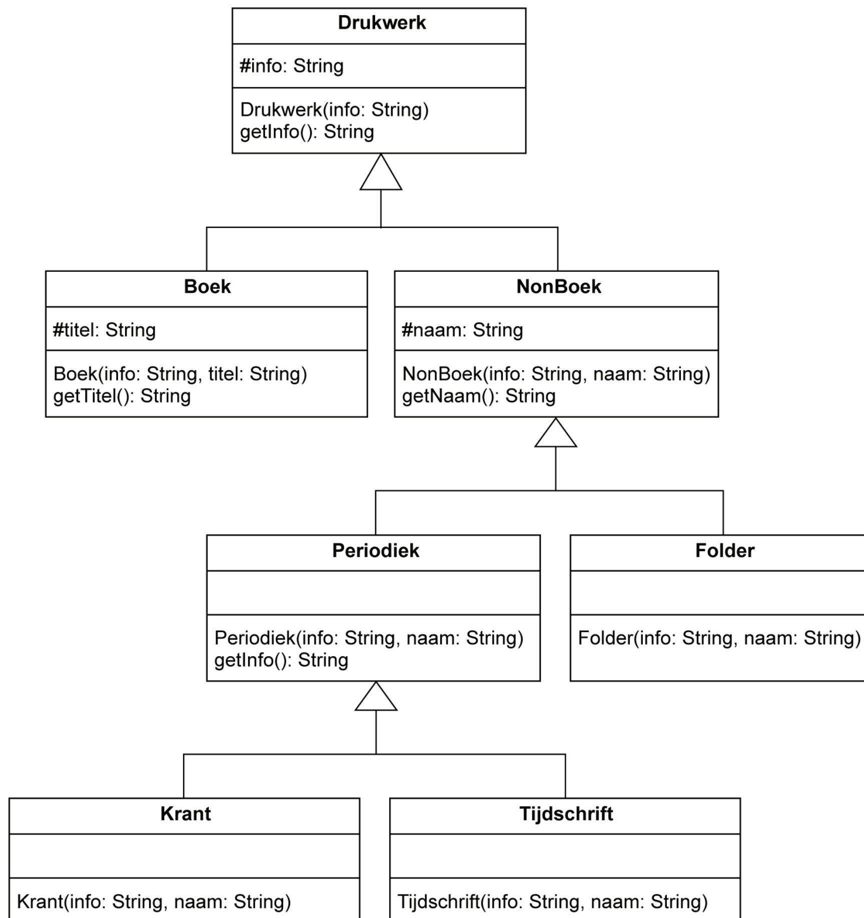
d De methode getInfo behorend bij het actuele type van d1 wordt aangeroepen. Het actuele type is Boek. Boek zelf heeft geen methode getInfo, maar erft deze van Drukwerk. De waarde van s1 wordt daarmee:

```
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<Persoon>();
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<Docent>();
  // 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);
    }
  }
}
```