KU Leuven | Campus De Nayer
Industrieel ingenieur
3Ba Elektronica-ICT | Schakelprogramma Elektronica-ICT

Objectgericht programmeren 1

Academiejaar 2019-20

Lesgever: J. Vennekens
Cursustekst ism. H. Crauwels

II

Inhoudsopgave

1 Inleiding **1**

2 Imperatief programmeren in Java **3**
2.1 Dag wereld! . . . 3
2.2 Commentaar . . . 4
2.3 Variabelen en types . . . 4
2.3.1 Primitieve types . . . 5
2.3.2 Omzettingen tussen types . . . 7
2.4 Rijen . . . 7
2.5 Methodes . . . 8
2.6 Controle-instructies . . . 9
2.6.1 If-then(-else) . . . 9
2.6.2 While . . . 10
2.6.3 For . . . 10
2.7 Voorbeelden . . . 11

3 Objectoriëntatie **13**
3.1 Abstractie in Java . . . 13
3.2 Inleiding tot OO . . . 13
3.2.1 Klassen . . . 13
3.2.2 Objecten . . . 15
3.2.3 Herhaling van rijen . . . 17
3.2.4 Objecten versus primitieve types . . . 18
3.3 Overerving . . . 19
3.3.1 Polymorfisme . . . 21
3.3.2 De klasse Object . . . 24
3.3.3 Een voorbeeld. . . . 24
3.3.4 Types: variabelen versus objecten . . . 26
3.3.5 Typecasting . . . 27
3.3.6 instanceof . . . 28
3.4 Encapsulatie . . . 28
3.5 Statische methodes en variabelen . . . 34
3.6 Abstracte klassen en interfaces . . . 35
3.7 Inwendige klassen . . . 40
3.8 Naamgevingsconventies . . . 40
3.9 Klassediagramma . . . 41
3.9.1 Object . . . 42
3.9.2 Klasse . . . 42
3.9.3 Associatie . . . 44
3.9.4 Generalisatie en overerving . . . 44

4 Gevorderdere features van Java **47**
4.1 Paketten . . . 47
4.1.1 Bestandsstructuur . . . 48
4.1.2 Zelf paketten definiëren . . . 48
4.1.3 Java Language package . . . 48
4.2 Fouten opvangen met Exception . . . 49
4.3 Collecties . . . 53
4.3.1 Vector . . . 54
4.3.2 Generics en collections . . . 54

5 Applicaties met een grafische user-interface **57**
5.1 Het MVC paradigma . . . 57
5.2 Een GUI bouwen . . . 58
5.3 Gedrag aan een GUI koppelen . . . 61
5.4 Een volledig MVC voorbeeld . . . 62
5.5 Nog meer listeners . . . 65
5.6 Nog meer componenten . . . 66
5.7 Menu's . . . 66
5.8 Eigen componenten . . . 67

Hoofdstuk 1

Inleiding

Java is, samen met C en C++, één van de meest populaire programmeertalen. De taal werd in het begin van de jaren '90 ontwikkeld en sterk gepromoot door *Sun Microsystems*, en is momenteel de voornaamste concurrent van het .NET raamwerk van *Microsoft*.

Java is een *obectgeöriënteerde*, *imperatieve* programmeertaal. Deze omschrijving zelf geeft al aan dat Java twee belangrijke voorouders heeft:

- Java is een opvolger van de imperatieve programmeertaal C;
- Terzelfdertijd is Java ook een opvolger van objectgeöriënteerde programmeertalen zoals *Smalltalk*.

Omwille van het eerste puntje, is het mogelijk om in Java op (bijna) exact dezelfde manier te programmeren als in C. Het tweede puntje betekent echter dat een goede Java programmeur dit zelden of nooit zal doen: de objectgeöriënteerde manier van werken is immers eenvoudiger en levert programma's op met minder bugs, die bovendien beter te onderhouden zijn.

In de volgende twee hoofdstukken wordt dit verder uitgelegd. We beginnen met te kijken naar de overeenkomsten tussen Java en C, en leggen uit hoe "gewone" imperatieve programma's in Java geschreven kunnen worden. Daarna bespreekt het tweede hoofdstuk de objectgeöriënteerde kant van Java en leggen we uit waarom deze nuttig is.

Hoofdstuk 2

Imperatief programmeren in Java

2.1 Dag wereld!

De klassieke manier om een programmeertaal in te leiden is door middel van een voorbeeld dat de tekst "Hello world!" afprint. In Java doet men dit bijvoorbeeld zo:

```
1 public class HelloWorld {
2     public static void main (String[] args) {
3       System.out.println("Hello World!");
4     }
5 }
```

Als we dit programmaatje opslaan in een bestand met als naam `HelloWorld.java`, kunnen we dit als volgt compileren:

```
$ javac HelloWorld.java
```

Als resultaat hiervan wordt een nieuw bestand `HelloWorld.class` aangemaakt, waarin de gecompileerde code zich bevindt. Eénmaal de compilatie achter de rug is, kan het programma worden uitgevoerd:

```
$ java HelloWorld
```

Het resultaat van dit commando is – natuurlijk – dat er `Hello World!` wordt afgeprint.

Laten we nu eens in wat meer detail naar dit programma kijken. Op de eerste lijn zien we dat er een "klasse" gedeclareerd wordt met als naam `HelloWorld`. Over dit concept van een klasse vertellen we later (veel) meer, maar momenteel kan je je dit voorstellen alsof we hier gewoon de naam van het programma opgeven. Het is dan ook niet toevallig dat we dit programma opslaan in een bestand `HelloWorld.java`, en dat we het uitvoeren dmv. een commando `java HelloWorld`.

Na deze klasse-declaratie volgt er dan een *codeblok*, dwz. een aantal regels code die tussen akkolades {} staan. De eerste regel in dit codeblok declareert een *methode*: een methode is een stukje code dat van op een andere plaats kan worden opgeroepen; andere programmeertalen gebruiken hiervoor ook wel de termen *functie* of *procedure*. De methode die we hier declareren heeft:

- Als *naam*: `main`
- Als *resultaattype*: `void` (dwz. dat de methode geen resultaat teruggeeft)
- Als *argument(en)*: `String[] args`

Dit zijn de belanrijkste eigenschappen van deze methode. De betekenis van de sleutelwoorden `public` en `static` komt later nog wel aan bod.

De naam `main` is niet toevallig gekozen. Net zoals bijvoorbeeld in C, is dit de naam die de hoofdmethode van een programma móet hebben. Concreet, als we het volgende commando intypen:

```
$ java BlaBla
```

Dan zal de Java omgeving op zoek gaan naar een methode met als naam `main` die zich binnen een `class BlaBla` bevindt, en deze methode uitvoeren.

In het huidige geval bestaat onze `main` methode uit slechts één *statement* of *opdracht*, die wordt afgesloten met een puntkomma. Deze opdracht zal de tekst "Hello world!" afprinten. Hiervoor wordt gebruikt gemaakt van een methode `println`, die een lijn tekst afdrukt (inclusief het nieuwelijnsymbool "`\n`"). Deze methode "hoort bij" de standaard uitvoer, die we in C zouden aanduiden met `stdout`, maar die in Java `System.out` heet. Ook dit wordt later duidelijker.

Tot slot merken we ook nog even op dat de structuur van het programma volledig gedefiniëerd wordt door de akkolades {}. Deze zijn dus uitermate belangrijk en moeten zorgvuldig in de gaten gehouden worden! De indentatie van het programma (dwz. de witruimte die we openlaten in het begin van sommige lijnen) heeft – net zoals in C, maar in tegenstelling tot talen zoals bijvoorbeeld *Python* – voor de compiler geen enkele betekenis. Voor de menselijke lezer/programmeur is een gepaste indendatie echter zo goed als onontbeerlijk! Een goede teksteditor voor programmeurs zal een programma automatisch juist indenteren op basis van zijn akkolade-structuur.

2.2 Commentaar

Om een programma leesbaar te houden, is commentaar van groot belang. Java kent twee verschillende soorten van commentaar:

- Er is lokale commentaar, die de werking van een bepaald stukje code toelicht en die relevant is voor mensen die jouw code zouden willen *veranderen*. Hiervoor worden twee slashes // gebruikt, die als effect hebben dat alles van aan de slashes tot het einde van de lijn als commentaar beschouwd wordt.
- Er is globale commentaar, die informatie verschaft die relevant is voor iedereen die jouw code zou willen *gebruiken*. Deze commentaar wordt ingesloten tussen /** en **/, en kan meerdere lijnen omvatten.

Een voorbeeld:

```
1 public class HelloWorld {
2    /** Dit is de main methode van het programma.
3        Het effect van deze methode is dat de tekst
4        "Hello world!" getoond wordt. **/
5     public static void main (String[] args) {
6       // Hier printen we de tekst af
7       System.out.println("Hello World!");
8     }
9 }
```

2.3 Variabelen en types

Net zoals in C en de meeste andere programmeertalen, zijn er in Java natuurlijk variabelen. Het volgende code-fragmentje laat zien hoe we de waarde van twee variabelen bij elkaar kunnen optellen en toekennen aan een nieuwe variabele:

```
1 int x = 5;
2 int y = 10;
3 int z = x + y;
```

Java is een *streng getypeerde* taal: elke variabele heeft een type en in die variabele mogen enkel maar waardes van het type in kwestie worden opgeslagen. De variabelen in het bovenstaande code-fragment zijn alledrie van het type `int` (afkorting van "integer") en kunnen dus een geheel getal bevatten. Java is een stuk strenger in zijn typering dan bijvoorbeeld C. Zo kent Java een apart type `boolean` en is het niet toegestaan om booleaanse waardes op te slaat in `int` variabelen.

Een opdracht als "`int x = 5;`" slaat twee vliegen in één klap: er wordt hier een variabale x van type `int` aangemaakt én er wordt aan deze variabele de waarde 5 toegekend. Het is natuurlijk ook mogelijk om deze twee operaties van elkaar te scheiden:

```
1 int x;
2 x = 5;
```

De levensduur van een variabele loopt vanaf zijn declaratie tot aan het einde van het blok waarin hij gedeclareerd werd. Als we bijvoorbeeld proberen om deze code te compileren:

```
1 class Bla {
2     public static void main (String [] args) {
3     { // begin van een codeblok
4         int x;
5         x=5;
6     } // einde van dat codeblok
7     x = 10;
8     }
9 }
```

zien we dat dit niet werkt:

```
$ javac test.java
test.java:7: cannot find symbol
symbol  : variable x
location: class Bla
       x = 10;
       ^
1 error
```

De variabele x bestaat immers niet meer op het moment dat we deze toekenning proberen te doen. Ook het volgende programma compileert niet:

```
1 class Bla {
2     public static void main (String [] args) {
3         int x;
4         x=5;
5         int x = 10;
6     }
7 }
```

Hier is de reden dat we x trachten te declareren op een plaats waar deze variable reeds bestaat:

```
$ javac test.java
test.java:5: x is already defined in main(java.lang.String[])
           int x = 10;
               ^
1 error
```

2.3.1 Primitieve types

Java heeft een aantal ingebouwde primitieve datatypes, waarvan de volgende de belangrijkste zijn:

- `int`: gehele getallen (voorgesteld in 32 bits)
  Voorbeelden: `17, -27, 88999912`
- `long`: grote gehele getallen (voorgesteld in 64 bits)
  Voorbeelden: `17, -27, 88999912, 5000000000`
- `float`: reëele getallen (voorgesteld als een 32 bit floating point getal)
  Voorbeelden: `1.0f, 3.4f`
- `double`: reëele getallen met extra precisie (voorgesteld als een 64 bit floating point getal)
  Voorbeelden: `1.0, -4.6, 3.4d, 5.4e9`
- `boolean`: een booleanse waarde
  Voorbeelden: `true`, `false`
- `char`: een letter
  Voorbeelden: `'a', 'b', 'c', 'z'`
- `String`: hoewel het technisch gezien geen primitief type is en dus eigenlijk niet in deze sectie thuishoort, vermelden we hier toch ook even dit type, waarin tekst kan worden voorgesteld.
  Voorbeelden: `"hAllo", "17 flesjes bier", "To be! Or not to be?!??"`

Elk van deze types laat ook een aantal bewerkingen toe. Voor de numerieke types zijn er, zoals je zou verwachten: `+,-,*,/, %`. (Het `%`-teken is de modulo-bewerking, die de rest bij gehele deling produceert.) Daarnaast zijn er ook de decrement en increment operators `++` en `--`, die als volgt gebruikt worden:

```
1 int i = 9;
2 System.out.println(i++); // Print 9 af
3 // Nu is i gelijk aan 10
4 System.out.println(++i); // Print 11 af
5 // Nu is i gelijk aan 11
6 System.out.println(--i); // Print 10 af
7 // Nu is i gelijk aan 10
8 System.out.println(i--); // Print 10 af
9 // Nu is i gelijk aan 9
```

Als deze operators vóór een variabele geplaatst worden, hebben ze dus het effect dat de waarde van de variable met 1 verminderd/vermeerderd wordt vooraleer ze gebruikt wordt; als ze daarentegen ná de variabele komen, wordt de variabele eerst gebruikt en dan pas met 1 verminderd/vermeerderd.

Er zijn ook nog volgende afkortingen:

```
1 double x = 9.0;
2 x += 11; // Afkorting voor x = x + 11
3 x *=  2; // Afkorting voor x = x * 2
4 x -= 11; // Afkorting voor x = x - 11
5 x /=  2; // Afkorting voor x = x / 2
```

De vergelijkingsoperatoren:`==`, `!=` `>`, `<`, `>=`, `<=` produceren elk een booleaanse waarde (`true` of `false`), die aangeeft of het linkerlid respectievelijk gelijk is aan het rechterlid, verschillend is van het rechterlid, strikt groter is dan het rechterlid, strikt kleiner is dan het rechterlid, groter of gelijk is aan het rechterlid, of kleiner of gelijk is aan het rechterlid.

Booleaanse waarden kennen de volgende operatoren: `!, &&, ||`, die respectievelijk "niet", "en", en "of" voorstellen. Bijvoorbeeld:

```
1 8 == 8; //true
2 8 == 8 && 8 != 8; // false
3 8 == 8 || 8 != 8; // true
4 !(8 == 8 || 8 != 8); // false
```

Voor strings vermelden we tot slot nog de concatentatie-operator +, die twee strings aan elkaar plakt:

```
1 String h = "Hello";
2 String w = "world";
3 System.out.println(h + " " + w + "!"); // Print "Hello world!" af
```

2.3.2 Omzettingen tussen types

Sommige types zijn meer algemeen dan andere types, in de zin dat elke waarde voor het ene type ook een geldige waarde voor het andere type is. Zo is `long` bijvoorbeeld algemener dan `int`, en is `float` algemener dan `int`. Het is toegestaan om een waarde van een bepaald type op te slaan in een variabele van een *algemener* type. Bijvoorbeeld:

```
1 int x = 8;
2 float y = x;
3 long z = 8;
```

De omgekeerde operatie mag niet:

```
1 float x = 8;
2 int y = x;
```

Dit produceert:

```
bash-3.2$ javac test.java
test.java:4: possible loss of precision
found   : float
required: int
        int y = x;
                ^
1 error
```

Hoewel de compiler hier weigert om stilzwijgend de `float`-waarde om te zetten naar een `int`, kunnen we hem wel expliciet opdragen om dit te doen met behulp van een zogenaamde *typecast*, waarbij we het type dat we wensen tussen haakjes voor de uitdrukking schrijven:

```
1 float x = 8.9f;
2 int y = (int) x; // Nu is y gelijk aan 8
```

Merk op dat een typecast van een `float` naar een `int` het gedeelte na de komma gewoon weggooit; er wordt dus niet afgerond.

2.4 Rijen

Java beschikt ook over ingebouwde *rijen* of *arrays*. Alle waardes in een rij moeten van hetzelfde type zijn. Een rij van gehele getallen declareer je bijvoorbeeld zo:

```
1 int[] rij;
```

Een rij van strings zou zijn:

```
1 String[] rij;
```

Er zijn twee manieren om een rij aan te maken.

```
1 int[] rij = new int[50];
```

Deze opdracht maakt een nieuwe rij van 50 elementen aan. De lengte van een rij is onveranderlijk en wordt vastgelegd bij het aanmaken. De elementen van de aangemaakte rij worden allemaal op 0 geïnitializeerd.

Het is ook mogelijk om met één commando een rij aan te maken én te initializeren. Dit gebeurt zo:

```
1 int[] rij = { 1, 2, 3 };
```

De lengte van de rij wordt hierbij automatisch berekend op basis van de initializator. Hierboven is dat dus een rij van lengte 3.

De lengte van een rij kan worden opgevraagd met het attribuut `rij.length`:

```
1 int[] rij = { 1, 2, 3 };
2 System.out.println(rij.length); // Print 3 af
```

De elementen van een rij r van lengte n worden aangeduid als:

rij[0], rij[1], ... rij[n-1]

Bijvoorbeeld:

```
1 int[] rij = { 1, 2, 3 };
2 System.out.println(rij[1]); //Print 2 af
3 rij[2] = 4;
4 rij[0] = 0;
5 // Rij is nu: { 0, 2, 4 }
```

Java laat ook multidimensionale "rijen" toe, zoals bijvoorbeeld:

```
1 int aantal_rijen = 5;
2 int aantal_kolommen = 3;
3 int[][] matrix = new int[aantal_rijen][aantal_kolommen];
4 matrix[4][2] = 10;
5 int [][] matrix2 = { {1,2}, {3,4} };
```

Aangezien je rijen enkel maar kan gebruiken indien je bij het aanmaken reeds (een bovengrens op) hun lengte kent, kiezen Java programmeurs vaak voor alternatieven als bijvoorbeeld `Vectors` of `ArrayLists` – hierover meer later in de curus.

2.5 Methodes

Eén van de belangrijkste manieren om de complexiteit van een ingewikkelde programma toch beheersbaar te houden, is door het op te splitsen in een aantal kleinere delen, die elk voor instaan voor één bepaald, goed afgelijnd aspect van de functionaliteit van het hele programma. Zoals we later nog zullen zien, biedt Java hiervoor heel wat ondersteuning. De meest eenvoudige vorm van ondersteuning, die we terugvinden in elke hedendaagse imperatieve programmeertaal, is het opdelen van het programma in verschillende methodes (of functies, of procedures), al naargelang de terminologie van de taal in kwestie).

We hebben al één methode besproken, namelijk de `main` methode die het verloop van het programma bepaalt. Het is mogelijk om nog andere methodes – hulpmethodes – te definiëren, die dan door de `main` methode worden aangeroepen, en die op hun beurt weer nog andere hulpmethodes kunnen aanroepen, enzovoort.

Het grote voordeel hiervan is dat een goede opdeling in methodes de code veel overzichtelijker en eenvoudiger te begrijpen maakt.

Een methode definiëren gaat als volgt:

```
1 resultaatType methodeNaam(Type1 arg1, Type2 arg2, ...) {
2   // Codeblok
3 }
```

Een dergelijke methode moet dus een resultaat van het type `resultaatType` teruggeven. Dit gebeurt door middel van het sleutelwoord `return`, zoals bijvoorbeeld:

```
1 int volgendeGetal(int i) {
2   return i + 1;
3 }
```

Sommige methodes geven geen resultaat terug. Dit wordt aangegeven met het sleutelwoord `void`, zoals in:

```
1 void methodeNaam(Type1 arg1, Type2 arg2, ...) {
2   // Code blok
3 }
```

Optioneel kunnen er voor het resultaattype nog een aantal sleutelwoorden komen, waarvan we de betekenis later zullen uitleggen.

```
1 private resultaatType methodeNaam(Type1 arg1, Type2 arg2, ...) {
2   // Code blok
3 }
4 protected final resultaatType methodeNaam(Type1 arg1, Type2 arg2, ...) {
5   // Code blok
6 }
7 public static resultaatType methodeNaam(Type1 arg1, Type2 arg2, ...) {
8   // Code blok
9 }
```

In Java moeten alle methodes zich steeds *binnen* een klasse-declaratie bevinden. Het volgende is een eenvoudig voorbeeld van een methode die gedefinieerd en opgeroepen wordt:

```
1 class Bla {
2   static int telOp(int i, int j) {
3     return i + j:
4   }
5   public static void main (String [] args) {
6     System.out.println(telOp(4,5)); //Print 9 af
7   }
8 }
```

Het woordje `static` is hier noodzakelijk om dit voorbeeld te laten compileren – later in deze tekst zie je waarom.

2.6 Controle-instructies

Een imperatieve programmeertaal heeft natuurlijk ook nood aan instructies waarmee de programmeur kan bepalen in welke volgorde de opdrachten in zijn programma worden uitgevoerd. Deze sectie zet kort de meest voorkomende controle-instructies in Java op een rijtje.

2.6.1 If-then(-else)

De beroemde `if-then` instructie ziet er in Java als volgt uit:

```
1 if (test) {
2   // codeblok
3 }
```

Hierbij is `test` een uitdrukking die een booleaanse waarde teruggeeft. De haakjes rond `test` zijn verplicht. De betekenis van deze uitdrukking is dat het codeblok enkel maar wordt uitgevoerd als

de test resulteert in waarde `true`. De akkolades {} rond het codeblok zijn verplicht als dit blok uit meerdere opdrachten bestaat; als er slechts één opdracht is, mogen ze ook worden weggelaten.

Er is ook de mogelijkheid om een `else`-tak toe te voegen:

```
1 if (test) {
2   // codeblok 1
3 }
4 else {
5   // codeblok 2
6 }
```

Als de test resulteert in waarde `false`, dan wordt hier dus codeblok 2 uitgevoerd.

Een voorbeeld:

```
1 public int absoluteWaarde(int x) {
2   int resultaat;
3   if (x >= 0)
4       resultaat = x;
5   else
6       resultaat = -x;
7   return resultaat;
8 }
```

2.6.2 While

Om een codeblok uit te voeren zolang aan een bepaalde voorwaarde voldaan is, kan een *while*-lus gebruikt worden.

```
1  public static int grootsteGemeneDeler(int x,int y) {
2    int minimum;
3    if (x < y)
4      minimum = x;
5    else
6      minimum = y;
7    int resultaat = minimum;
8    while (x % resultaat != 0 || y % resultaat != 0)
9      resultaat--;
10   return resultaat;
11 }
```

Het idioom om met een *while*-lus over een rij te itereren is als volgt:

```
1 int[] rij = {1,2,3};
2 int i = 0;
3 while (i < rij.length) {
4   System.out.println(rij[i]);
5   i++;
6 }
```

2.6.3 For

Voor iteratie kan ook een `for`-lus gebruikt worden. Een dergelijke lus heeft drie componenten:

- Een *initializatie* die wordt uitgevoerd voor de lus een eerste keer doorlopen wordt;
- Een *test* die bepaalt of de lus doorlopen wordt of niet;

- Een *stap* die wordt uitgevoerd telkens de lus een keer doorlopen geweest is.

Het idioom om een rij te doorlopen met een *for*-lus is:

```
1 int[] rij = {1,2,3};
2 for (int i = 0; i < rij.length; i++) {
3   System.out.println(rij[i]);
4 }
```

In Java 5 kan dit trouwens ook een stuk eenvoudiger geschreven worden:

```
1 int[] rij = {1,2,3};
2 for (int x: rij)
3   System.out.println(x);
4 }
```

2.7 Voorbeelden

Het volgende eenvoudige programma zoekt het grootste en het kleinste element in een rij.

```
1     static void printLargestAndSmallestElements (int[] n) {
2         int max = n[0];
3         int min = n[0];
4         for (int i=1; i < n.length; i++) {
5             if (max < n[i])
6                 max = n[i];
7             if (min > n[i])
8                 min = n[i];
9         }
10        System.out.println("Maximum: " + max);
11        System.out.println("Minimum: " + min);
12    }
```

Als je een rij verschillende keren moet doorzoeken, kan het nuttig zijn om deze eerst te sorteren. Er bestaan een groot aantal verschillende zoekalgoritmes. Een van de meest eenvoudige (maar vaak niet het meest efficiënte) is het algoritme dat *bubble sort* genoemd wordt. De basisoperatie van het algoritme is heel simpel: we vergelijken één element met het volgende element en als deze twee nog niet in de juiste volgorde staan, wisselen we ze om. We doorlopen nu de rij en passen telkens dit eenvoudige procédé toe op. Dit herhalen we net zo vaak als nodig om een gesorteerde rij te bekomen. Op deze manier zien we dat het kleinste element dus uiteindelijk zal "opborrelen" tot op de eerste plaats van de rij.

```
1 import java.util.*;
2
3 class BubbleSort {
4     public static void main(String args[]) {
5         int[] n;
6         n = new int[10];
7         // initialize the array with some random integers
8         Random myRand = new Random();
9         for (int i = 0; i < n.length; i++) {
10            n[i] = myRand.nextInt();
11        }
12        // print the array's initial order
13        System.out.println("Before sorting:");
14        for (int i = 0; i < n.length; i++) {
```

```
15              System.out.println("n["+i+"] = " + n[i]);
16          }
17          boolean sorted = false;     // definition and initialisation
18          // sort the array
19          while (!sorted) {
20              sorted = true;
21              for (int i=0; i < n.length-1; i++) {
22                  if (n[i] > n[i+1]) {
23                      int temp = n[i];
24                      n[i] = n[i+1];
25                      n[i+1] = temp;
26                      sorted = false;
27                  }
28              }
29          }
30          // print the sorted array
31          System.out.println();
32          System.out.println("After sorting:");
33          for (int i = 0; i < n.length; i++) {
34              System.out.println("n["+i+"] = " + n[i]);
35          }
36      }
37  }
```

Hoofdstuk 3

Objectoriëntatie

3.1 Abstractie in Java

Abstractie is een van de manieren waarop mensen proberen ingewikkelde dingen overzichtelijk en begrijpelijk te maken. Het betekent dat een aantal aspecten van de werkelijkheid bewust vergeten of verborgen worden, om iets eenvoudigers over te houden dat we dan gemakshalve de "essentie" noemen. Goede bedrijfsleiders abstraheren voortdurend: "Geef me de hoofdlijnen, bespaar me de details." Ook wie een complex informatiesysteem ontwerpt, probeert best niet alle details van de implementatie de hele tijd in het hoofd te houden.

Java biedt op verschillende niveaus goede ondersteuning voor abstract ontwerp. Een methode bijvoorbeeld verbergt de details van een handeling achter één aanroep vanuit het hoofdprogramma. In dit hoofdstuk bespreken we de volgende mechanismen:

gegevensabstractie: de programma-ontwerper definieert nieuwe gegevenstypes, die vervolgens gebruikt kunnen worden om veranderlijken te declareren, en als parametertype en terugkeertype van methoden.

afscherming van informatie: sommige gegevensstructuren kunnen slechts geïnspecteerd en/of gewijzigd worden door gebruik te maken van bepaalde methode-aanroepen. De gedetailleerde interne opbouw van zo'n gegevensstructuur blijft verborgen.

veralgemening: als verschillende gegevenstypes bepaalde eigenschappen en methoden gemeenschappelijk hebben, dan kan het gemeenschappelijke deel in een afzonderlijke klasse worden ondergebracht.

3.2 Inleiding tot OO

3.2.1 Klassen

Een klasse is een gegevenstype. De mogelijke waarden die een veranderlijke of uitdrukking van een dergelijk type kan aannemen, worden **objecten** genoemd.

Je kan zelf een nieuw gegevenstype opbouwen door bestaande gegevenstypes te combineren in structuren. Stel dat het informatiesysteem dat we ontwerpen, te maken krijgt met bankrekeningen. Vanuit informatie-analytisch standpunt bestaat een bankrekening uit verschillende elementaire gegevens: de naam van de houder, het adres, het rekeningnummer en het saldo. In Java programmeren we dit als volgt.

```
class Bankrekening {
  String naam;
  String adres;
  int rekeningnummer;
```

```
    double saldo;
}
```

(We gaan er even van uit dat een rekeningnummer kan worden voorgesteld als een **int**, wat in België alvast niet klopt.)

De elementen naam, adres, rekeningnummer en saldo noemen we de *eigenschappen* of *attributen* van een bankrekening. Iedere eigenschap heeft een type, net zoals veranderlijken.

Met een bankrekening associëren we niet alleen de vier bovenstaande eigenschappen, maar ook een aantal *handelingen*: stortingen, overschrijvingen en terugtrekkingen. Het raadplegen van de rekeningstand is eveneens een frequente handeling. Deze handelingen brengen we onder in de definitie van de klasse Bankrekening, en wel in de vorm van *methoden*.

```
class Bankrekening {
    private String naam;
    private String adres;
    private int rekeningnummer;
    protected double saldo;

    void stort(double bedrag) {
        saldo += bedrag;
    }
    void trekTerug(double bedrag) {
        saldo -= bedrag;
    }
    double geefStand() {
        return saldo;
    }
    void schrijfOver(double bedrag, Bankrekening begunstigde) {
        trekTerug(bedrag);
        begunstigde.stort(bedrag);
    }
}
```

In de implementatie van een methode mogen de eigenschappen gebruikt worden alsof het gewone veranderlijken waren. Zo kan de methode `stort` het saldo verhogen met de opdracht

```
saldo += bedrag;
```

Methoden kunnen een uitdrukking van een bepaalde waarde teruggeven aan hun oproeper, zoals de methode `geefStand` dat doet met het huidige rekeningsaldo. Als een methode geen waarden teruggeeft, wordt ze gedeclareerd met het terugkeertype **void**.

De klasse `Bankrekening` definieert een volwaardig nieuw gegevenstype in Java. Dat blijkt ondermeer uit de methode `schrijfOver`, die als tweede parameter een uitdrukking van het type `Bankrekening` verwacht.

In de methode `schrijfOver` zien we twee typische manieren om een methode aan te roepen:

```
trekTerug(bedrag);
begunstigde.stort(bedrag);
```

De aanroep van de methode `stort` wordt voorafgegaan door de referentie naar de parameter `begunstigde`, namelijk, precies door de rekening *waarop* moet gestort worden. Bij de aanroep van `trekTerug` is dat niet het geval: de terugtrekking slaat namelijk op de rekening waarvoor de methode `schrijfOver` zelf wordt aangeroepen. Je kan dit (optioneel) benadrukken door gebruik te maken van het sleutelwoord **this**:

```
void schrijfOver(double bedrag, Bankrekening begunstigde) {
    this.trekTerug(bedrag);
    begunstigde.stort(bedrag);
}
```

Een methode wordt binnen eenzelfde klasse geïdentificeerd door de combinatie van haar naam en het aantal en de types van de parameters. De naam alleen hoeft dus niet uniek te zijn! Om dit te illustreren maken we nog een tweede methode met de naam `trekTerug`, ditmaal echter met een beveiliging tegen kredietoverschrijding. De tweede versie neemt naast de parameter `bedrag` ook nog een tweede parameter `limiet`, eveneens van het type **double**, die aangeeft hoeveel het eindsaldo maximaal onder nul mag gaan. Als de limiet overschreden zou worden, gaat de terugtrekking eenvoudigweg niet door.

```
void trekTerug(double bedrag) {
    saldo -= bedrag;
}
void trekTerug(double bedrag, double limiet) {
    if (saldo - bedrag + limiet >= 0)
            saldo -= bedrag;
}
```

De Java-compiler maakt het onderscheid tussen aanroepen van deze twee versies aan de hand van het aantal en de types van de parameters waarmee ze worden opgeroepen. Eventuele dubbelzinnigheden geven aanleiding tot compilatiefouten (en kunnen dus tijdig worden opgelost, voor de gebruikers er last van hebben).

We hadden de tweede versie dus ook kunnen schrijven met gebruikmaking van de eerste versie:

```
void trekTerug(double bedrag, double limiet) {
    if (saldo - bedrag + limiet >= 0)
            trekTerug(bedrag);
}
```

Het gebruik van dezelfde methode-naam voor twee of meer verschillende methoden binnen één klasse heet het *overladen* van die naam. Overladen kan een programma leesbaarder maken, op voorwaarde dat het zinvol gebruikt wordt. Als twee methoden echt verschillende functies hebben, kunnen ze beter verschillende namen dragen.

Een declaratie van een attribuut kan worden vergezeld van één enkele toekenningsopdracht om er een beginwaarde aan toe te kennen. We spreken dan van *initialisatie.* Als een attribuut geen initialisatie krijgt, begint ze automatisch met een waarde die zo goed mogelijk het begrip "niets" uitdrukt. Voor getaltypes als **int** en **double** betekent dit gewoon het getal nul.

We zouden dus het saldo van een rekening voor alle zekerheid in het begin op nul kunnen stellen:

```
class Bankrekening {
    private String naam;
    private String adres;
    private int rekeningnummer;
    protected double saldo = 0.0;
    // ...
}
```

maar dit is strikt genomen niet nodig, aangezien attributen automatisch geïnitialiseerd worden.

3.2.2 Objecten

De eigenlijke geheugen-inhouden die een klasse-gegevenstype vertegenwoordigen, heten *objecten.* Om een object te creëren, volstaat het niet een veranderlijke te declareren. Veranderlijken *verwijzen* slechts naar objecten, en een pas gedeclareerde veranderlijke verwijst nog nergens naar. In Java is er een speciale waarde `null` om aan te duiden dat een veranderlijke nog niet naar een object verwijst.

De meest voorkomende wijze om een nieuw object van een bepaald type te creëren, is door gebruik te maken van het sleutelwoord **new**. Dat sleutelwoord wordt gevolgd door een aanroep van een *constructor.*

Een constructor is iets wat op een methode gelijkt, met drie verschillen:

- de naam van de constructor is precies gelijk aan de naam van de klasse (inclusief hoofdletters en kleine letters);
- een constructor heeft geen resultaattype (zelfs niet **void**);
- een constructor kan alleen maar worden aangeroepen na het sleutelwoord **new**.

Net als andere methoden, kunnen constructoren nul of meer parameters van diverse types aannemen. En net als bij andere methoden bestaat de mogelijkheid van overladen: er kunnen verschillende constructoren in dezelfde klasse optreden, als het aantal of de types van de parameters maar duidelijk verschillen.

De klasse Bankrekening, uitgebreid met een constructor, zou er als volgt kunnen uitzien.

```
class Bankrekening {
    private String naam;
    private String adres;
    private int rekeningnummer;
    protected double saldo;

    Bankrekening(String nm, String ad, int nr, double sa) {
        naam = nm;
        adres = ad;
        rekeningnummer = nr;
        saldo = sa;
    }

    void stort(double bedrag) {
        saldo += bedrag;
    }
    void trekTerug(double bedrag) {
        saldo -= bedrag;
    }
    double geefStand() {
      return saldo;
    }
    void schrijfOver(double bedrag, Bankrekening begunstigde) {
      trekTerug(bedrag);
      begunstigde.stort(bedrag);
    }
}
```

In het opdrachtenblok van een constructor worden typisch de nodige opdrachten geschreven om de attributen een zinvolle beginwaarde te geven. De eerste opdracht in een constructor *mag* een aanroep van een andere constructor zijn. Met het sleutelwoord **this**, gevolgd door een lijst van parameters tussen haakjes, wordt een andere constructor van dezelfde klasse aangeroepen.

De volgende alternatieve constructor laat toe een bankrekening te openen zonder dat het adres van de houder bekend is.

```
Bankrekening(String nm, int nr, double sa) {
    this(nm, "", nr, sa);
}
```

Als de programmeur een klasse declareert zonder constructor, dan maakt de Java-compiler zelf een *standaardconstructor* aan. Deze neemt geen parameters en doet niets. Het zou hetzelfde zijn alsof we schreven

```
Bankrekening() {
```

```
}
```

De declaratie van minstens één constructor, zelfs een die parameters aanneemt, zorgt ervoor dat de compiler *geen* standaardconstructor maakt. De volgende code is dus niet correct:

```
Bankrekening b = new Bankrekening();
```

Er bestaat immers geen parameterloze constructor voor de klasse Bankrekening.

Met onze constructors kunnen we nu in het hoofdprogramma naar hartelust nieuwe bankrekeningen openen.

```
1  class TestBankrekening {
2    public static void main(String[] args) {
3      Bankrekening b;       // declaratie, geen constructie
4      if (b == null)
5          System.out.println("Dit zie je!");
6      // b.stort(2000);  <-- dit mag nu nog niet!
7      b = new Bankrekening("Lieven", "Affligem", 1, 0.0);
8      // b verwijst nu naar een bestaand object vh type Bankrekening
9      if (b == null)
10         System.out.println("Dit zie je niet!");
11     b.stort(2000);
12     b.trekTerug(150);
13     System.out.println("Stand van rekening b: " + b.geefStand());
14     // We gebruiken de alternatieve constructor:
15     Bankrekening b2 = new Bankrekening("Jan", 2, 0.0);
16     b.schrijfOver(100, b2);
17     System.out.println("Stand van rekening b: " + b.geefStand());
18     System.out.println("Stand van rekening b2: " + b2.geefStand());
19   }
20 }
```

Zoals aangegeven in lijn 6, mogen er op een veranderlijke die nog de waarde `null` heeft, vanzelfsprekend nog geen methodes worden toegepast. Als men dit toch probeert, treedt er volgende runtime fout op:

```
$ java  TestBankrekening
Exception in thread "main" java.lang.NullPointerException
at Test.main(Test.java:6)
```

Bovenstaand voorbeeld maakt gebruik van twee verschillende bankrekening-objecten. Voor elk van de twee declareren we een afzonderlijke veranderlijke. Soms is het praktisch een hele collectie objecten (van dezelfde klasse) tegelijk te declareren. Dat gebeurt door middel van een rij, zoals vroeger reeds beschreven. Nu we wat meer weten over objecten, overlopen we nog eens kort hoe dit werkt.

3.2.3 Herhaling van rijen

Net als een veranderlijke van een object-type, is ook een rij-veranderlijke niet meer dan een verwijzing naar een object. De eigenlijke constructie van het rij-object gebeurt met de operator **new**, gevolgd door de naam van het basistype en het aantal elementen van de rij tussen vierkante haakjes.

In het volgende voorbeeld wordt een rij van tien bankrekeningen eerst gedeclareerd, en vervolgens geïnstantieerd.

```
Bankrekening[] lijst;
lijst = new Bankrekening[10];
```

In dit voorbeeld is een rij van 10 bankrekeningen geïnstantieerd; er is echter nog geen enkele bankrekening geopend! Het aantal elementen van een rij-object is onveranderlijk na de instantiatie. Het kan steeds worden opgevraagd met de eigenschap `length`. Ieder element van de rij is op zijn beurt een referentie, en moet afzonderlijk worden geïnstantieerd. Bijvoorbeeld

```
for (int i = 0; i < lijst.length; i++) {
    lijst[i] = new Bankrekening("Lieven", i, 0.0);
}
```

opent tien verschillende bankrekeningen op naam van dezelfde persoon. De elementen `lijst[0]`, `lijst[1]` tot en met `lijst[9]` kunnen gebruikt worden als gewone veranderlijken van het type `Bankrekening`.

3.2.4 Objecten versus primitieve types

Objecten en primitieve types (zoals `int, double` of `boolean`) worden in Java anders behandeld. We overlopen even kort de voornaamste verschilpunten.

- Op objecten kunnen methodes worden toegepast met behulp van de syntax `object.methode()`. Bij primitieve types gaat dit niet.
- Voor objecten moet geheugen gealloceerd worden, normaalgezien met de `new` operator. Primitieve types krijgen automatisch een geheugenplaatsje toegewezen.
- Objecten kennen twee verschillende soorten van gelijkheid: er kan gekeken worden of twee variabelen verwijzen naar objecten op dezelfde *geheugenplaats* door middel van de `==` operator. Daarnaast is er ook de `equals` methode waarmee gekeken wordt of de *inhoud* van twee objecten hetzelfde is. Primitieve types kennen maar enkel maar de gelijkheid `==`, die bij primitieve types de waarde van twee variabelen vergelijkt.

```
1  Integer i = new Integer(5);
2  Integer j = new Integer(5);
3  if (i == j) // is false
4      System.out.println("dit zie je niet!");
5  if (i.equals(j)) // is true
6      System.out.println("dit zie je wel!");
7  int x = 5;
8  int y = 5;
9  if (x == y) // is true
10     System.out.println("dit zie je wel!");
11 if (x.equals(y)) // <-- geeft een compiler fout
12     System.out.println("dit compileert niet eens!");
```

- Bij methode oproepen worden primitieve types met *call by value* doorgegeven, terwijl objecten met *call by reference* worden doorgegeven. Met andere woorden, van primitieve types krijgt de methode een eigen kopie, maar objecten worden zelf doorgegeven.

```
1  public class Test {
2    private int waarde;
3    public void veranderPrimitief(int x) {
4       x = 99;
5    }
6    public void veranderObject(Test x) {
7      x.waarde = 99;
8    }
9    public Test(int i) {
10     waarde = i;
```

```
11    }
12    public static void main(String[] args) {
13      int i = 1;
14      verander(i);
15      System.out.println(i); // geeft 1
16      Test object = new Test(1);
17      verander(object);
18      System.out.println(object.waarde); geeft 99
19    }
20  }
```

Deze verschillen lijken misschien op het eerste zicht niet erg logisch. Ze worden mogelijk wel duidelijker wanneer we beseffen dat een variabele die gedeclareerd is met een object als type eigenlijk niet dit object zelf bevat maar een *referentie* naar een geheugenplaats waar dit object zich bevindt. Als we dit vergelijken met een programmeertaal zoals C, dan zou een Java variabele `Object o` eigenlijk overkomen met een C pointer `Object* o`, en is een `new` hetzelfde als een `malloc`. Bij een vergelijking `o1 == o2` tussen object variabelen `o1` en `o2` worden dus eigenlijk de geheugenadressen van `o1` en `o2` vergeleken. Bij een methode-oproep `methode(o1)` krijgt de methode wel zijn eigen kopie van de verwijzing naar het geheugenadres van `o1`, maar blijft deze kopie natuurlijk gewoon wijzen naar hetzelfde object.

Merk trouwens op dat strings in Java objecten zijn en geen primitieve types. Het enige onderscheid tussen strings en "normale" objecten is dat er een speciale syntax bestaat om snel strings te construeren en concateneren. Het is dus perfect legaal om bijvoorbeeld te schrijven:

```
1  public static void main(String[] args) {
2      System.out.println("tekstje".length()); // print 7 af
3  }
```

Dit betekent dus ook dat bijvoorbeeld ("java" == "java") de booleanse waarde `false` zal opleveren, hoewel "java".equals("java") natuurlijk resulteert in `true`. De meeste Java programmeurs spenderen ooit wel eens minstens een uurtje aan het opsporen van een bug die te wijten is aan dit gedrag, dus u weze gewaarschuwd!

Net zoals strings, hebben ook rijen (*arrays*) hun eigen speciale syntax, maar zijn het voor de rest eigenlijk gewone objecten.

Soms zijn primitieve types nogal onhandig om mee te werken, bijvoorbeeld omdat Java een aantal nuttige functies aanbiedt die een `Object` (de voorouder-klasse van alle klassen in Java – zie ook Sectie 3.3.2) als argument verwachten. Hierom biedt Java voor al de primitieve types ook zogenaamde *wrapper classes* of enveloppeklassen aan. Deze heten `Integer, Double, Boolean, Character` enzovoort. Ze bieden allen een constructor aan met een argument van het overeenkomstige primitieve type en bevatten daarnaast ook nog een aantal andere nuttige methodes, zoals bijvoorbeeld de statische methode `Integer.parseInt(String s)`, waarmee een string zoals "42" kan worden omgezet in zijn corresponderende `int` waarde 42.

3.3 Overerving

Een klasse in een Java programma stelt een bepaalde type van objecten voor. Het komt vaak voor dat een programma nood heeft aan twee verschillende klassen, zodanig dat de ene eigenlijk een *speciale soort* van de andere is. Bijvoorbeeld, een programma voor de personeelsadministratie van een hogeschool kan nood hebben aan een klasse `Werknemer` en een klasse `Docent`. Nu is een docent niets anders dan een *speciaal soort* werknemer. Of korter gezegd, een docent *is* een werknemer.

Dit betekent dat ons programma alles wat het voor werknemers kan doen *ook* voor docenten moet kunnen doen. Als we bijvoorbeeld de naam van een werknemer moeten kunnen opvragen, dan moeten we de naam van een docent natuurlijk *ook* kunnen opvragen, want een docent is nu eenmaal een werknemer. Omgekeerd is het daarentegen natuurlijk niet zo dat elke werknemer een

docent is, dus is het wel mogelijk dat ons programma bepaalde dingen kan doen voor docenten (zoals bijvoorbeeld een lijst van de vakken van die docent opvragen), dat het niet kan doen voor werknemers die geen docent zijn.

Zonder overerving zouden we dit bijvoorbeeld als volgt moeten programmeren:

```
1  public class Werknemer {
2      private Datum geboorteDatum;
3      private String naam;
4      public Werknemer(Datum d) {
5        geboorteDatum = d;
6      }
7      public String getNaam() {
8          return naam;
9      }
10     public void setNaam(String n) {
11         naam = n;
12     }
13 }
14 public class Docent {
15     private Datum geboorteDatum;
16     private String naam;
17     public Docent(Datum d) {
18       geboorteDatum = d;
19     }
20     public String getNaam() {
21         return naam;
22     }
23     public void setNaam(String n) {
24         naam = n;
25     }
26     private Vak[] vakken;
27     public boolean doceert(Vak v) {
28         for (int i = 0; i < vakken.length; i++)
29             if (vakken[i].equals(v))
30                 return true;
31         return false;
32     }
33 }
```

Dit werkt, maar is verre van ideaal: we hebben nu namelijk al onze code uit de `Werknemer`-klasse gedupliceerd. Dit maakt het programma een stuk langer én moeilijker te onderhouden. Als er nu bijvoorbeeld iets verandert aan hoe we de naam van een werknemer/docent voorstellen (misschien willen we die opsplitsen in een voor- en achternaam), dan moeten we deze wijziging op twee verschillende plaatsen doorvoeren.

Om dit te vermijden, kunnen we overerving gebruiken. Door eenvoudigweg "`extends Werknemer`" toe te voegen aan onze definitie van `Docent`, zal deze klasse alle attributen en methodes van `Werknemer` overerven. Onderstaande twee klassen hebben dus allebei een attribuut `naam` en `geboortedatum` met bijhorende methodes:

```
1  public class Docent {
2      private Datum geboorteDatum;
3      private String naam;
4      public Docent(Datum d) {
5        geboorteDatum = d;
6      }
7      public String getNaam() {
8          return naam;
9      }
10     public void setNaam(String n) {
11         naam = n;
12     }
13     private Vak[] vakken;
14     public boolean doceert(Vak v) {
15         for (int i = 0;
16           i < vakken.length; i++)
17             if (vakken[i].equals(v))
18                 return true;
19         return false;
20     }
21 }
```

```
1  public class Docent extends Werknemer {
2
3
4      public Docent(Datum d) {
5        geboorteDatum = d;
6      }
7
8
9
10
11
12
13     private Vak[] vakken;
14     public boolean doceert(Vak v) {
15         for (int i = 0;
16           i < vakken.length; i++)
17             if (vakken[i].equals(v))
18                 return true;
19         return false;
20     }
21 }
```

We kunnen dus nu bijvoorbeeld het volgende doen:

```
1  Docent d = new Docent();
2  d.setNaam("Joost Vennekens");
3  System.out.println(d.getNaam());
```

Dit werkt omdat de methode `setNaam` en het attribuut `naam` dat deze methode manipuleert omwille van de overerving impliciet aanwezig zijn in de klasse `Docent`, ook al hebben we ze niet expliciet getypt. Het verschil tussen deze versie van `Docent` en de vorige is dat deze onze code van de `Werknemer`-klasse *hergebruikt*, terwijl de andere versie deze code *dupliceert*.

Om nog eens samen te vatten, we schrijven

```
1  class X extends Y {
2    ...
3  }
```

als elke X ook een Y is, en als het nodig is dat de functionaliteit van X een uitbreiding is van de functionaliteit van Y. De oorspronkelijke klasse Y wordt ook wel de *superklasse* genoemd en de afgeleide klasse X de *subklasse*.

Dergelijke overerving zorgt ervoor dat alle attributen en methodes van de superklasse ook beschikbaar zijn in de subklasse. Er is hierop één uitzondering, met name constructoren: deze worden niet overgedragen. Vandaar dat onze klasse `Docent` nog steeds zijn eigen constructor moet voorzien. Het is wel mogelijk om in een constructor van een subklasse een constructor van de superklasse aan te roepen. Dit moet gebeuren als eerste instructie van de methode en kan door middel van het sleutelwoord `super`. (Dit is natuurlijk volledig analoog aan het oproepen van een andere constructor van dezelfde klasse dmv. `this`.) Een voorbeeldje:

```
1  public Docent(Datum d) {
2    super (d);
3  }
```

Met het oog op hergebruik van code verdient het natuurlijk aanbeveling om dit te gebruiken waar mogelijk. Deze versie van de constructor is dus beter dan de eerste versie hierboven.

3.3.1 Polymorfisme

Een subklasse erft alle methodes (behalve constructors) van zijn superklasse. Soms zijn deze overgeërfde methodes echter niet precies wat de subklasse nodig heeft. Dit kan in twee gevallen voorkomen:

1. De algemene methode uit de superklasse kan eenvoudiger geïmplementeerd worden als we weten dat we in het speciale geval van de subklasse zitten.

2. De subklasse biedt extra functionaliteit aan en om deze te verwezenlijken, moeten bepaalde methodes uit de superklasse worden uitgebreid met extra instructies.

Dit is een voorbeeld van het eerste geval:

```
1  public class Rechthoek {
2
3      protected double breedte;
4      protected double hoogte;
5
6      public Rechthoek(double h, double b) {
7          breedte = b;
8          hoogte = h;
9      }
10
11     public double omtrek() {
12         return 2* breedte + 2 * hoogte;
13     }
14
15     /** Roteert de rechthoek 90 graden **/
16     public void draai() {
17         double tmp = breedte;
18         breedte = hoogte;
19         hoogte = tmp;
20     }
21 }
22 public class Vierkant extends Rechthoek {
23
24    /** Een Rechthoek met gelijke breedte en hoogte **/
25     public Vierkant(double grootte) {
26         super (grootte, grootte);
27     }
28
29     public double omtrek() {
30       return 4 * breedte;
31     }
32
33     public void draai() {
34     }
35 }
```

We implementeren hier een klasse `Vierkant` en een klasse `Rechthoek`. Aangezien het een wiskundig feit is dat elk vierkant een rechthoek is, is overerving hier inderdaad op zijn plaats. In het speciale geval dat een rechthoek een vierkant is, kunnen we aan aantal methodes eenvoudiger implementeren. Zo heeft een rotatie van 90° geen enkel effect op een vierkant en is de omtrek van een vierkant eenvoudigweg gelijk aan vier maal de lengte van zijn zijden.

Java laat ons toe om de ingewikkeldere, algemene methodes uit de klasse `Rechthoek`-methode te "overschrijven" met eenvoudigere methodes voor het specifieke geval van een `Vierkant`. Dit heeft als effect dat de `Vierkant`-methode zal worden opgeroepen voor elk `Vierkant` object en dat de `Rechthoek` zal worden opgeroepen voor elke `Rechthoek` die géén `Vierkant` is. Met andere woorden, *de meest specifieke methode die van toepassing is*, wordt uitgevoerd.

```
1  Vierkant v = new Vierkant(12);
```

```
2 v.draai(); //De methode uit klasse Vierkant wordt uitgevoerd
3 Rechthoek r = new Rechthoek(5,5);
4 r.draai(); //De methode uit klasse Rechthoek wordt uitgevoerd
```

Het tweede geval dat hierboven vermeld werd, is dat waarin een subklasse meer functionaliteit aanbiedt dan een superklasse, waardoor bepaalde methodes uit de superklasse moeten worden aangepast.

Voor een voorbeeld hiervan, kunnen we terugkeren naar `Docent`en en `Werknemer`s. Elke docent is een werknemer – dus overerving is op zijn plaats – en verder breidt de klasse `Docent` de functionaliteit van de klasse `Werknemer` uit door ook rekening te houden met de vakken die gedoceerd worden door de docent. Laten we nu eens nadenken over wat er moet gebeuren als bijvoorbeeld een werknemer ontslag neemt: een aantal van deze zaken, bijvoorbeeld het opnemen van dit ontslag is het personeelsdossier van de werknemer, zal voor iedereen moeten gebeuren; daarnaast zijn er ook een aantal dingen die te maken hebben met gedoceerde vakken en die dus enkel voor docenten noodzakelijk zijn.

```
1  class Werknemer {
2    ...
3    public void neemtOntslag() {
4      status = "uitdienst";
5      datumUitDienstTreding = new Date(); //de huidige datum
6    }
7  }
8  class Docent extends Werknemer {
9    ...
10   public void neemtOntslag() {
11     status = "uitdienst";
12     datumUitDienstTreding = new Date(); //de huidige datum
13     for (int i = 0; i < vakken.length; i++) {
14       vakken[i].setDocent(null); //geen docent meer
15     }
16     vakken = null; //geen gedoceeerde vakken meer
17   }
18 }
```

We zien dus dat bij een docent eerst hetzelfde gebeurt als bij een gewone werknemer, maar dat er daarna nog iets extra gebeurt. We hebben dit nu opgelost door de instructies uit de `neemtOntslag` methode van `Werknemer` te kopiëren naar de klasse `Docent`, maar dit is natuurlijk – met het oog op het hergebruik van code – weer niet ideaal. Vandaar dat Java het sleutelwoord `super` aanbiedt om methodes uit een superklasse te hergebruiken. We kunnen hiermee onze klasse `Docent` als volgt herschrijven:

```
1  class Docent extends Werknemer {
2    ...
3    public void neemtOntslag() {
4      super.neemtOntslag();
5      for (int i = 0; i < vakken.length; i++) {
6        vakken[i].setDocent(null); //vakken hebben geen docent meer
7      }
8      vakken = null; //docent doceert geen vakken meer
9    }
10 }
```

Tussen haakjes, herinner je dat we een constructor van een superklasse moesten aanroepen met:

```
1 super (argmumenten);
```

dus *zonder* de naam van die constructor te vermelden, terwijl we nu dus voor gewone methodes moeten schrijven:

```
1 super.methodeNaam(argmumenten);
```

3.3.2 De klasse Object

Er is in Java precies één klasse die zelf geen superklasse heeft, en dit is de klasse `Object`. Als je als progammeur een klasse schrijft die geen expliciete superklasse heeft, dan zal de compiler deze klasse toch `Object` als superklasse geven. Dit:

```
public class Bla { ... }
```

is met andere woorden net hetzelfde als dit:

```
public class Bla extends Object { ... }
```

Elke klasse in Java is dus een (directe of indirect) nakomeling van `Object`. Dit betekent dat als je een methode wil schrijven die eender welk object als argument aanvaardt, je dit kan schrijven als:

```
public void aanvaardAlles(Object o) { ... }
```

Java biedt een aantal ingebouwde methodes aan die hier handig gebruik van maken.

Een voorbeeld is de methode `toString` die in de klasse `Object` gedefiniëerd is en die eender welk object omzet naar een String. Zoals je hieronder kan zien, beschikt dus ook elke klasse die je zelf schrijft over een overgeërfde implementatie van deze methode:

```
1 class Test {
2     public static void main(String[] args) {
3         Test t = new Test();
4         System.out.println(t);
5     }
6 }
```

Dit produceert:

```
$ java Test
Test@66848c
```

De standaard implementatie die wordt overgeërfd produceert dus de naam van de klasse van het object in kwestie en het geheugenadres waarop dit zich bevindt. Dit is vaak niet erg nuttig, dus als je een eigen klasse maakt, is het vaak een goed idee om je eigen `toString()`-methode te schrijven.

3.3.3 Een voorbeeld.

Stel dat we een nieuw gegevenstype Spaarrekening willen invoeren. Een spaarrekening is een bankrekening waarop intrest wordt uitgekeerd. Alle eigenschappen en methoden van een gewone bankrekening zijn van toepassing op deze bijzondere soort, maar bovendien moeten we een extra eigenschap hebben om de rentevoet te onthouden.

```
class Spaarrekening extends Bankrekening {
    private double rentevoet; // in fracties, dus 10 procent is 0.1
}
```

We kunnen nu gewoon programmeren met spaarrekeningen, gebruikmakend van alle eigenschappen en methoden van een spaarrekening, zowel de rentevoet als de geërfde eigenschappen en methoden. We moeten wel nog een constructor voorzien.

```
Spaarrekening(String nm, String ad, int nr, double sa, double rv) {
    super(nm, ad, nr, sa);
    rentevoet = rv;
}
```

Als we aan het einde van het jaar de intrest willen uitkeren, dan hangt die natuurlijk af van de tijdsduur waarover het geld op de spaarrekening heeft gestaan. We lossen dat op door een interne veranderlijke voorlopigeIntrest die aan het begin op nul wordt gezet, en die wordt aangepast bij elke storting (in positieve zin) en bij elke terugtrekking (in negatieve zin). De aanpassing is evenredig met het bedrag van de operatie en met de rentevoet, maar ook met de tijd die er nog rest in het lopende jaar. De volgende tabel illustreert dit aan de hand van enkele voorbeelden, bij een rentevoet van 0.05 ofwel 5 percent:

bewerking	bedrag	datum	resterende fractie van een jaar	invloed op de intrest
storting	1000	1 jan	1.00	+50.00
storting	1000	1 jul	0.50	+25.00
terugtrekking	1000	1 okt	0.25	-12.50
storting	5000	1 dec	0.0833...	+20.83

Een spaarrekening die op 1 januari is geopend en waarop de vier bovenstaande verrichtingen zijn uitgevoerd, levert dus op 31 december een intrest van 83.33 op. Het eindsaldo bedraagt 6083.33

De definitie van de klasse Spaarrekening gaat er dan als volgt uitzien.

```
1  class Spaarrekening extends Bankrekening
2  {
3      private double rentevoet;     // in fracties, dus 10 procent is 0.1
4      private double voorlopigeIntrest;
5      /** Bedrag van de intrest op 31 december, als
6       *  geen verdere verrichtingen meer plaatsvinden.
7       */
8
9      Spaarrekening(String nm, String ad, int nr, double sa, double rv)
10     {
11         super(nm, ad, nr, sa);
12         rentevoet = rv;
13         voorlopigeIntrest = saldo * rentevoet;
14     }
15     void stort(double bedrag)
16     {
17         super.stort(bedrag);
18         voorlopigeIntrest = voorlopigeIntrest +
19                             bedrag * rentevoet * Datum.jaarfractie;
20     }
21
22     void trekTerug(double bedrag)
23     {
24         super.trekTerug(bedrag);
25         voorlopigeIntrest = voorlopigeIntrest -
26                             bedrag * rentevoet * Datum.jaarfractie;
27     }
28     /** Deze methode mag slechts 1 keer per jaar,
29      *  op 31 december, worden uitgevoerd.
30      */
31     void keerIntrestUit()
32     {
33         saldo += voorlopigeIntrest;
34         voorlopigeIntrest = saldo * rentevoet;
35     }
36 }
```

In de implementatie van de methodes die het saldo aanpassen maken we gebruik van een veranderlijke "Datum.jaarfractie"die aangeeft welk deel van het jaar er nog rest. Dit kan door een extra klasse te voorzien met één statische variabele:

```
class Datum {
    static double jaarfractie;
}
```

Oefening. Schrijf een programma met de naam TestSpaarrekening dat een spaarrekening opent en vervolgens de vier verrichtingen uit de tabel uitvoert. Verifieer op het einde dat het eindsaldo na het uitkeren van de intrest inderdaad 6083.33 bedraagt.

Om de voorlopige intrest telkens juist te berekenen, moet voor elke bewerking op de rekening een toekenningsopdracht uitgevoerd worden van de vorm:

```
Datum.jaarfractie = 0.50;    // (of een ander getal)
```

3.3.4 Types: variabelen versus objecten

Eigenlijk heeft een variabele in Java niet één maar twee types. Er is in de eerste plaats het type waarmee de variabele zelf gedeclareerd geweest is. Daarnaast is er echter ook nog het type van het *object* waarnaar een variabele verwijst. Het eerste type is statisch – je kan het gewoon aflezen uit de declaratie van de variabele – en blijft gedurende de hele levensduur van de variabele hetzelfde. Het tweede type is dynamisch en kan veranderen.

Natuurlijk moet er wel een overeenkomst zijn tussen deze twee types: een variabele mag enkel maar refereren naar objecten van een type dat een *subklasse* is van de klasse waarmee de variabele gedeclareerd werd. Met andere woorden, als we schrijven:

```
EenType x;
x = new AnderType();
```

dan is dit enkel maar toegestaan als `AnderType` een subklasse is van `EenType`. Bijvoorbeeld, dit mag:

```
Rechthoek r = new Vierkant(5); //Een vierkant is een rechthoek
```

maar dit mag niet:

```
String r = new Docent(); //Een docent is geen string
```

In de vorige sectie beweerden we dat, als meerdere klassen een methode met dezelfde naam definiëren, het altijd de *meest specifieke* methode die van toepassing is, die zal worden uitgevoerd. In het licht van bovenstaande discussie kunnen we ons nu afvragen of er hiervoor eigenlijk gekeken wordt naar het statische type van de variabele of naar het dynamische type van het object. Als je hierover nadenkt, zal je tot het besluit komen dat de tweede optie de beste is. Dit is gelukkig dan ook precies wat Java zal doen:

```
1 Rechthoek r;
2 r = new Vierkant(5);
3 r.draai(); //De methode uit klasse Vierkant wordt uitgevoerd
4 r = new Rechthoek(5,5);
5 r.draai(); //De methode uit klasse Rechthoek wordt uitgevoerd
```

Let wel, dit geldt enkel maar voor het object waarop de methode wordt uitgevoerd en (helaas?) niet voor de argumenten van een methode. Als we bijvoorbeeld dit schrijven:

```
1 class Rechthoek {
2     Double verschilInOmtrekMet(Rechthoek r) {
3       return Math.abs(omtrek() - r.omtrek()); // Versie 1
```

```
4       }
5       Double verschilInOppervlakteMet(Vierkant v) {
6         return Math.abs(omtrek() - v.omtrek()); // Versie 2
7       }
8   }
```

dan krijgen we dit:

```
1  Rechthoek r;
2  Rechthoek ander = new Vierkant(5);
3  r.verschilInOmtrekMet(ander); // Versie 1 wordt uitgevoerd.
```

Hier wordt dus de methode met als argument een `Rechthoek` (het statische type van `r`) uitgevoerd, ook al zit er op dat moment eigenlijk een `Vierkant` (het dynamische type van `r`) in deze variabele. Niettemin is het natuurlijk wel zo dat de methode "`r.omtrek()`" die versie 1 zal oproepen de methode uit de klasse `Vierkant` zijn.

3.3.5 Typecasting

Soms hebben we als progammeur meer informatie over de inhoud van een bepaalde variabele dan de compiler heeft. Een voorbeeld:

```
1  public class Werknemer {
2    private Werknemer baas;
3    public Werknemer getBaas(){
4      return baas;
5    }
6    public Werknemer(Werknemer b) {
7      baas = b;
8    }
9    ... //Nog wat dingen over werknemers
10 }
11 public class Docent extends Werknemer {
12   private Vak[] vakken;
13   public Vak[] getVakken() {
14     return vakken;
15   }
16   public Docent(Werknemer b) {
17     super (b);
18   }
19   ... //Nog wat dingen over docenten
20 }
```

De klasse `Werknemer` heeft een algemene methode `getBaas` die de dienstoverste van een werknemer teruggeeft. De klasse `Docent` breidt de klasse werknemer uit met informatie over de vakken die de docent geeft. Veronderstel nu verder dat we weten dat het organigram van de school in kwestie zo is opgebouwd dat de baas van een docent altijd zelf ook een docent is[1]. In dit geval kan het voorkomen dat we bijvoorbeeld geïnteresseerd zijn in de vakken die gegeven worden door de baas van een docent:

```
1  Docent hcr = new Docent();
2  Docent jve = new Docent(hcr);
3  jve.getBaas().getVakken();
```

De compiler zal dit echter niet aanvaarden:

[1] Een goede programmeur zou hier natuurlijk niet blind op vertrouwen, maar dit in zijn programma ook controleren.

```
$ javac Docent.java
Docent.java:17: cannot find symbol
symbol  : method getVakken()
location: class Werknemer
        jve.getBaas().getVakken();
                      ^
1 error
```

De reden hiervoor is eenvoudig: de compiler weet enkel maar dat de methode `getBaas` een `Werknemer` object zal teruggeven, maar niet dat de baas van een docent ook een docent is. We kunnen dit hem wel vertellen door middel van een *typecast*, als volgt:

```
((Docent) jve.getBaas()).getVakken();
```

Met een typecast omzeil je dus de stikte controle van de compiler op juiste typering. Als je dit doet, moet je natuurlijk wel zeker zijn van je zaak: als `jve.baas()` om één of andere reden toch een werknemer zou teruggeven die geen docent is, dan zal er een *runtime* fout optreden.

3.3.6 instanceof

Soms kan het handig zijn om te testen of een bepaald object tot een bepaalde klasse behoort. Bijvoorbeeld, als je een typecase wilt doen, maar niet 100% zeker bent dat het object wel degelijk behoort tot de klasse in kwestie, kan het nuttig zijn om dit toch maar eens te testen. Je kan dit doen met het sleutelwoord `instanceof`.

```
1 class Test {
2   public static void main(String[] args) {
3      Werknemer jve = new Docent();
4      if (jve instanceof Docent)
5          System.out.println("Ik heb een docent gevonden.");
6   }
7 }
```

3.4 Encapsulatie

Een belangrijk voordeel van het objectgeöriënteerd werken is dat het ons toelaat om irrelevante gegevens af te schermen. Om dit te illustreren kunnen we eens nadenken over de implementatie van een `Datum` klasse[2]. Er zijn verschillende manieren om een datum voor te stellen. Dit is één van de meest voor de hand liggende:

```
1  class Datum {
2
3    int dag;
4    int maand;
5    int jaar;
6
7    public Datum(int d, int m, int j) {
8        dag = d;
9        maand = m;
10       jaar = j;
11   }
12
13 }
```

[2]Er bestaat natuurlijk een standaard implementatie, namelijk `java.util.Date`, maar deze negeren we nu even.

Hier voorzien we drie `int`s (dwz. 3×32 bytes) om één datum voor te stellen. Op hedendaagse hardware kunnen we ons dit gemakkelijk veroorloven, maar het is wel een beetje kwistig. In principe zou één enkele `int` al ruimschoots volstaan, aangezien er sinds het begin van onze jaartelling nog niet eens 750.000 ($\ll 2^{32}$) dagen verstreken zijn. We zouden dus ook iets als dit kunnen doen:

```
1  class Datum{
2
3    /** Aantal dagen versteken sinds 01/01/1980
4     **  Mag negatief zijn voor vroegere datums **/
5    int sindsEenJanuari1980;
6
7    public Datum(int verstreken) {
8      sindsEenJanuari1980 = verstreken;
9    }
10
11 }
```

Elk van deze twee voorstellingswijzen heeft zijn voor- en zijn nadelen. Veronderstel dat we een programma aan het schrijven zijn dat twee functionaliteiten verwacht van een `Datum` object: het moet zijn jaartal kunnen teruggeven en het moet een nieuw `Datum` object kunnen teruggeven dat één dag vroeger voorstelt. Met de eerste voorstellingswijze:

```
1  /** Datum klasse - versie 1 **/
2  class Datum {
3
4    int dag;
5    int maand;
6    int jaar;
7
8    public Datum(int d, int m, int j) {
9        dag = d;
10       maand = m;
11       jaar = j;
12   }
13
14   public int jaar() {
15     return jaar;
16   }
17
18   public Datum eenDagVroeger() {
19     Datum gisteren = new Datum();
20     if (vandaag.dag > 1) {
21         gisteren.dag = vandaag.dag - 1;
22         gisteren.maand = vandaag.maand;
23         gisteren.jaar = vandaag.jaar;
24     }
25     else {
26         if (vandaag.maand > 1) { // 1e dag van maand
27             gisteren.maand = vandaag.maand - 1;
28             gisteren.jaar = vandaag.jaar;
29             gisteren.dag = laatsteDag(gisteren.maand);
30         }
31         else { // Nieuwjaar
32             gisteren.maand = 12;
33             gisteren.jaar = vandaag.jaar - 1;
```

```
34              gisteren.dag = 31;
35          }
36      }
37      return gisteren;
38    }
39  }
```

Met de tweede voorstellingswijze zou dit worden:

```
1  /** Datum klasse - versie 2 **/
2  public Datum {
3
4      int sindsEenJanuari1980;
5
6      public Datum eenDagVroeger() {
7          Datum gisteren = new Datum();
8          gisteren.sindsEenJanuari1980 = this.sindsEenJanuari1980 - 1;
9          return gisteren;
10     }
11
12     boolean isSchrikkeljaar(int j) {
13       return j % 4 == 0 && (j % 100 != 0 || j % 400 == 0);
14     }
15
16     /** Code van Microsoft
17      **  Niet uitvoeren op de laatste dag van een schrikkeljaar!
18      **  Zie ook http://bit-player.org/2009/the-zune-bug
19      **/
20     public int jaar() {
21       int jaar = 1980;
22       int dagen = sindsEenJanuari1980
23       while (dagen > 365) {
24          if (isSchrikkeljaar(jaar)) {
25            if (dagen > 366) {
26              dagen -= 366;
27              jaar += 1;
28            }
29          }
30          else {
31             dagen -= 365;
32             jaar += 1;
33          }
34       }
35     }
36 }
```

Een belangrijk voordeel van obectoriëntatie is dat we *alle* code die afhankelijk is van hoe we een datum juist voorstellen nu op één plaats bij elkaar hebben. Als we dus bijvoorbeeld oorspronkelijk gekozen hebben voor de eerste voorstelling, kunnen we later – als om één of andere reden deze keuze toch niet ideaal blijkt te zijn – nog altijd gemakkelijk veranderen. Aangezien alles wat van deze voorstellingswijze afhangt toch netjes bij elkaar staat, hoeven we hiervoor immers enkel maar de eerste definitie van de `Datum` klasse te vervangen door de tweede. We besparen ons met andere woorden de tijdrovende en foutgevoelige taak van een heel programma te moeten daarlopen op zoek naar plaatsen waar we de attributen `jaar, maand` en `jaar` gebruikt hebben.

Dat klinkt allemaal goed, maar laat ons toch niet te snel zijn. Is het allemaal echt wel zo

eenvoudig als we het hier laten uitschijnen? Stel je even voor dat we meewerken aan de implementatie van een groot software-pakket, en dat één van onze verantwoordelijkheden de `Datum`-klasse is. We hebben deze klasse al lang geleden geïmplementeerd – zo moeilijk is ze tenslotte niet – en de andere programmeurs van het project gebruiken ze vaak in hun code. Op zekere dag krijgen we echter de opdracht om iets te veranderen aan onze code – bijvoorbeeld, we moeten een methode `eenDagVroeger` bijmaken, die er oorspronkelijk nog niet was.

Na hier even over te hebben nagedacht, komen we tot het besluit dat dit veel eenvoudiger zou zijn, moesten we een datum niet voorstellen als dag/maand/jaar, maar dmv. het aantal dagen dat vertreken is sinds 1/1/1980. We zouden dus graag onze oorspronkelijke implementatie van `Datum` (versie 1) vervangen door een nieuwe implementatie (versie 2). Het probleem is echter dat we er zelf totaal geen idee van hebben waar of hoe onze `Datum` klasse gebruikt wordt in de code van de andere programmeurs. Aangezien we natuurlijk niet willen dat onze wijziging hun code kapot zou maken, mogen we niet ondoordacht te werk gaan.

Onze nieuwe implementatie verandert de voorstelling van de datums, maar zorgt er tegelijkertijd voor dat de methodes `jaar()` en `eenDagVroeger()` nog steeds hetzelfde resultaat produceren als vroeger. Moesten we er dus zeker van kunnen zijn dat de andere programmeurs enkel maar gebruik gemaakt hebben van deze methodes – en dat ze nergens rechtstreeks iets gedaan hebben met de variabelen `dag`, `maand` en `jaar` – dan zouden we ook zeker weten dat onze wijziging in orde is. Maar hoe kunnen we nu van zoiets ooit zeker zijn?

Het antwoord wordt ons geboden door de *toeganscontrole* van Java. We kunnen in de definitie van een klasse aangeven welke variabelen/methodes toegankelijk zijn voor anderen en welke niet. Dit gebeurt door middel van een aantal sleutelwoorden zoals `public` en `private`. Het sleutelwoord `public` betekent dat de methode of het attribuut voor iedereen toegankelijk is, en `private` betekent dat de methode of het attribuut enkel maar binnen de klasse zelf gebruikt mag worden. In een goed Java-programma zal het overgrote deel van de attributen als `private` geklasseerd staan, omdat dit de programmeur de vrijheid geeft om later zijn implementatiekeuzes nog te kunnen veranderen, zonder hiermee andere code kapot te maken.

Een betere versie van onze `Datum` zou dan ook zijn:

```
1  class Datum {
2
3    private int dag;
4    private int maand;
5    private int jaar;
6
7    public Datum(int d, int m, int j) {
8        dag = d;
9        maand = m;
10       jaar = j;
11   }
12   public int jaar() {
13     return jaar;
14   }
15   public Datum eenDagVroeger() {
16     ... //Zoals hierboven
17   }
18 }
```

Als we `Datum` op deze manier implementeren, is het *enige* wat er in andere klassen met `Datum` objecten mag gebeuren:

```
1 Datum d = new Datum(27,09,1980);
2 d.jaar();
3 d.eenDagVroeger();
```

Iets als dit:

```
1 Datum d = new Datum(27,09,1980);
2 if (d.maand == 9)
3     System.out.println("september");
```

mag dus niet meer voorkomen in een andere klasse, en zal door de compiler geweigerd worden:

```
$ javac Test.java
Test.java:3: maand has private access in Datum
         if (d.maand  == 9)
              ^
1 error
```

We kunnen nu met een gerust hart de bovenstaande klasse `Datum` vervangen door deze:

```
1  class Datum {
2
3    private int sindsEenJanuari1980;
4
5    private int[] geenSchrikkel = { 31, 28, 31, 30,
6                                    31, 30, 31, 31, 30, 31, 30, 31 };
7
8    private int dagenPerMaand(int maand, int jaar) {
9      int resultaat;
10     if (isSchikkeljaar(jaar) && maand == 2)
11         resultaat = 29;
12     else
13         resultaat = geenSchrikkel[maand - 1];
14     return resultaat;
15   }
16
17   public Datum(int dag, int maand, int jaar) {
18     int verstreken = 0;
19     if (jaar >= 1980)
20         for (int j = 1980; j < jaar; j++)
21             for (int m = 1; m <= 12; m++)
22                 verstreken += dagenPerMaand(m,j);
23     else
24        for (int j = 1980; j >= jaar; j--)
25             for (int m = 1; m <= 12; m++)
26                 verstreken -= dagenPerMaand(m,j);
27     for (int m = 1; m <= maand; m++)
28            verstreken += dagenPerMaand(m,j);
29     sindsEenJanuari = verstreken + dag;
30   }
31   public int jaar() {
32     ... //Zoals hierboven
33   }
34   public Datum eenDagVroeger() {
35     SindsDatum gisteren = new SindsDatum();
36     gisteren.sindsEenJanuari1980 = this.sindsEenJanuari1980 - 1;
37     return gisteren;
38   }
39 }
```

Wat betreft het gedrag van alle `public` methodes, inclusief de constructor, is deze klasse identiek aan de dag/maand/jaar-versie, wat dus betekent dat een andere klasse het verschil tussen de twee versies van `Datum` nooit kán merken.

Elke klasse in Java heeft dus eigenlijk twee verschillende gezichten. *Intern* zijn alle implementatiedetails van de klasse te zien, maar *extern* zijn enkel de publieke methodes en variabelen zichtbaar. Dit geeft de programmeur een grote vrijheid: zolang het gedrag van de externe methodes en variabelen hetzelfde blijft, mag hij aan de interne aspecten veranderen wat hij wilt.

Een bijkomend voordeel is dat een programmeur op deze manier de correcte werking van een goed ontworpen klasse kan *garanderen*. Dit wil zeggen, hij is staat om na te gaan dat *gelijk wat* iemand anders met zijn klasse doet, deze zich altijd zo zal gedragen als hij in zijn eigen documentatie beweert. Bij een goed ontworpen klasse heeft is hijzelf immers de enige die genoeg toegang heeft tot deze klasse om dit gedrag te bepalen.

Omdat toegansbeperkingen zo nuttig zijn, zal praktisch elk Java programma hier uitvoerig gebruik van maken. Dit heeft ondere andere geleid tot het idioom van "getters" en "setters": in plaats van bijvoorbeeld dit te schrijven:

```
1  public class Datum {
2    public int jaar;
3    ...
4  }
```

schrijft men dan:

```
1  public class Datum {
2    private int jaar;
3    public int getJaar() {
4      return jaar;
5    }
6    public void setJaar(int j) {
7      jaar = j;
8    }
9    ...
10 }
```

Het voordeel van deze aanpak is weer hetzelfde: als men later besluit om de variabale `jaar` te verwijderen of op een andere manier voor te stellen, dan kan men gewoon de "getter" `getJaar()` en de "setter" `setJaar(int j)` aanpassen. Door de "setter" van een variabele `private` te maken (of gewoon weg te laten), kan een variabele "read-only" gemaakt worden.

Hoewel `public` en `private` de meest voorkomende toegangsbeperkingen zijn, zijn er nog twee andere mogelijkheden. Men kan het sleutelwoord `protected` gebruiken of gewoon geen sleutelwoord schrijven. De vier mogelijkheden zijn, gerangschikt van los naar streng:

sleutelwoord	welke klassen hebben toegang
public	alle klassen
protected	klassen uit hetzelfde *pakket* (zie later)
	en bovendien subklassen van de huidige klasse
(geen sleutelwoord)	klassen uit hetzelfde pakket
private	alleen de klasse zelf

Het verdient aanbeveling om het toegangsniveau altijd *zo streng mogelijk* te kiezen.

Ook klassen zelf kunnen trouwens ofwel publiek toegankelijk ofwel pakkettoegankelijk zijn (de andere twee mogelijkheden zijn voor een klasse niet erg zinvol). Je schrijft dan ofwel:

```
public class Bla { ... }
```

ofwel:

```
class Bla { ... }
```

3.5 Statische methodes en variabelen

Normaalgezien horen variabelen en methodes bij een *object*. Als we bijvoorbeeld schrijven:

```
1  public class Vector {
2    private int x;
3    private int y;
4    public int getX() {
5      return x;
6    }
7    public int getY() {
8      return y;
9    }
10   public Vector(int i, int j) {
11     x = i;
12     y = j;
13   }
14   public void telOp(Vector v) {
15     x += v.getX();
16     y += v.getY();
17   }
18 }
```

Dan heeft elke `Vector` v zijn eigen x-coördinaat $v.x$ en zijn eigen y-coördinaat $v.y$. Een methode zoals `getX()` moet altijd worden toegepast op een bepaald `Vector` object en zal dan toegang hebben tot de coördinaten van dát object.

Soms zijn er echter methodes of variabelen die wel thuishoren bij een *klasse*, maar niet bij een individueel object van deze klasse. Veronderstel dat we bijvoorbeeld een methode `nulVector()` willen schrijven die een nulvector $(0, 0)$ construeert. De enige logische plaats om een dergelijke methode te definiëren is in de klasse `Vector`, maar deze methode hoort niet bij een bepaalde vector. Als we gewoon zouden schrijven:

```
1  public class Vector {
2    ...
3    public Vector nulVector() {
4      return new Vector(0,0);
5    }
6  }
```

dan zouden we deze methode op deze manier moeten oproepen:

```
1  (new Vector(1,1)).nulVector();
```

We zijn hier dus gedwongen om één of andere vector aan te maken, gewoon om de methode `nulVector()` op deze vector te kunnen oproepen *hoewel deze methode hier helemaal niets mee doet*. Dit is vanzelfsprekend niet erg elegant.

Om deze reden voorziet Java ook *statische* methodes en variabelen. Deze worden aangeduid met het sleutelwoord `static` en horen niet bij een individueel object, maar bij een klasse in het algemeen. Bijvoorbeeld:

```
1  public class Vector {
2    ...
3    public static Vector nulVector() {
4      return new Vector(0,0);
5    }
6  }
```

Deze methode kan nu worden opgeroepen op de klasse `Vector` zelf:

```
Vector.nulVector();
```

We zijn trouwens in de loop van deze cursus al een aantal statische methodes en variabelen tegen het lijf gelopen:

- De `main` methode van een programma moet statisch zijn. (Op het moment dat het programma gestart wordt, zijn er immers nog geen objecten aangemaakt waarop methodes zouden kunnen worden opgeroepen.)
- De standaard uitvoer `System.out` is een statische variabele in de klasse `System` van het type `PrintWriter`.

3.6 Abstracte klassen en interfaces

Een *abstracte* klasse is herkenbaar aan het sleutelwoord **abstract** dat voorafgaat aan de definitie van de klasse.

```
public abstract class MijnKlasse
{
    // methoden en attributen...
}
```

Een abstracte klasse is niet rechtstreeks instantieerbaar. Dat wil zeggen dat objecten van dat type niet kunnen bestaan, tenzij ze tot een (niet-abstracte) subklasse behoren. De klasse MijnKlasse hierboven heeft dus slechts zin als er subklassen voor gecreëerd worden, bijvoorbeeld

```
public class MijnAndereKlasse extends MijnKlasse
{
   // methoden en attributen...
}
```

De volgende code geeft dus aanleiding tot een compilerfout.

```
MijnKlasse m = new MijnKlasse();
```

Let wel: de *declaratie* van een veranderlijke van het type MijnKlasse is niet verboden, het is de rechtstreekse constructor-aanroep die voor problemen zorgt. De volgende code is dus wél aanvaardbaar.

```
MijnKlasse m = new MijnAndereKlasse();
```

Een abstracte klasse kan *abstracte methoden* bevatten. Een abstracte methode wordt voorafgegaan door het woord **abstract**. Ze bevat geen implementatie (geen akkoladen), maar alleen een specificatie van de toegangsbeperking (`public, protected`, niets, of `private`), een resultaattype (of **void**), een naam, een lijst van parameters en eventuele exceptions (zie later). Als een niet-abstracte klasse afstamt van een abstracte klasse (als dochter, of kleindochter, of nog verder), dan moet de niet-abstracte klasse *alle* abstracte methoden herdefiniëren.

Een interessant voorbeeld van een abstracte klasse is de volgende:

```
1  public abstract class Schaakstuk {
2
3    private boolean wit;
4
5    public Schaakstuk(boolean w) {
6      wit = w;
7    }
8
9    public boolean isWit() {
10     return wit;
```

```
11    }
12
13    private SchaakbordVakje positie;
14
15    public SchaakbordVakje getPositie() {
16      return positie;
17    }
18
19    public void verplaats(SchaakbordVakje nieuw)
20                                throws IllegalArgumentException {
21      if (magNaar(nieuw))
22          positie = nieuw;
23      else // de zet is niet toegestaan
24          throw new IllegalArgumentException("Dit is geen
25                        toegestane beweging van het schaakstuk!");
26    }
27
28    public abstract boolean magNaar(SchaakbordVakje nieuw);
29
30  }
```

Deze klasse stelt een schaakstuk voor dat een bepaalde kleur heeft en een bepaalde positie. Er is ook een methode voorzien die probeert om het stuk te verplaatsen naar een nieuwe positie op het spelbord. Deze methode controleert eerst of de voorgestelde zet wel legaal is en gooit een uitzondering (zie verderop in deze cursus) indien dit niet zo is. Deze methode maakt gebruik van een *abstracte* methode `magNaar`. De reden waarom deze laatste methode abstract gelaten is, ligt voor de hand: enkel een specifieke subklasse van `Schaakstuk` kan beslissen welke zetten voor dat specifieke soort van schaakstuk legaal zijn. Bijvoorbeeld:

```
1  public class Loper extends Schaakstuk {
2     public Loper(boolean w) {
3       super (w);
4     }
5     public boolean magNaar(SchaakbordVakje nieuw) {
6        boolean voorwaarts =
7            positie.getRij() - nieuw.getRij()
8           == positie.getKolom() - nieuw.getKolom();
9        boolean achterwaarts =
10           positie.getRij() - nieuw.getRij()
11          == nieuw.getKolom() - positie.getKolom();
12       boolean verschillend =
13         nieuw.getRij() != positie.getRij()
14         || nieuw.getKolom() != positie.getKolom();
15       return verschillend && (voorwaarts || achterwaarts);
16    }
17 }
```

In dit voorbeeld heeft het dus inderdaad geen zin om de klasse `Schaakstuk` zelf te instantiëren, maar natuurlijk wel zijn subklassen:

```
1 // Dit mag niet: Schaakstuk s = new Schaakstuk(true);
2 Schaakstuk[] stukken = new Schaakstuk[32];
3 Schaakstuk[0] = new Loper(true); //Mag wel
4 Schaakstuk[1] = new Loper(true);
5 // Enzovoort
```

Java is een streng getypeerde taal. Dit heeft belangrijke voordelen: het laat de compiler toe om een groot aantal fouten al tijdens de compilatiestap te detecteren. Met enige zin voor overdrijving, zegt men van streng getypeerde talen soms wel eens dat als een programma al compileert, de kans groot is dat het ook wel zal werken.

Een nadeel van een streng typesysteem is echter dat het soms het hergebruik van code in de weg lijkt te staan. Een typisch voorbeeld is de implementatie van een sorteertalgoritme. Zoals we in het eerste hoofdstuk gezien hebben, kan een sorteeralgortime als `Bubblesort` tamelijk veel implementatiewerk vragen. Stel je nu eens voor dat we rechthoeken willen sorteren volgens grootte. We zouden dan kunnen schrijven:

```
1  public class Rechthoek {
2    public static Rechthoek[] sorteer(Rechthoek[] rij) {
3       //Een sorteeralgoritme
4    }
5    ...
6  }
```

Maar wat als we daarna werknemers zouden willen gaan sorteren volgens leeftijd? Hiervoor zouden we vanzelfsprekend bovenstaande methode niet kunnen gebruiken, aangezien `Werknemer` geen subklasse is van `Rechthoek`. In plaats daarvan zouden we een methode

```
1  public static Werknemer[] sorteer(Werknemer[] rij) {
2    // Een kopie van het sorteeralgoritme
3  }
```

moeten schrijven. Dit is natuurlijk niet erg ideaal. We willen niet dat we voor elk soort van object dat we willen sorteren een nieuwe methode moeten schrijven.

Abstracte klassen zouden hiervoor een oplossing kunnen bieden. Bijvoorbeeld:

```
1  abstract class Sorteerbaar {
2
3      /** Vult "uitvoer" met de elementen van "invoer",
4       **  maar dan gesorteerd. Hiervoor wordt telkens het
5       **  minimum van de nog niet geplaatste elementen vooraan
6       **  gezet.  (Deze functie zou ook
7       **  "static Sorteerbaar[] sorteer(Sorteerbaar[] invoer)"
8       **  kunnen zijn, maar om zelf een nieuwe array
9       **  te kunnen aanmaken, moeten we weten welke
10      **  subklasse van Sorteerbaar er juist in "invoer"
11      **  zit, en dit wordt in deze cursus niet behandeld.)
12      **/
13     public static void sorteer(Sorteerbaar[] invoer,
14                                Sorteerbaar[] uitvoer) {
15         for (int i = 0; i < invoer.length; i++) {
16           int min = indexVanMinimum(invoer);
17           uitvoer[i] = invoer[min];
18           invoer[min] = null;
19         }
20     }
21
22     /** Zoekt de index van het minimum van een rij,
23      **  waarin mogelijk "null"s kunnen voorkomen --
24      **  deze worden genegeerd.
25      **/
26     public static int indexVanMinimum(Sorteerbaar[] rij) {
27         int start = 0;
```

```
28          // Nulls aan het begin van de lijst negeren
29          while (rij[start] == null) {
30               start++;
31           }
32          Sorteerbaar minimum = rij[start];
33          int indexVanMinimum = start;
34          for (int i = start; i < rij.length; i++) {
35              // De evaluatie van X && Y stopt zonder Y
36              // uit te voeren als X == false
37              if (rij[i] != null && rij[i].kleinerDan(minimum)) {
38                  minimum = rij[i];
39                  indexVanMinimum = i;
40              }
41          }
42          return indexVanMinimum;
43      }
44
45      public abstract boolean kleinerDan(Sorteerbaar x);
46
47  }
48
49  /** Een klasse van getallen die omgekeerd
50   **  gesorteerd worden.
51   **/
52  class Omgekeerd extends Sorteerbaar {
53      private int waarde;
54      public Omgekeerd(int w) {
55          waarde = w;
56      }
57      public boolean kleinerDan(Sorteerbaar ander) {
58          return waarde > ((Omgekeerd) ander).waarde;
59      }
60      public static void main(String[] args) {
61          Omgekeerd a = new Omgekeerd(15);
62          Omgekeerd b = new Omgekeerd(23);
63          Omgekeerd c = new Omgekeerd(7);
64          Omgekeerd d = new Omgekeerd(42);
65          Omgekeerd[] rij = {a,b,c,d};
66          Omgekeerd[] gesorteerd = new Omgekeerd[rij.length];
67          Sorteerbaar.sorteer(rij, gesorteerd);
68          for (int i = 0; i < gesorteerd.length; i++) {
69              System.out.println(gesorteerd[i].waarde);
70          }
71      }
72  }
```

Deze aanpak heeft inderdaad het gewenste effect, maar er is één groot nadeel: in Java kan een klasse maar ten hoogste van één andere klasse overerven! Iets als

```
1  class Omgekeerd extends Sorteerbaar, Integer { //Fout!
2    ...
3  }
```

is dus *niet* toegestaan. Hiervoor zijn er trouwens goede redenen, aangezien objectgeöriënteerde talen waarin meervoudige overerving wel is toegstaan vaak nogal ingewikkeld worden.

Om deze reden biedt Java een alternatief aan met de notie van een *interface*. Een interface is eigenlijk niets meer dan een een abstracte klasse die "zuiver abstract" is, d.w.z. dat ze geen attributen noch niet-abstracte methoden bevat. Met andere woorden, een interface beschrijft enkel maar een aantal methodes die door een klasse geïmplementeerd moeten worden. Bijvoorbeeld:

```
1 public interface Sorteerbaar {
2       public boolean kleinerDan(Sorteerbaar x);
3 }
```

Merk op dat het sleutelwoord `abstract` hier nu niet meer gebruikt wordt, aangezien een interface toch enkel maar abstracte methodes mag bevatten. Waar een abstracte klasse uitgebreid kan worden (`extends`), kan een interface geïmplementeerd (`implements`) worden. Dit betekent niets anders dan dat de implementerende klasse al de methodes die in de interface staan, moet aanbieden. Bijvoorbeeld:

```
1 public class Omgekeerd implements Sorteerbaar {
2     public boolean kleinerDan(Sorteerbaar ander) {
3         return waarde > ((Omgekeerd) ander).waarde;
4     }
5     ...
6 }
```

In tegenstelling tot overerving, mag een klasse zoveel interfaces implementeren als ze wil, en daarnaast ook nog overerven van een andere klasse. Iets als dit mag dus wel:

```
1 public class Omgekeerd extends Integer implements Sorteerbaar,
2                                                    Serializable {
3     ...
4 }
```

In ons voorbeeld is het dus een betere oplossing om van `Sorteerbaar` een interface te maken ipv. een abstracte klasse. Dit betekent dan natuurlijk wel dat onze sorteermethode zelf zich niet langer in de interface `Sorteerbaar` mag bevinden:

```
1 public interface Sorteerbaar {
2       public boolean kleinerDan(Sorteerbaar x);
3 }
4 public class SorteerAlgoritme {
5     public static void sorteer(Sorteerbaar[] invoer,
6                                Sorteerbaar[] uitvoer) {
7       ...  // Een implementatie van een sorteeralgoritme
8     }
9 }
10 public class Omgekeerd implements Sorteerbaar {
11     public boolean kleinerDan(Sorteerbaar ander) {
12         return waarde > ((Omgekeerd) ander).waarde;
13     }
14     public static void main(String[] args) {
15         ... // Initializatie van a,b,c,d
16         Omgekeerd[] rij = {a,b,c,d};
17         Omgekeerd[] gesorteerd = new Omgekeerd[rij.length];
18         SorteerAlgoritme.sorteer(rij, gesorteerd);
19     }
20     ... // De rest van de klasse Omgekeerd
21 }
```

Tot slot van deze sectie, vatten we nog even de voornaamste verschilpunten tussen abstracte klassen en interfaces samen:

- Een abstracte klasse mag variabelen of implementaties van methoden bevatten, en deze worden dan gewoon overgeërfd door subklassen. Een interface legt enkel een aantal voorwaarden op aan klassen die de interface willen implementeren: een implementerende klasse belooft in wezen gewoon dat ze al de methodes die de interface vermeldt zal aanbieden (en deze belofte wordt door de compiler gecontroleerd).
- Een klasse mag slechts van één klasse overerven, maar een willekeurig aantal interfaces implementeren.

Wanneer maak je nu best gebruik van een abstracte klasse (**class** X **extends** Y) en wanneer van een interfaces (**class** X **implements** Y) ? Een goede vuistregel is: als je de klasse X in een enkele zin zou beschrijven, zou je dan de klasse Y vermelden of niet? Indien ja, dan kan je overwegen om een abstracte klasse te maken van Y; anders is het waarschijnlijk best een interface. Bij een loper zou je inderdaad altijd vermelden dat het een schaakstuk is, terwijl je bij bv. een rechthoek niet snel zou vermelden dat een rechthoek iets is dat volgens grootte gesorteerd kan worden.

3.7 Inwendige klassen

Een **inwendige klasse** (*inner class*) is een klasse gedefinieerd binnen een andere klasse. Ze heeft toegang tot de eigenschappen en methodes van de omsluitende klasse, zelfs indien deze **private** zijn. Inwendige klassen worden voornamelijk gebruikt voor data structuren, hulpklassen, en *event handlers* (zie later). Een klein voorbeeld waarin een inwendige klasse als data structuur gebruikt wordt:

```
1  public class Groep
2  {
3     class Person
4     {
5          // inner class defines the required structure
6          String first;
7          String last;
8     }
9
10    Person personArray[] = {new Person(), new Person(), new Person()};
11 }
```

In de *outer class* of uitwendige klasse wordt een array van person objecten gecreëerd met specifieke eigenschappen. Deze objecten kunnen bijvoorbeeld aangesproken worden met `personArray[1].last`.

Objecten van de inwendige klasse kunnen alleen maar bestaan binnen een instantie van de omvattende klasse, in het voorbeeld de klasse Groep.

In sommige situaties hoeft een inwendige klasse geen naam te hebben. We spreken dan van een anonieme klasse. Voorbeelden worden gegeven in het hoofdstuk over grafische userinterfaces.

3.8 Naamgevingsconventies

Het zal je misschien al zijn opgevallen dat deze cursus bepaalde naamgevingsconventies aanhoudt. Deze zijn tamelijk wijdverspreid doorheen de Java gemeenschap en kunnen er dan ook voor zorgen dat je code gemakkelijker te begrijpen valt door andere programmeurs (of examinators). We zetten deze nog even kort op een rijtje.

Namen die bestaan uit meerdere woorden worden geschreven in zogenaamde *camel case*: alles wordt aan elkaar geschreven, maar elk nieuw woord begint wel met een hoofdletters. Dit ziet er dan bijvoorbeeld zo uit: `elkWoordBegintMetEenHoofdletter`.

Voor de allereerste letter van de naam geldt:

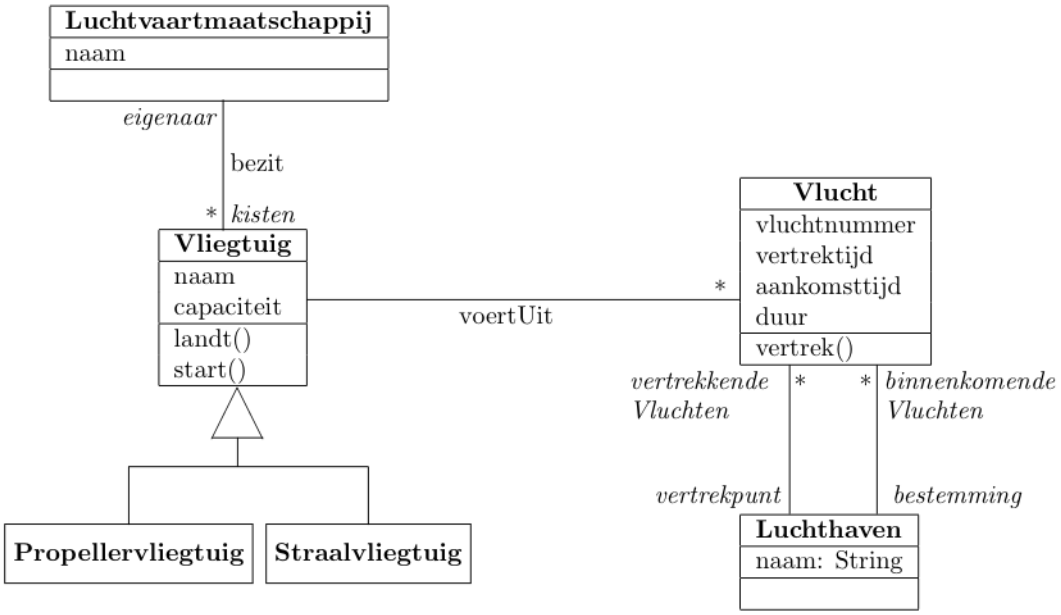
- Als de naam een *klasse* of *interface* aanduidt, begint hij met een hoofdletter.

- Als de naam daarentegen verwijst naar een *variabele* of een *attribuut* (hetzij van een objecttype, hetzij van een primitief type) dan begint hij met een kleine letter.

3.9 Klassediagramma

OO is meer dan een programmeertaal. OO is een paradigma: een manier om naar de wereld en naar softwaresystemen te kijken en er over na te denken, en een methode om de ontwikkeling van ingewikkelde systemen aan te pakken. Slechts een beperkt deel van de mentale inspanning die nodig is om een software systeem te ontwikkelen, wordt effectief gespendeerd aan het schrijven van code. Andere belangrijke taken zijn een grondige *analyse* van de vereisten van het systeem en het domein waarin het moet werken, en natuurlijk het *ontwerp* van de structuur van het systeem.

Naast de gekende hulpmiddelen voor programmeerwerk, zijn er natuurlijk ook een aantal technieken ontwikkeld om de software-ontwikkelaar te helpen bij analyse en ontwerp. In OO is het belangrijkste van deze het *klassediagramma*. Zoals de naam al aangeeft, is dit een grafische voorstelling van de klassen – dwz. soorten van objecten – die een rol spelen in het systeem, en van de relatie tussen hen. Verderop in dit hoofdstuk bespreken we in detail hoe zo'n diagramma eruit ziet, maar dit is alvast een voorbeeld:



Er zijn twee verschillende soorten klassediagramma's, die in verschillende fases van het softwareontwikkelingsproces een rol spelen:

- In het begin van dit proces, tijdens de *analyse*-fase, wordt gebruik gemaakt van een klassediagramma dat de structuur van het *domein* beschrijft;
- Later, tijdens de *ontwerp*-fase, kan een tweede klassediagramma gemaakt worden, dat de structuur van het *programma* in detail beschrijft.

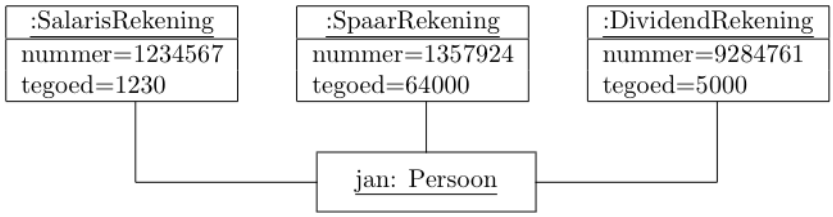
Bijvoorbeeld, veronderstel dat we een systeem moeten implementeren voor het beheer van een openbare bibliotheek. In de analyse-fase zullen we dan het domein in kwestie beschrijven met een klassediagram dat spreekt over boeken, klanten, leningen, enzovoort. Ruwweg zou dit diagramma dus ongeveer een klasse bevatten voor elk zelfstandig naamwoord dat een bibliothecaris zou gebruik om uit te leggen wat het systeem zou moeten doen.

Op basis van dit domein-klassediagramma, kunnen we dan later een klassediagramma van het programma maken. Hierin vinden we dan ook klassen terug die ons programma nodig heeft,

maar die een bibliothecaris niets zouden zeggen, zoals bijvoorbeeld een klasse `DBManager` die ons programma verbindt met een databank, of een klasse `UserInterface` die zorgt voor de interactie met de gebruiker.

Naast klassediagramma's, bestaan er ook nog *objectdiagramma's*. Het verschil mag duidelijk zijn: in een klassediagramma worden de relaties tussen verschillende *soorten* van objecten in het algemeen aangegeven, terwijl een objectdiagramma relaties tussen verschillende individuele objecten weergeeft.

Een objectdiagram is een afbeelding *op een bepaald tijdstip* van objecten die gecreëerd zijn volgens de structuur in het klassediagram. Dit diagram wordt zelden gebruikt, bijvoorbeeld wel als voorbeeld of uitleg bij een wat minder gemakkelijk te begrijpen klassediagram. Dit is een voorbeeld:



We bespreken nu in wat meer detail de verschillende begrippen en hoe ze in klasse- of objectdiagramma's kunnen voorkomen.

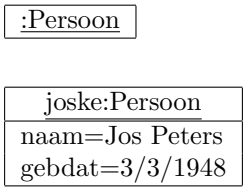
3.9.1 Object

Een *object* is iets dat een zelfstandig bestaan leidt. Het is een afspiegeling van een fysiek ding uit de werkelijkheid of van een concept uit die werkelijkheid. Een object wordt gekenmerkt door zijn toestand (data) en zijn gedrag (operaties). Om objecten van elkaar te kunnen onderscheiden, heeft elk object zijn eigen unieke identiteit: de *object identiteit*.

Een object is voor de andere objecten in zijn omgeving van belang omdat het dingen voor die andere objecten kan doen: het kan diensten of *services* uitvoeren. Het is de verantwoordelijkheid van het object om deze diensten correct uit te voeren. Een object verbergt de interne details voor de andere objecten (*inkapseling*). Het object kan alleen benaderd worden door een door hemzelf bepaalde interface.

Een synoniem voor object is *instantie*. Deze term wordt vooral gebruikt om aan te geven in welke klasse een object thuishoort.

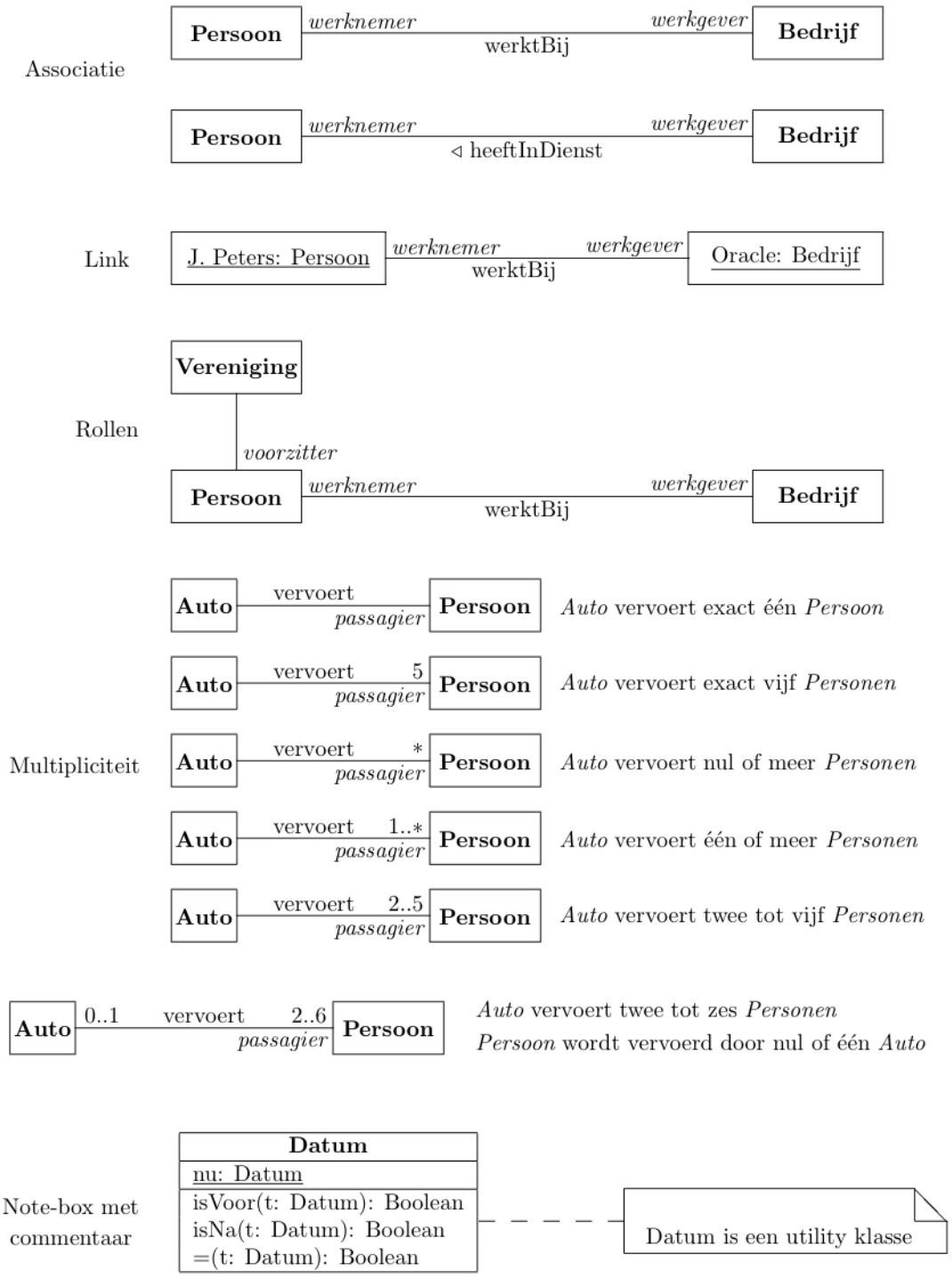
Een instantie kan getekend worden als een rechthoek met daarin onderstreept de symbolische naam van het object, gevolgd door een dubbele punt en de naam van de klasse. Een klassenaam wordt normaal met een hoofdletter geschreven en de namen van instanties met een kleine letter. Soms wordt de symbolische naam van het object weggelaten. Onder de naam mogen eventueel de namen van de attributen en hun waarden geschreven worden.



3.9.2 Klasse

Een *klasse* is een verzameling van objecten met overeenkomstige eigenschappen. De klassebeschrijving geeft de naam van de klasse en de beschrijving van de eigenschappen (attributen en operaties) van de instanties.

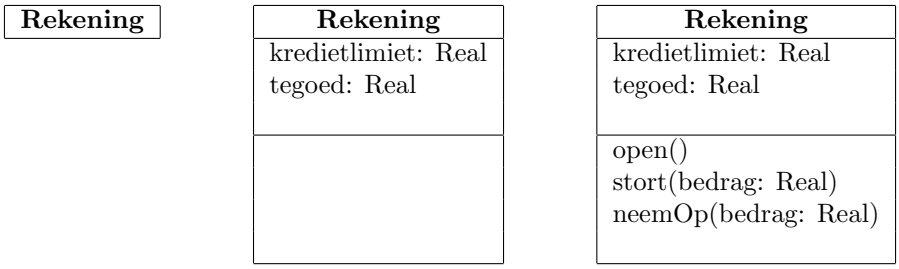
Een *attribuut* is informatie die door een object beheerd wordt. Ieder attribuut heeft precies één waarde voor iedere instantie van de klasse. De toestand van een object wordt weergegeven door de waarden van al zijn attributen. Twee of meer objecten uit dezelfde klasse waarvan op een gegeven moment de waarden van al hun attributen dezelfde zijn, zijn in dezelfde toestand maar blijven verschillende objecten.



Figuur 3.1: Associatie, multipliciteit, rol, note-box

Een *operatie* beschrijft een service die een object levert en bepaalt dus mede het gedrag van een object. Een operatie kan argumenten hebben en een resultaat opleveren. Omdat alle instanties van een klasse dezelfde operaties hebben, worden de operaties beschreven bij de klasse.

De voorstelling van een klasse is een rechthoek met drie delen: bovenaan de naam van de klasse, daaronder de attributen en tot slot de operaties. Het gewenste detailniveau bepaalt of de operaties en/of attributen weggelaten worden.



Objecten communiceren met elkaar door middel van het sturen van *boodschappen*. Een boodschap is een verzoek van een object (*zender* of *actor*) aan een ander object (*ontvanger*) om een bepaalde service te leveren. De ontvanger voert als gevolg hiervan een operatie uit.

3.9.3 Associatie

Een *associatie* is een structurele relatie tussen twee klassen: een instantie uit de ene klasse is gedurende het grootste deel van zijn bestaan in het systeem verbonden met een instantie uit de tweede klasse. Welke instantie uit de tweede klasse dit is, is hiervoor niet relevant en dit kan gedurende de levensloop dan ook wijzigen.

Een associatie wordt aangegeven door een doorgetrokken lijn tussen twee klassen. De naam van de associatie wordt langs de lijn geschreven; er wordt normaal van links naar rechts en van boven naar onder gelezen. Wanneer de leesrichting anders is, wordt bij de naam van de associatie een pijl gezet.

Een *link* is een instantie van een associatie: een associatie beschrijft concrete links tussen concrete instanties.

Een object kan in verschillende associaties verschillende *rollen* spelen: een rol is context-afhankelijk. De rol die een object in de context van een associatie speelt, wordt aangegeven door de rolnaam weer te geven bij het uiteinde van de associatie bij de betreffende klasse. Wanneer de rolnaam weggelaten wordt (niet aan te raden), is deze rolnaam gelijk aan de naam van de klasse.

De *multipliciteit* geeft het aantal instanties van de geassocieerde klasse aan waarmee één instantie van de klasse een link kan hebben. Deze multipliciteit kan één zijn (geen teken), nul of meer (een ster) of een specifiek aantal (een getal).

Multipliciteiten kunnen aan beide zijden van de associatie gezet worden: een auto vervoert twee tot zes personen en een persoon wordt door nul of één auto's vervoerd.

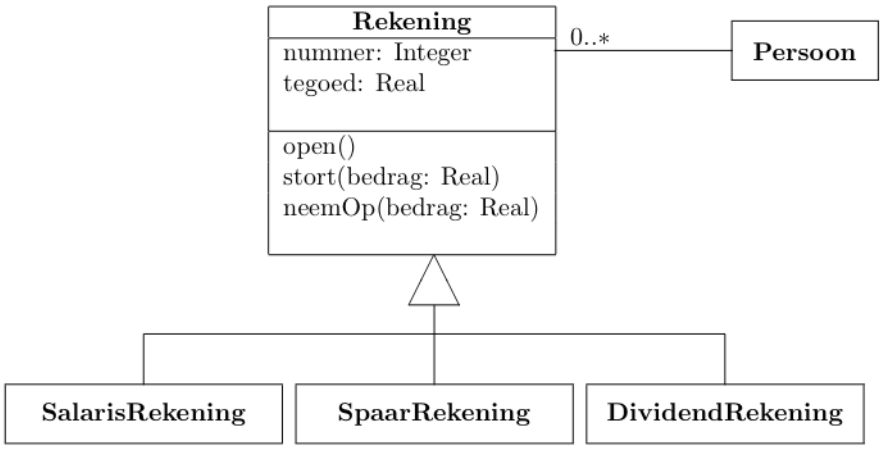
Commentaar kan toegevoegd worden in een *note-box*. Een note-box kan met behulp van een onderbroken lijn gekoppeld worden aan een element in een diagram.

3.9.4 Generalisatie en overerving

Generalisatie is het classificeren van klassen: een aantal klassen die identieke kenmerken (toestand en gedrag) hebben, worden samengebracht in een nieuwe klasse die de gezamenlijke kenmerken bevat. De nieuwe klasse is de *superklasse* van de klassen met dezelfde kenmerken (de *subklassen*). De superklasse wordt de *generalisatie* genoemd, een subklasse een *specialisatie*.

Een super-subklasse relatie wordt aangegeven door een pijl met een grote, gesloten pijlpunt; deze pijl is gericht naar de superklasse. Iedere subklasse erft alle eigenschappen van zijn superklasse. Iedere toevoeging aan een superklasse wordt automatisch toegevoegd aan al zijn subklassen.

Een voorbeeldje van een klassediagramma dat overerving gebruikt is te zien in Figuur 3.2.



Figuur 3.2: Een klein klassediagram

Hoofdstuk 4

Gevorderdere features van Java

4.1 Paketten

Zoals al blijkt uit het vorige hoofdstuk, hecht Java veel belang aan het zoveel mogelijk afschermen van implementatiedetails. Een concept dat hierbij een belangrijke rol speelt, is dat van een *pakket*. Een pakket bestaat uit een verzameling klassen (en eventueel ook interfaces) die samen een bepaalde functionaliteit aanbieden.

Klassen die bij hetzelfde pakket horen hebben bepaalde voorrechten ten opzichte van elkaar: ze hebben toegang tot elkaars pakket-toegankelijke methodes en attributen en ze kunnen naar elkaar op een bondige manier naar elkaar verwijzen door middel van enkel maar de klassenaam. Klassen uit verschillende paketten hebben geen toegang tot elkaars pakket-toegankelijke methodes of attributen en moeten naar elkaar verwijzen met een langere naam, die bestaat uit de naam van het pakket, een punt, en tot slot de naam van de klasse.

Naar de klasse `BufferReader` uit het pakket `java.io` kan bijvoorbeeld als volgt verwezen worden:

```
class IOVoorbeeld
{
  public static void main(String[] args) throws java.io.IOException
  {
    // ...
    java.io.BufferedReader toetsenbord = new java.io.BufferedReader(
                    new java.io.InputStreamReader(System.in));
    //...
  }
}
```

Indien men vaak naar dezelfde klasse moet verwijzen, wordt dit met de volledige gekwalificeerde naam "pakket.klasse" natuurlijk al gauw veel typwerk. Men kan ook een `import`-opdracht gebruiken om toch naar een klasse te kunnen verwijzen met enkel maar zijn naam:

```
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

class IOVoorbeeld
{
    public static void main(String[] args) throws IOException
    {
        // ...
        BufferedReader toetsenbord
```

```
                = new BufferedReader(new InputStreamReader(System.in));
            //...
        }
    }
```

Als men veel klassen uit hetzelfde pakket nodig heeft, kan men dit ook bondiger noteren met behulp van een sterretje:

```
import java.io.*;
```

Dit ene commando zal *alle* klassen uit het pakket `java.io` importeren.

4.1.1 Bestandsstructuur

In de bestandsstructuur op de harde schijf van een computer komen klassen overeen met bestanden en paketten overeen met directories (mappen). Op een Windows machine waar de java klassen geïnstalleerde zijn in een directory `C:\JavaInstallatie\`, zou bijvoorbeeld het volgende commando:

```
import java.io.BufferedReader;
```

op zoek gaan naar dit bestand:
`C:\JavaInstallatie\java\io\BufferedReader.java`

4.1.2 Zelf paketten definiëren

Je kan ook zelf pakketten samenstellen. Neem dan bovenaan in elk bronbestand dat tot dit pakket behoort de volgende opdracht op:

```
1 package <pakketnaam> ;
```

Het spreekt voor zich dat je dan ook je bestand met de juiste naam op de juiste plaats moet opslaan, zoals beschreven in de vorige sectie.

4.1.3 Java Language package

Een interessante eigenschap van Java is dat een installatie standaard al een rijke bibliotheek vol nuttige paketten aanbiedt aan de programmeur. Hier vinden we bijvoorbeeld ondersteuning voor het werken met bestanden, netwerken en allerhande grafische dingen, alsook een aantal algemene voorzieningen.

We vermelden even de volgende paketten:

`java.lang`	een aantal klassieke functies die altijd impliciet geïmporteerd worden
`java.util`	bijkomende algemene voorzieningen
`java.io`	invoer/uitvoer
`java.text`	formattering
`java.awt`	grafische user interfaces
`javax.swing`	een beter pakket voor grafische user interfaces
`java.net`	netwerk

In `java.lang` vinden we bijvoorbeeld de klassen `Object`, `Integer`, `String`, enzovoort.

We vinden in dit pakket ook de klasse `java.lang.System`, waarin zich onder andere volgende stromen bevinden, waarvan we er al twee zijn tegengekomen:

```
public static InputStream in; // Standaard invoer
public static PrintStream out;  // Standaard uitvoer
public static PrintStream err;  // Stroom voor foutmeldingen
```

Daarnaast bevat de klasse `System` ook een methode

```
public static void exit(int  status)
```

Deze wordt gebruikt om een programma te beëindigen met een bepaalde status code. De conventie is dat een status code van 0 aangeeft dat het programma "succesvol" gelopen heeft, en dat andere codes aangeven dat er iets in misgegaan.

Een andere nuttige methode uit `System` is de methode `getProperty`, waarmee een programma informatie aan de weet kan komen over de omgeving waarin het draait.

```
class GetPropertyDemo {
    public static void main(String args[]) {
        // Print de naam van het besturingssysteem af
        System.out.println ( System.getProperty("os.name") );
    }
}
```

Een andere nuttige klasse die we al zijn tegengekomen is de klasse `java.lang.String`. Hier zijn een paar interessante methoden uit deze klassen:

```
  // Returns the character at offset index
public char charAt(int  index);
  // Compares string with another, returning 0 if there's a match
public int compareTo(String  anotherString);
  // Returns a new string equal to anotherString appended to the current string
public String concat(String  anotherString);
  // Returns the length of the current string
public int length();
  // Returns true if the current string begins with prefix
public boolean startsWith(String  prefix);
  // Returns true if the current string ends in suffix
public boolean endsWith(String  suffix);
  // Returns the remainder of the string, starting from offset beginIndex
public String substring(int  beginIndex);
  // Returns a substring from offset beginIndex to offset endIndex
public String substring(int  beginIndex, int endIndex);
  // Returns the current string, converted to lowercase
public String toLowerCase();
  // Returns the current string, converted to uppercase
public String toUpperCase();
```

Merk op dat Strings zelf onveranderlijke objecten zijn, dus geven methodes zoals `toUpperCase` telkens een nieuwe String terug in plaats van de huidige String te veranderen.

4.2 Fouten opvangen met Exception

Professioneel programma-ontwerp vereist dat een programma gewapend is tegen een omgeving die zich niet gedraagt zoals het hoort. Met name netwerkverbindingen en input/output, de configuratie van de eigen bestanden, maar ook interne programmafouten kunnen een informatiesysteem grondig blokkeren.

Daarom zal een goed programmeur regelmatig tests inbouwen of de zojuist uitgevoerde opdrachten het verhoopte resultaat hebben gehad. Als de test positief uitvalt, gaat het programma verder met de normale afwikkeling; als de test negatief is, probeert het programma de fout te herstellen alvorens verder te gaan, of stopt het met een duidelijke foutmelding.

Om te vermijden dat dit soort kwaliteitszorg aanleiding geeft tot een onoverzichtelijk kluwen van **if**/**else**-opdrachten die het programma erg onleesbaar zouden maken, beschikken Java en enkele andere talen over het mechanisme van *uitzonderingen* of *exceptions.*

De Java-klasse Exception modelleert een algemene, niet-fatale fout in de werking van een programma. Van deze klasse zijn tientallen andere rechtstreeks of onrechtstreeks afgeleid om bijzondere soorten van fouten te modelleren. Enkele voorbeelden:

- ArithmeticException: een niet-toelaatbare rekenkundige bewerking, zoals het delen van twee getallen waarbij het tweede nul blijkt te zijn (*bij getallen met vlottende komma genereert dit geen uitzondering, maar het bijzondere getal 'Not a Number' ofwel NaN*).
- ArrayIndexOutOfBoundsException: verwijzen naar een element van een rij met een index die negatief is, of groter dan of gelijk aan het totale aantal elementen.
- ClassNotFoundException: een declaratie of opdracht verwijst naar een klasse die niet beschikbaar is – deze exception treedt ook op als je een niet-bestaand Java-programma probeert op te roepen, of als je een typfout maakt in de naam van de klasse.
- IOException: een fout bij het lezen of schrijven van een extern informatiekanaal, bijvoorbeeld een bestand.
- NullPointerException: gebruik maken van eigenschappen of methoden van een veranderlijke of parameter van een referentietype die niet naar een object verwijst, bijvoorbeeld omdat er sinds de declaratie geen waarde aan toegekend is.

Bij het mechanisme van uitzonderingen horen vijf nieuwe Java-sleutelwoorden: **try**, **catch**, **finally**, **throws** en **throw**.

De eerste drie horen bij elkaar en dienen voor het *opvangen en afhandelen* van een fouttoestand. De algemene vorm luidt

```
1  try
2  {
3      // normale opdrachten
4  }
5  catch (UitzonderingsType1 naam1)
6  {
7      // afhandeling van fouten van type UitzonderingsType1
8  }
9  catch (UitzonderingsType2 naam2)
10 {
11     // afhandeling van fouten van type UitzonderingsType2
12 }
13 // ...
14 catch (UitzonderingsTypeN naamN)
15 {
16     // afhandeling van fouten van type UitzonderingsTypeN
17 }
18 finally
19 {
20     // afsluitende opdrachten
21 }
```

De normale opdrachten worden in de aangegeven volgorde uitgevoerd, totdat voor het eerst een uitzonderingstoestand optreedt. Vanaf dat punt worden alle verder opdrachten in het blok normale opdrachten genegeerd, en gaat het systeem op zoek naar het eerste van de rij uitzonderingstypes dat overeenkomt met de optredende fout. Het bijbehorende afhandelingsblok wordt uitgevoerd.

Als tijdens de normale opdrachten geen uitzondering optreedt, worden na de afwerking van dit blok alle **catch**-gedeelten overgeslagen. Of er nu een uitzondering is opgetreden of niet, de afsluitende opdrachten in het **finally**-blok worden in elk geval uitgevoerd. Dit laatste blok mag overigens ook worden weggelaten.

Voorbeeld. In het volgende programma komt ogenschijnlijk een oneindige lus voor: de voorwaarde **true** van de **while**-opdracht zorgt ervoor dat deze iteratie nooit op de normale wijze eindigt. Maar de deling door nul die na verloop van tijd onvermijdelijk optreedt, zorgt ervoor dat

het programma overgaat naar het **catch**-gedeelte waarin de ArithmeticException wordt opgevangen.

```
1  class DelingDoorNul
2  {
3      public static void main(String[] args)
4      {
5          try
6          {
7              int i = 10;
8              while (true)
9              {
10                 System.out.println("10 : " + i + " = " + 10/i);
11                 i--;
12             }
13         }
14         catch (ArithmeticException e)
15         {
16             System.out.println("rekenfoutje:" + e);
17         }
18         finally
19         {
20             System.out.println("eind goed, al goed");
21         }
22     }
23 }
```

We krijgen de volgende uitvoer.

```
10 : 10 = 1
10 : 9 = 1
10 : 8 = 1
10 : 7 = 1
10 : 6 = 1
10 : 5 = 2
10 : 4 = 2
10 : 3 = 3
10 : 2 = 5
10 : 1 = 10
rekenfoutje:java.lang.ArithmeticException: / by zero
eind goed, al goed
```

Het sleutelwoord **throw** wordt gevolgd door een uitdrukking van het type Exception en doet het omgekeerde: het onderbreekt de lopende code met een foutconditie, zodat de virtuele machine op zoek gaat naar het eerstvolgende passende **catch**-blok. Als de **throw**-opdracht zich niet uitdrukkelijk binnen een **try**-blok bevindt, wordt de uitzondering doorgegeven aan de methode van waaruit de huidige methode werd aangeroepen.

Als geen enkele methode in de ketting van aanroepen vanaf main tot de plaats van de **throw**-opdracht een passende **catch** formuleert voor de optredende uitzondering, dan wordt de uitzondering doorgegeven aan de runtime-omgeving. Deze vangt standaard *alle* uitzonderingen op met het afbreken van het programma en het afdrukken van een reeks nogal cryptische meldingen. Als we in bovenstaand programma *geen* voorzorgsmaatregelen inbouwen:

```
1  class DelingDoorNulOnveilig
2  {
3      public static void main(String[] args)
```

```
4       {
5           int i = 10;
6           while (true)
7           {
8               System.out.println("10 : " + i + " = " + 10/i);
9               i--;
10          }
11      }
12  }
```

dan valt dit nogal mee, de runtime-omgeving meldt ons

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DelingDoorNulOnveilig.main(DelingDoorNulOnveilig.java:5)
```

en stopt.

Omdat een niet-opgevangen uitzondering zonder pardon doorgegeven wordt aan de aanroepende methode, zouden programmeurs die andermans werk gebruiken, voor onaangename verrassingen kunnen staan. Om dat te vermijden is het verplicht dat *elke uitzondering die binnen een methode kan optreden, en die niet opgevangen wordt binnen die methode, gesignaleerd wordt* door het sleutelwoord **throws** in de hoofding van de methode-declaratie.

Voorbeeld. De volgende methode waarschuwt dat ze uitzonderingen van het type IOException en SQLException kan veroorzaken, hetzij rechtstreeks, hetzij onrechtstreeks door de aanroep van andere methoden.

```
public int registAantalDeelnemers throws SQLException, IOException
{
  //...
}
```

Deze verplichting heet de "catch or specify"-regel en wordt effectief door de Java-compiler afgedwongen. Ze geldt *niet* voor uitzonderingen die afgeleid zijn van het subtype RuntimeException, omdat dit aanleiding zou geven tot een al te groot aantal **throws**-clausules.

Zo is bijvoorbeeld een ArithmeticException een bijzonder geval van een RuntimeException. Gelukkig maar: anders zou elke methode waarin een deling tussen gehele getallen voorkwam, een exception moeten opvangen of specificeren!

Als je zelf foutcondities wil signaleren aan andere programmagedeelten, dan creëer je beter een nieuw type uitzondering dan dat je gebruikmaakt van de bestaande types. De bestaande types slaan allemaal op welgedefinieerde soorten fouten. Je kan een nieuw type uitzondering definiëren door gewoon een nieuwe subklasse te maken van een bestaand type uitzondering. In het volgende voorbeeld maken we een nieuw type, VerkeerdWachtwoord, door het af te leiden van Exception.

```
class VerkeerdWachtwoord extends Exception
{
}
```

Voorbeeld. In het programma MachtsRecursie moet opgelet worden voor een negatieve exponent. De methode zou eventueel oneindig lang kunnen blijven lopen. We declareren hiervoor een speciaal uitzonderingstype en gebruiken het om eventuele fouten van dit type te signaleren aan het hoofdprogramma.

```
1  import java.io.*;
2
3  class NegatieveExponent extends Exception
4  {
```

```
5       NegatieveExponent(String s)
6       {
7           super(s);
8       }
9   }
10
11  class MachtsRecursieVeilig
12  {
13      public static int machtsverheffing(int grondtal, int exponent)
14                                  throws NegatieveExponent
15      {
16          if (exponent == 0)
17              return 1;
18          else if (exponent < 0)
19              throw new NegatieveExponent("" + exponent);
20          else
21              return grondtal * machtsverheffing(grondtal, exponent - 1);
22      }
23
24      public static void main(String[] args) throws IOException
25      {
26          int grondtal, exponent, macht;
27          BufferedReader toetsenbord;
28          toetsenbord = new BufferedReader(new InputStreamReader(System.in));
29          System.out.println("grondtal: ");
30          grondtal = Integer.parseInt(toetsenbord.readLine());
31          System.out.println("exponent: ");
32          exponent = Integer.parseInt(toetsenbord.readLine());
33          try
34          {
35              macht = machtsverheffing(grondtal, exponent);
36              System.out.println(grondtal + " tot de macht " + exponent
37                                          + " is gelijk aan " + macht);
38          }
39          catch (NegatieveExponent fout)
40          {
41              System.out.println("Exponent mag niet negatief zijn:" + fout);
42          }
43      }
44  }
```

We kunnen nu verklaren waarom het lezen van het toetsenbord gepaard moest gaan met de clausule **throws** IOException. Telkens wanneer we gebruik maken van de methode readLine() kan namelijk een uitzondering van dat type optreden. In plaats daarvan kunnen we ook de **throws**-clausule weglaten en de readLine-opdrachten in één of meer **try**-blokken plaatsen.

4.3 Collecties

Om verzamelingen van objecten bij te houden, kan je de ingebouwde rijen (*arrays*) gebruiken. Dit is echter niet erg flexibel, aangezien een rij een vaste grootte heeft, en dus niet kan groeien of krimpen naargelang er meer of minder objecten in moeten. Om deze reden biedt Java een aantal andere nuttige klassen aan.

4.3.1 Vector

Een `Vector` is conceptueel identiek aan een rij van `Objecten`, maar dan met het verschil dat een `Vector` zijn grootte automatisch aanpast aan het aantal objecten dat erin zit.

```
1  Vector v = new Vector();
2  v.addElement(new Integer(5));
3  v.addElement(new Integer(10));
4  for (int i = 0; i < v.size(); i++) {
5    System.out.println(v.elementAt(i));
6  }
7  v.setElementAt(new Integer(0), 1);
```

De klasse `Vector` zit in het pakket `java.util`, dus om haar met haar korte naam te gebruiken, moet je haar importeren:

```
import java.util.Vector;
```

Je kan natuurlijk ook objecten van je eigen klassen in een `Vector` steken:

```
1  public class EigenKlasse {
2    public void eigenMethode() {
3    }
4    public static void main(String[] args) {
5      Vector v = new Vector();
6      EigenKlasse a = new EigenKlasse();
7      v.addElement(a);
8      ((EigenKlasse) v.elementAt(0)).eigenMethode();
9    }
10 }
```

Je ziet dat er in de laatste lijn een typecast nodig is: de methode `elementAt` geeft immers een `Object` terug, dus we moeten de compiler vertellen dat dit `Object` in feite van de klasse `EigenKlasse` is, vooraleer we de methode `eigenMethode` erop mogen toepassen. Dit is niet ideaal. Het levert ons natuurlijk wat onnodig schrijfwerk op, maar belangrijker is dat we op deze manier gedwongen worden om de strenge typering van Java te omzeilen. Met andere woorden, de compiler kan nu minder streng de correctheid van ons programma te controleren, wat het risico op *runtime* fouten verhoogt.

Recente versies van Java passen hier een mouw aan door middel van het *Collections* framework, dat gebruik maakt van zogenaamde *Generics* om dit probleem te vermijden.

4.3.2 Generics en collections

In feite zouden klassen zoals `Vector` zich wat betreft typering beter gedragen zoals rijen, waarbij je bij het declareren kan aangeven welk type van objecten je erin gaat bewaren. Hiervoor kan het *Collections* raamwerk gebruikt worden. Dit bestaat uit een aantal zogenaamd *generische* interfaces. Een voorbeeld is de interface `List<T>`, die staat voor een lijst van objecten van het type `T`. Zo is `List<Integer>` een lijst van `Integer` objecten, `List<String>` een lijst van strings, enzoverder.

`List<T>` is een interface, die geïmplementeerd wordt door een aantal klassen, zoals bijvoorbeeld `ArrayList<T>`.

```
1  List<String> list = new ArrayList<String>();
2  list.add("hallo");
3  for (int i = 0; i < list.size() ; i++) {
4      String s = list.get(i);
5  }
6  list.set(0, "hoi");
```

Merk op dat de methodes `get`, `set` en `add` allemaal een iets kortere naam hebben dan hun tegenhanger uit `Vector`.

Naast lijsten, bevat het *Collections* raamwerk ook andere nuttige interfaces zoals `Set<T>` (een verzameling van objecten van type `T`), `Map<K,T>` (een afbeelding van sleutels van type `K` op waarden van type `T`) en `Queue<T>` (een wachtrij van obectjen van type `T`). Al deze interfaces erven over van de basisinterface `Collection`, en hebben dus een aantal gemeenschappelijke methoden, zoals `add(T)` (het toevoegen van een object) en `contains(T)` (nagaan of een object al in de collectie zit.

De interface `Collection` (en dus ook alle interfaces die hiervan overerven) biedt ook de mogelijkheid om met een `Iterator` te werken: dit is een object dat gebruikt wordt om al de objecten in de collectie af te lopen. Concreet bevat deze interface volgende methode:

```
public interface Collection<T> {
  ...
  Iterator<T> iterator();
  ...
}
```

Deze methode geeft een `Iterator` over objecten van type `T` terug. Een `Iterator` is op zijn beurt een interface, die twee eenvoudige methodes aanbiedt:

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

We kunnen dus bijvoorbeeld een `ArrayList` ook als volgt doorlopen:

```
List<String> list = new ArrayList<String>();
list.add("hallo");
String s;
for ( Iterator<String> it = list.iterator(); it.hasNext() ;  ) {
  s = it.next()
  System.out.println(s);
}
list.set(0, "hoi");
```

Een nog handigere manier aan om over collecties te itereren is de zogenaamde *for-each* lus.

```
1 List<String> list = new ArrayList<String>();
2 list.add("hallo");
3 for ( String s: list ) {
4   System.out.println(s);
5 }
```

Al deze klassen en interfaces zitten ook in het pakket `java.util`.

Hoofdstuk 5

Applicaties met een grafische user-interface

5.1 Het MVC paradigma

Om code te verkrijgen die onderhoudbaar en herbruikbaar is, is het van groot belang dat GUI code strikt gescheiden wordt van de rest code. Om dit verwezenlijken, maken we gebruik van het MVC paradigma, waarbij we een onderscheid maken tussen volgende drie componenten:

- De *View* bevat alle GUI code, dwz. alle code die te maken heeft met het tonen van informatie aan de gebruiker of met het opvragen van informatie aan de gebruiker.
- De *Controller* stuurt de applicatie aan. Hierin zit de code die zorgt voor het opstarten van de applicatie en de code die bepaalt wat er wanneer gebeurt.
- Het *Model* bevat alle domeinkennis. Hierin bevindt zich de code die zorgt voor het beheer van gegevens waarmee het programme moet werken.

Een typische MVC applicatie begint als volgt:

- De controller initialiseert het model
- De controller initialiseert de view en geeft de verschillende view componenten hierbij een link naar de gegevens uit het model die ze moeten visualiseren
- De view toont de gegevens aan de gebruiker

Als een gebruiker een actie onderneemt, wordt deze als volgt afgehandeld:

- De view component die de actie van de gebruiker opmerkt, geeft dit door aan de controller
- De controller vraagt eventuele invoer van de gebruiker op aan de view en geeft deze gegevens door aan het model
- Het model past zijn gegevens aan aan de nieuwe informatie
- De controller verwittigt de view dat er gegevens veranderd zijn
- De view raadpleegt het model om te ontdekken wat er veranderd is, en past zichzelf hieraan aan

5.2 Een GUI bouwen

Om GUIs te maken in Java gebruik je de pakketten `java.awt` en `javax.swing`. Swing is een nieuwere versie van de GUI, waarin een aantal tekortkomingen van het oudere AWT (Abstract Windowing Toolkit) worden opgelost. Om verwarring te vermijden met gelijknamige AWT klassen, beginnen klassenamen in Swing vaak met de letter J.

Een GUI is opgebouwd uit een aantal vensters. In Swing is elk venster normaalgezien een object van de klasse JFrame. In een venster bevinden zich een aantal GUI-componenten: objecten van (een subklasse van) de klasse JComponent. Een belangrijke subklasse van JComponent is de klasse JContainer: dit zijn componenten die op hun beurt weer andere componenten bevatten. Om componenten toe te voegen aan een container gebruik je de methode `JContainer.add(JComponent c)`.

Elk JFrame heeft een JContainer waarin alle componenten geplaatst worden die in dit frame te zien moeten zijn. Deze vraag je op via de methode `JFrame.getContentPane()`.

De volgende soorten componenten worden vaak gebruikt:

- `JPanel`: een generieke JContainer, waarin je andere componenten kan onderbrengen
- `JButton`: een knop
- `JLabel`: een stukje tekst
- `JTextField`: tekstveldje waarin de gebruiker tekst kan invullen. Met `getText()` vraag je deze tekst op.
- `JTextArea`: tekstveld van meerdere lijnen
- `JCheckBox`: valt aan en uit te vinken. Met `isSelected()` kom je de toestand ervan te weten.
- `JRabioButton`: om één mogelijkheid uit een aantal opties te laten selecteren. Al deze opties moeten dan samen in dezelfde `ButtonGroup` zitten, omdat Swing anders niet weet welke opties samenhoren. Met `isSelected()` kom je van elke RadioButton te weten of hij geselecteerd is.

Onderstaand voorbeeld toont hoe je een frame kan maken met daarin een tekstboodschap:

```
1  public class Test {
2
3    public static void main(String[] args) {
4      JFrame jf = new JFrame();
5      jf.getContentPane().add(new JLabel("Hallo wereld!");
6      jf.setVisible(true);
7      jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8    }
9
10 }
```

De laatste lijn van deze methode dient om ervoor te zorgen dat de applicatie ook beëindigd wordt op het moment dat de gebruiker dit frame sluit. Het werk dat we nu in deze `main` methode doen, gebeurt ook vaak in de constructor van een eigen subklasse van JFrame.

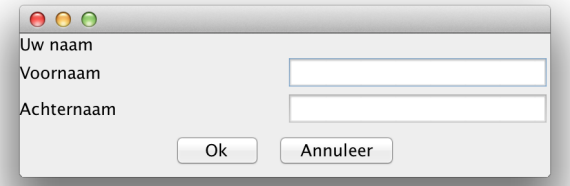
Als er meerdere componenten in dezelfde container zitten, beslist de LayoutManager die bij de container hoort hoe deze juist zullen worden afgebeeld, dwz. waar elke component zal staan en hoe groot hij zal zijn. De LayoutManager die bij een container hoort, kan je aanpassen met de methode `JContainer.setLayout(LayoutManager m)`. Er zijn o.a. volgende mogelijkheden:

- GridLayout(n,m): een matrix van n rijen en m kolommen.
- FlowLayout: componenten worden van links naar rechts en van boven naar onder gerangschikt, zoals wanneer je letters op een blad papier schrijft.

- BoxLayout(*container*,*orientatie*): al de componenten die in *container* zitten worden ofwel naast elkaar ofwel onder elkaar gezet; dit hangt af van of *orientatie* = `BoxLayout.X_AXIS` of *orientatie* = `BoxLayout.Y_AXIS`.
- BorderLayout: er is een groot middenvlak en in elke windrichting is er een klein gebied. Bij het toevoegen van een component aan een container met een BorderLayout, specifieer je waar de component moet komen. Bijvoorbeeld: `cont.add(comp, BorderLayout.CENTER)` of `cont.add(comp, BorderLayout.NORTH)`. (Ipv. windrichtingen kunnen ook `PAGE_START`, `LINE_END`, enz. gebruikt worden: deze aanduidingen zijn relatief tov. de localisatie-instellingen van de gebruiker.)
- CardLayout: deze layout toont altijd maar één van de componenten die hij bevat. Met de methode `CardLayout.show(container,naam)` wissel je naar de component met een bepaalde `naam` (de naam van een component wordt in dit geval meegegeven als tweede argument van de `add` methode).

Dankzij het feit dat containers genest kunnen worden (JContainer is immers een subklasse van JComponent), volstaan deze op zich erg eenvoudige layouts om toch complexe GUIs te kunnen ontwerpen.

Onderstaand voorbeeld is een BorderLayout, waarvan de CENTER positie wordt ingenomen door een 2x2 GridLayout:



De code om het frame te produceren is als volgt:

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  class NaamKaartje extends JFrame {
6
7      private TekstVeldPaneel tekstveldjes = new TekstVeldPaneel();
8      private KnopjesPaneel knopjes;
9      private JLabel titel = new JLabel("Uw naam");
10
11     public NaamKaartje (Controller c) {
12     knopjes = new KnopjesPaneel(c);
13     Container paneel = getContentPane();
14     paneel.setLayout(new BorderLayout());
15     paneel.add(tekstveldjes, BorderLayout.CENTER);
16     paneel.add(knopjes, BorderLayout.SOUTH);
17     paneel.add(titel, BorderLayout.NORTH);
18     pack();
19     setVisible(true);
20     }
21
```

```
22
23      public String getVoornaam() {
24      return tekstveldjes.getVoornaam();
25      }
26
27      public String getAchternaam() {
28      return tekstveldjes.getAchternaam();
29      }
30
31  }
```

Het JPanel in het midden van de BorderLayout wordt als volgt gemaakt:

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  class TekstVeldPaneel extends JPanel {
5
6       private JLabel vnaamLabel;
7       private JTextField vnaamVeld;
8       private JLabel anaamLabel;
9       private JTextField anaamVeld;
10
11      TekstVeldPaneel() {
12      vnaamLabel = new JLabel("Voornaam");
13      vnaamVeld = new JTextField(16);
14      anaamLabel = new JLabel("Achternaam");
15      anaamVeld = new JTextField(16);
16      setLayout(new GridLayout(2,2));
17      add(vnaamLabel);
18      add(vnaamVeld);
19      add(anaamLabel);
20      add(anaamVeld);
21      }
22
23      String getVoornaam() {
24      return vnaamVeld.getText();
25      }
26     String getAchternaam() {
27      return anaamVeld.getText();
28      }
29
30  }
```

De code van het KnopjesPaneel in het zuiden stellen we nog even uit tot Sectie 5.4.

Naast werken met een LayoutManager, is het ook mogelijk om zelfs de layout van een container te regelen. Hiervoor doe je een `container.setLayout(null)`. Daarna kan je dan de componenten zelf een plaatsje geven dmv. hun setLocation en setSize methodes. Dit valt echter af te raden, omdat dit vaak lelijke resultaten geeft als de gebruiker de vensters een andere grootte geeft dan de programmeur in gedacht had. Als je toch de grootte van de individuele componenten wilt bepalen, kan je dit doen door methodes als setMinimumSize, setMaximumSize of setPreferredSize. De LayoutManager zal met deze voorkeuren rekening houden in de mate van het mogelijke.

5.3 Gedrag aan een GUI koppelen

Om je applicatie te laten reageren op acties van de gebruiker, maak je gebruik van *Listeners*. Deze listeners zijn er in een hele hoop verschillende soorten en je kan aan elke component bepaalde soorten listeners koppelen. Al deze listeners werken op dezelfde manier. We illustreren deze werking aan de hand van een van de meest gebruikte listeners, namelijk de *ActionListener*. Dit is wat je o.a. gebruikt wanneer de gebruiker op een knop klikt.

ActionListener zelf is een interface:

```
1 public interface ActionListener {
2   public void actionPerformed(ActionEvent a);
3 }
```

Dit betekent dat je, om een ActionListener te kunnen gebruiken, zelf een klasse moet maken die deze interface implementeert (en die dus ook een methode `actionPerformed(ActionEvent a)` heeft). Bijvoorbeeld:

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class MijnListener implements ActionListener{
5     public void actionPerformed(ActionEvent a) {
6         System.out.println("Hallo");
7     }
8 }
```

Eenmaal deze klasse gedefinieerd is, kan je dan een object hiervan koppelen aan een "klikbare" component, bv. een JButton. Dit gaat zo:

```
1 ...
2 JButton knop = new JButton("Klik mij");
3 MijnListener mijn = new MijnListener();
4 knop.addActionListener(mijn);
5 ...
```

Als de gebruiker nu op de knop drukt, zal deze al de ActionListeners die er aan werden toegevoegd (in dit geval dus het object `mijn`), verwittigen door van elk van deze listeners de actionPerformed methode op te roepen. Het ActionEvent argument dat hierbij als argument wordt meegegeven, geeft extra informatie over de actie van de gebruiker, zoals bv. de component waarop geklikt werd (`ActionEvent.getSource()`).

In de praktijk wordt hierbij vaak gebruik gemaakt van een zogenaamde *anonymous inner class*. Dat ziet er zo uit:

```
1 ...
2 JButton knop = new JButton("Klik mij");
3 knop.addActionListener(new ActionListener() {
4     public void actionPerformed(ActionEvent a) {
5         System.out.println("Hallo");
6     }
7 });
8 ...
```

Wat hier gebeurt, is dat er een nieuwe klasse (zonder naam) wordt gemaakt, die de interface ActionListener implementeert en die de methode actionPerformed bevat. Van deze nieuwe klasse wordt er dan een nieuw object aangemaakt en dit nieuwe object wordt meegegeven als argument van addActionListener. Aangezien de nieuwe klasse geen naam heeft, is het onmogelijk om deze klasse nadien nog te gebruiken.

Als er verschillende knoppen op de user interface zijn, is het aan te raden om aan elke knop een aparte actionListener te koppelen. Een alternatief is om dezelfde actionListener te gebruiken voor verschillende knoppen en dan code als dit te schrijven:

```
1       public void actionPerformed(ActionEvent a) {
2          if (a.getSource() == knop1) {
3             ...
4          }
5          else {
6              if (a.getSource() == knop2) {
7                  ...
8              }
9              ...
10         }
11      }
```

Dit levert echter minder leesbare code op en wordt dus beter vermeden.

5.4 Een volledig MVC voorbeeld

In Sectie 5.2 zagen we reeds een gedeelte van een GUI waarmee we een gebruiker om zijn voor- en achternaam kunnen vragen. We breiden dit nu uit naar een volledige MVC applicatie.

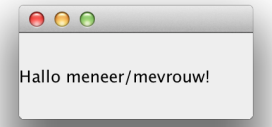
Bij het ontwikkelen (of bestuderen) van een dergelijke applicatie, begin je best bij het model, aangezien deze component volledig op zichzelf geschreven/begrepen moet kunnen worden, onafhankelijk van view en controller.

In dit geval wordt het model gevormd door volgende klasse Persoon:

```
1  class Persoon {
2
3      private String voornaam;
4      private String achternaam;
5
6      public Persoon(String v, String a) {
7      setVoornaam(v);
8      setAchternaam(a);
9      }
10
11     public void setVoornaam(String s) {
12     voornaam = s;
13     }
14
15     public void setAchternaam(String s) {
16     achternaam = s;
17     }
18
19     public String getVoornaam() {
20     return voornaam;
21     }
22
23     public String getAchternaam() {
24     return achternaam;
25     }
26
27     private static final Persoon anoniem = new Persoon("meneer/mevrouw", "");
28
```

```
29      public static Persoon onbekende() {
30      return anoniem;
31      }
32
33  }
```

Om de gegevens die in een Persoon object vervat zitten te visualeren, gebruiken we volgende View klasse.




```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class Begroeter extends JFrame {
5
6      private JLabel boodschap = new JLabel();
7      private Persoon persoon;
8
9      public Begroeter(Persoon p) {
10     setPersoon(p);
11     getContentPane().add(boodschap);
12     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13     setSize(200,200);
14     setVisible(true);
15     }
16
17     public void setPersoon(Persoon p) {
18     persoon = p;
19     update();
20     }
21
22     public void update() {
23     boodschap.setText("Hallo " + persoon.getVoornaam() + "!");
24     }
25
26 }
```

Dit is een typische MVC structuur: de gegevens zitten in het model, en een view klasse houdt een link bij naar de model-klasse wiens gegevens ze visualiseert. Als er iets verandert aan de gegevens in het model, volstaat het om de `update` methode van deze view klasse op te roepen om er voor te zorgen dat de visualisatie wordt aangepast aan de veranderingen in het model.

Het koppelen van view aan model gebeurt door de controller:

```
1  import java.awt.event.*;
2
3  public class Controller {
4
```

```
5       private Persoon persoon;
6       private Begroeter begroeter;
7       private NaamKaartje naamkaartje;
8
9       public Controller() {
10      persoon = Persoon.onbekende();
11      begroeter = new Begroeter(persoon);
12          naamkaartje = new NaamKaartje(this);
13      }
14      public void updatePersoon() {
15      persoon.setVoornaam(naamkaartje.getVoornaam());
16      persoon.setAchternaam(naamkaartje.getAchternaam());
17      begroeter.update();
18      }
19
20      public void sluitVenster() {
21      naamkaartje.dispose();
22      }
23
24      public static void main(String[] a) {
25      Controller c = new Controller();
26      }
27
28  }
```

Zoals gebruikelijk, is de controller de klasse die de `main` methode van de applicatie bevat. Deze main methode doet echter niet veel werk: ze maakt gewoon een nieuw Controller object aan. In de constructor van deze klasse worden dan model objecten aangemaakt, die worden meegegeven als argument bij het aanmaken van view klassen.

Naast deze initialisatie code, bevat de controller ook methodes (in dit geval `updatePersoon` en `sluitVenster`) die bepalen hoe de applicatie zal reageren op acties van de gebruiker. De code die schuilgaat achter het JPanel met de twee knoppen “Ok” en “Annuleer” doet dan ook niet veel meer dan deze twee contoller methodes oproeper. Ook dat is typisch.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  class KnopjesPaneel extends JPanel {
6
7       private JButton ok = new JButton("Ok");
8       private JButton annuleer = new JButton("Annuleer");
9
10      KnopjesPaneel(final Controller c) {
11      setLayout(new FlowLayout());
12      add(ok);
13      add(annuleer);
14      ok.addActionListener(new ActionListener() {
15          public void actionPerformed(ActionEvent a) {
16              c.updatePersoon();
17
18          }});
19      annuleer.addActionListener(new ActionListener() {
20          public void actionPerformed(ActionEvent a) {
21              c.sluitVenster();
```

```
22              }});
23          }
24
25  }
```

(De reden dat het keyword `final` hier gebruikt wordt, is dat we deze variabele gebruiken vanuit methodes van de anonieme inner classes die we hier definiëren. Het is niet evident dat deze variable nog *in scope* is binnen deze inner classes, maar Java laat dit toch toe, op voorwaarde dat hij als `final` gedeclareerd is (wat betekent dat zijn waarde niet meer mag veranderen na de initiële toekenning).)

Nu we alle code van deze applicatie gezien hebben, overlopen we nog eens wat er zal gebeuren op het moment dat de gebruiker zijn naam heeft ingevuld en op de "Ok"-knop klikt. De grote lijnen van deze control flow zijn immers hetzelfde voor alle MVC applicaties.



1. De actie van de gebruiker wordt geregistreerd in de view
2. De view verwittigt de controller van de actie
3. De controller vraagt de invoer van de gebruiker op aan de view
4. De controller geeft deze invoer door aan het model
5. Het model past zichzelf aan
6. De controller verwittigt de view dat het model is aangepast
7. De view vraagt de gegevens van het model op en past zichzelf aan

Dit lijkt misschien een eerder omslachtige manier van werken, aangezien het eindresultaat immers hetzelfde is als wanneer de view zelf het model en zichzelf zou aanpassen. De praktijk heeft echter uitgewezen dat de MVC manier van werken op lange termijn cruciaal is om onderhoudbare code te produceren.

5.5 Nog meer listeners

Naast de ActionListener, zijn er nog veel andere listeners, die allemaal op dezelfde manier gebruikt worden:

- ChangeListener: er iets veranderd
- DocumentListener: de gebruiker heeft iets verandert aan stukje tekst (bv. in JTextField of JTextArea). Typisch gebruik: `jtextfield.getDocument().addDocumentListener(...)`
- WindowListener: venster gesloten, geopend, geminimaliseerd, gemaximaliseerd, ...
- MouseListener: de muis is over een bepaalde component gekomen of er weer uitgegaan, of een knop is ingedrukt of weer losgelaten of geklikt (= op korte tijd ingedrukt en weer losgelaten). Handig als je bv. de gebruiker iets wilt laten verslepen. Meestal implementeer je niet rechtstreeks deze listener, maar erf je over van de "convenience class" MouseAdapter, waarin je enkel de methodes die je effectief nodig hebt moet implementeren.

Naast deze lijst, zijn er nog veel andere. Raadpleeg gerust de Java documentatie.

5.6 Nog meer componenten

- JSlider: om een numeriek waarde uit een interval te selecteren. Als je verwittigd wil worden wanneer de gebruiker de slider versleept, moet je een ChangeListener registreren. `getValue()` geeft de gekozen waarde terug.
- JProgressBar: om het reeds voltooide percentage van een proces te tonen (of te doen alsof).
- JScrollBar: lijkt erg op een JSlider, maar dan met het verschil dat een JSlider één waarde selecteert, terwijl de een scrollbar een bepaald subinterval (met een grootte die de `extent` genoemd wordt) uit het volledige interval selecteert. Met een `AdjustmentListener` kan je verwittigd worden als de gebruiker de scrollbar aanpast. Meestal gebruik je niet de JScrollBar zelf, maar wel:
- JScrollPane: aan de constructor hiervan geef je een andere component als argument en de JScrollPane zorgt dan automatisch voor het verschijnen van horizontale/verticale scrollbars indien nodig.
- JComboBox: Hetzelfde als een verzameling RadioButtons, maar dan in één drop-down-menuutje. Met `getSelectedItem()` kom je te weten welke optie gekozen is en met `getSelectedIndex()` de hoeveelste optie dit was.
- JList: zoiets als een JComboBox, maar waaruit je meerdere opties kan selecteren ipv. slechts één. `getSelectedValuesList` of `getSelectedIndices` vertellen je wat deze zijn.
- JTabbedPane: laat de gebruiker kiezen tussen aantal tabs. Met `addTab(String naam, JComponent component)` voeg je tabs toe. Als je verwittigt wil worden wanneer de gebruiker van tab verandert, moet je een ChangeListener registreren.

5.7 Menu's

Om keuzemenu's te maken gebruik je de klasse JMenu. Met de `add` methode kan je hieraan een `JMenuItem` toevoegen. Deze menuitems zijn er in verschillende soorten (= subklassen):

- JMenuItem zelf: gedraagt zich zoals een JButton
- JRadioButtonMenuItem: gedraagt zich zoals een JRadioButton
- JCheckboxMenuItem: gedraagt zich zoals een JChechbox
- JMenu: de klasse JMenu is zelf ook een subklasse van JMenuItem. Dit laat toe om geneste menuutjes te maken

Daarnaast kan je ook nog een JSeparator toevoegen: dit is gewoon een lijntje om het menu visueel wat overzichtelijker te maken.

Je kan een JFrame een menubalk geven dmv. —JFrame.setJMenuBar(...)—. Aan een JMenuBar object kan je met de `JMenuBar.add(JMenu m)` methode dan verschillende menu's toevoegen.

Naast de gewone menu's bestaan er ook nog `JPopupMenu`s. Hieraan voeg je met de `add` methode JMenuItems toe. Met de methode `JPopupMenu.show(JComponent parent, int x, int y)` toon je dit popupmenu aan de gebruiker op locatie (x, y) tov. de linkerbovenhoek van de component *parent*. Typisch gebeurt dit zo:

```
1 JPopupMenu m = ...;
2 component.add(new MouseAdapter() {
3     public void mouseClicked(MouseEvent e) {
4       if (e.isPopupTrigger())
5         popup.show(e.getSource(), e.getX(), e.getY());
6     }
7   });
```

Om het werken met menu's gemakkelijker te maken, kan je aan een menuitem een "mnemonic" koppelen. Als de gebruiker dan het juiste menu geselecteerd heeft, kan hij het menuitem selecteren door de juiste toets in te drukken. Bijvoorbeeld:

```
1 JMenuItem help = new JMenuItem("Help");
2 help.setMnemonic(KeyEvent.VK_H);
```

Als de gebruiker de toets H indrukt, zal dit nu exact hetzelfde effect hebben als wanneer hij op dit item klikt.

Naast mnemonics, bestaat er ook "accelerators". Dit zijn toetsencombinaties die de gebruiker kan gebruiken om een menuitem te selecteren, zonder dat het menu in kwestie daarvoor actief moet zijn. Bijvoorbeeld:

```
1 help.setAccelerator(
2         KeyStroke.getKeyStroke(KeyEvent.VK_P, ActionEvent.CTRL_MASK));
```

5.8 Eigen componenten

Elke JComponent is ervoor verantwoordelijk om zichzelf te tekenen op het stukje van het scherm dat door zijn container voor hem voorzien wordt. De Swing omgeving zet dit proces in gang door op de JComponent in kwestie de methode `paintComponent(Graphics g)` op te roepen. Hierbij stelt het Graphics object het stukje van het scherm voor waarop die component zichzelf moet tonen. Dit object bevat methodes om meetkundige vormen te tekenen, strings te tonen en andere grafische operaties uit te voeren.

Elk soort van component implementeert natuurlijk deze paintComponent methode op zijn eigen manier. Als je zelf het uiterlijk van een component wilt bepalen, volstaat het dus om een subklasse te maken van een bestaande JComponent (bv. JPanel, of JButton als de component klikbaar moet zijn) en hierin de paintComponent methode te overschrijven. Afhankelijk van het resultaat dat je wil bereiken, kan het een goed idee zijn om hierin te beginnen met een `super.paintComponent(g)` oproep.

Het is goed om weten dat paintComponent heel vaak wordt opgeroepen: minstens elke keer als de gebruikers iets doet met de vensters op zijn scherm, en mogelijk nog vaker. Het is dus zelden een goed idee om hierin langdurige berekeningen te doen.

Zoals hierboven al uitgelegd, zal de view in een MVC applicatie zichzelf typisch moeten updaten als er iets verandert aan het model. Hierbij kan het soms handig zijn om het proces waarmee componenten opnieuw op het scherm getekend worden zelf in gang te zetten. Dit kan door de `repaint()` methode op te roepen op een component (of op een container waarin de component vervat zit). Deze methode zal de volledige component opnieuw tekenen. Als de component toevallig een container is, wordt er bij dit hertekenen gebruik gemaakt van de Layout die de component al had. Als er in de tussentijd aan de container nieuwe componenten werden toegevoegd en/of als er uit de container componenten werden verwijderd, moet ook nog de methode `revalidate()` worden opgeroepen, zodat de LayoutManager een nieuwe layout kan berekenen. Dit moet dan gebeuren voor de `repaint()` operatie. Anders kan het voorkomen dat bijvoorbeeld de nieuwe componenten niet of slechts gedeeltelijk getekend worden.