READER TCIF-V1OOP-19

OBJECT ORIENTED PROGRAMMING 1

PROPEDEUSE HBO-ICT

Cursuseigenaar: Bart.vanEijkelenburg@hu.nl
Auteurs: Bart van Eijkelenburg / André Donk
Reviewer:

Versie: 0.6 (01-02-2020)

VERSIEBEHEER

#	Wijzigingen	Door:
0.4	Les 3 nieuw en 11 aangepast	Bart
0.5	Les 7 en 8 aangepast	Andre
1.0	Voorlopig definitief (les 9 / 10 volgen)	Bart

INHOUDSOPGAVE

INLEIDING

De kern waardoor een computer zijn werk kan doen bestaat uit een ***processor*** plus ***geheugen***. Invoer- en uitvoer-apparatuur (toetsenbord, muis, monitor, printer etc.) vallen hierbij ook in de categorie "geheugen". Een (programma-)***opdracht*** (of ***statement***) is een voorschrift om **geheugen te veranderen**. Een ***computerprogramma*** bestaat uit een reeks opdrachten. In een computerprogramma gaat het dus om ***gegevens*** en ***bewerkingen*** (op die gegevens).

Die reeks opdrachten bestaat uit een aantal cryptische processor-instructies. Die zijn voor mensen niet zo goed leesbaar. Gelukkig is het mogelijk om in allerlei **programmeertalen** toch in een voor mensen leesbare taal een programma te programmeren. Een van deze talen is Python, waar je in het eerste blok kennis mee hebt gemaakt. Maar er zijn nog vele andere talen zoals C, C++, C#, PHP, Haskell, F#, Pascal, Delphi etc. Al deze talen hebben zo hun eigen voor- en nadelen. In deze cursus maken we kennis met de programmeertaal **Java**. We gebruiken Java omdat deze taal een van de meest gebruikte programmeertalen is[1], de taal platform-onafhankelijk is, en geschikt is voor zowel standalone, client-side als server-side applicaties. Het is echter niet de bedoeling dat je aan het einde van deze cursus alleen een **Java-ontwikkelaar** bent, maar dat je **programmeerconcepten** kent die je daarna in allerlei talen zou kunnen toepassen!

Bij de cursus Python-programmeren waren de computerprogramma's vooral gebaseerd op de **bewerkingen** (of ***procedures / functies***) die werden uitgevoerd met de gegevens (**data**). Men spreekt in dat geval van ***procedureel programmeren***. De procedures zijn daardoor relatief eenvoudig terug te vinden. De **gegevens** daarentegen zijn dan van ondergeschikt belang, en staan dus **verspreid** in je programma.

Anders dan Python is Java strikt **Object geOrienteerd**. Dat wil zeggen dat de **gegevens het uitgangspunt** zijn bij het schrijven van een computerprogramma. Op deze manier is het eenvoudiger om de werkelijke wereld (die je wilt automatiseren) in een model te vangen. Gegevens hebben namelijk onderling een relatie en deze kan je groeperen. Deze manier van programmeren heet ***Object geOrienteerd*** (En: Object Oriented; O.O.) programmeren. Het is de bedoeling dat bij O.O. programmeren de gegevens zodanig staan opgeslagen dat er relatief eenvoudig wijzigingen in de gegevens kunnen worden aangebracht, wanneer dat nodig blijkt te zijn (dus zonder het complete programma te moeten herschrijven). Het cruciale punt hierbij is een correct gebruik van ***klasse*** en ***object***. De begrippen ***klasse*** en ***object*** worden nogal eens op één grote hoop gegooid. Dan wordt het wel heel erg lastig om te begrijpen hoe O.O. programmeren in zijn werk gaat. We nemen klassen en objecten daarom als startpunt voor Les 1.

---

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, http://pypl.github.io/PYPL.html http://githut.info/, en https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

LES 1 – KLASSEN EN OBJECTEN

1. INLEIDING

Voordat we beginnen over klassen en objecten moeten we eerst weten hoe je begint met het schrijven van een Java programma. In Python maak je dan een nieuw Python-bestand en je kunt met je programma beginnen. Java dwingt je echter om iets meer gestructureerd te werk te gaan. Elk Java programma bevat daarom minstens 1 klasse en een methode '**main**':

```
public class Main {
  public static void main(String[] arg) {
    System.out.println("Hello world!");
  }
}
```

Listing 1: Het befaamde Hello World voorbeeld in Java

De code in deze listing begint met een klasse 'Main'. Deze klasse mag ook een andere naam hebben, maar een **klassenaam begint altijd met een hoofdletter**. De methode '`main`' moet wel per se zo heten en **methodenamen beginnen altijd met een kleine letter!** Ook moet een **main**-methode altijd een parameter hebben om extra informatie aan het programma mee te geven, in dit geval parameter '`arg`' (van 'arguments'). De keywords '`public`', '`static`' en '`void`' zullen we later behandelen.

ⓘ *In Python geef je met indentatie (inspringen door tabs) aan wat bij elkaar hoort, maar in Java staat alles wat in een klasse en methode bij elkaar hoort, binnen accolades: { en }.*

Als een programma geschreven is, moet het in Java eerst **gecompileerd** worden voordat je het kunt uitvoeren. Daarbij kijkt de **compiler** of er fouten in je programma zitten. Zo ja, dan volgt er een foutmelding. Zo niet, dan zet de compiler het programma om naar **bytecode**.

Bytecode is een voor mensen onleesbare code, maar deze code is wel geoptimaliseerd om met de **Java Virtuele Machine (JVM)** de vertaalslag naar de machinecode te maken die geschikt is voor jouw processor. Dat gebeurt op het moment dat je het programma uitvoert! De **JVM** is als het ware de kern van Java waar jouw programma op draait.

Dit programma print met de methode `System.out.println()` een string naar het console-scherm van je programma. In Java kun je een string altijd herkennen aan de dubbele aanhalingstekens. Dat levert in dit geval de als uitvoer:

```
Hello world!
```

➔ MAAK NU EERST OPDRACHT 1_1 UIT HET WERKBOEK!

2. DATA EN VARIABELEN

Nu we weten hoe we een eenvoudig programma moeten opbouwen, gebruiken we de onderstaande casus om verder kennis te maken met de programmeertaal Java:

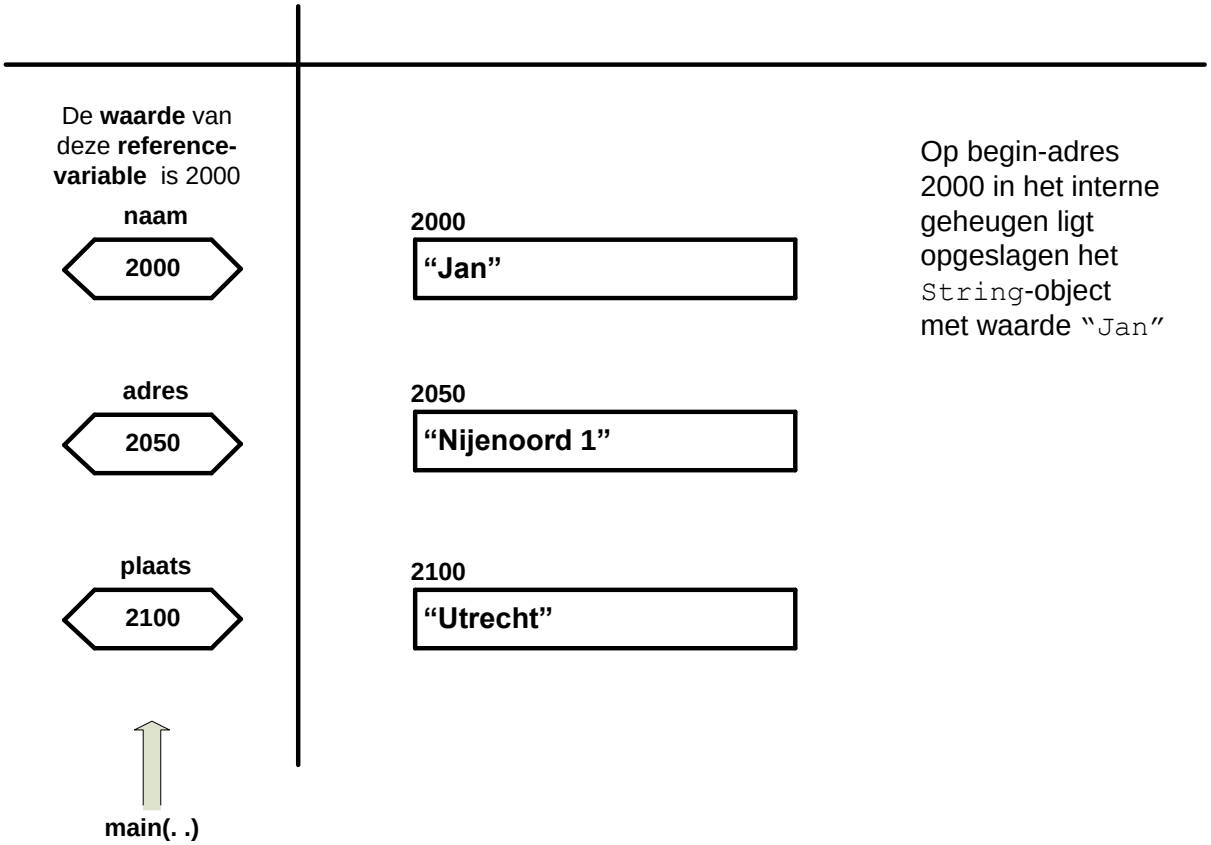
> De vader van Igor is bouwvakker en is sinds kort ZZP-er. Gelukkig loopt het bedrijfje al goed en heeft zijn vader al diverse klanten. De gegevens van de nieuwe klanten bewaart zijn vader nu nog in zijn mailbox. Igor (HBO-ICT student) is door zijn vader gevraagd om een applicatie te programmeren waar de NAW-gegevens van deze klanten in opgeslagen kunnen worden.

Igor begint met het aanmaken van variabelen waarin de naam van een klant kan worden opgeslagen. Net als in Python kun je ook gewoon variabelen in een Java programma declareren. Echter, in Java ben je **verplicht** om altijd van tevoren aan te geven wat voor **type** variabele je wilt. Dat heeft onder andere als voordeel dat Java bij het compileren voor je kan controleren of je wel de juiste gegevens in de variabele opslaat. Dat voorkomt fouten bij het uitvoeren van het programma. Igor begint met de volgende listing:

```
public class Main {
  public static void main(String[] arg) {
    String naam = "Jan";
    String adres = "Nijenoord 1";
    String plaats = "Utrecht";

    System.out.println(naam);
    System.out.println(adres);
    System.out.println(plaats);
  }
}
```

Listing 2: Er worden eerst variabelen gemaakt, maar de uitvoer blijft gelijk

In Java moet je onderscheid maken tussen de variabele-***naam*** (zoals bijvoorbeeld plaats) en zijn ***waarde*** (voor plaats is dat "Utrecht"). De waarde ("Utrecht") ligt opgeslagen in het interne geheugen, en is bereikbaar via zijn geheugenadres. Een waarde kan over meerdere geheugenadressen verspreid zijn, maar we spreken we in de volgende voorbeelden alleen over het begin-adres. In de afbeelding hieronder is dat het fictieve adres 2100.



Figuur 1: het interne geheugen van de computer tijdens het runnen van methode 'main'

We zeggen dat variabele plaats de waarde "Utrecht" heeft, maar in werkelijkheid is de waarde van plaats het (fictieve) begin-adres 2100. Dit begin-adres kan in Java niet worden opgevraagd (in de taal C kan dit wel). Omdat de interne werking van Java berust op deze begin-adressen, is er in deze reader voor gekozen om deze adressen in afbeeldingen soms zichtbaar te maken door een getal. Variabele plaats heet ook wel reference-variabele omdat de variabele verwijst (refereert) naar de waarde. Je moet de regel

```
String plaats = "Utrecht";
```

dan ook lezen als:

1. er is een reference-variabele met de naam plaats,
2. in die variabele plaats staat het begin-adres opgeslagen van een String. In de afbeelding is dat (het verzonnen adres) 2100.
3. de waarde, die staat opgeslagen in het geheugen, is "Utrecht"
4. in de afbeelding heeft het begin-adres van de String "Utrecht" een fictieve waarde gekregen: 2100. In werkelijkheid is dat begin-adres onbekend voor de programmeur.

Later zullen we zien dat String een klasse is, en "Utrecht" een object.

➔ MAAK NU EERST OPDRACHT 1_2 UIT HET WERKBOEK!

3. KLASSE KLANT

Met de aanpak van de vorige paragraaf moet voor elke klant afzonderlijk een set variabelen worden aangemaakt. De eerste klant heet Jan, en woont op Nijenoord 1 in Utrecht. Stel dat de tweede klant Wim heet en op de Oudenoord 340 in Utrecht woont, dan moeten voor die nieuwe klant 3 nieuwe variabelen gemaakt worden. Igor constateert hierbij **problemen**:

1. Er zijn dan 6 losse gegevens, terwijl er sprake is van 2 klanten, wat niet duidelijk is.
2. Als je nog een klant wilt opnemen moet je weer drie variabelen maken voor naam, adres en woonplaats, dit is onoverzichtelijk en kan erg foutgevoelig worden.

Als **oplossing** kiest Igor hier voor het ontwerpen van een OO-programma. Als je zo'n programma wilt ontwerpen, dan ga je altijd via een bepaalde aanpak te werk:

☛ Eerst bepaal je welke verschillende gegevens in de werkelijke wereld een rol spelen in het op te lossen probleem. Dan bepaal je welke gegevens bij elkaar horen. Voor elke gegevenseenheid (entiteit) wordt dan één klasse ontworpen. Een klasse kun je beschouwen als de "bouwtekening" voor een gegevensverzameling. Als je ontwerp klaar is, dan kan je dit omzetten naar programmacode in een OO-programmeertaal zoals Java.

Het gaat in de gegeven casus om de NAW-gegevens van klanten. Concreet zijn dat dus naam, adres en woonplaats. De gegevenseenheid (entiteit) die hierbij past is 'klant'. Een gegevensverzameling is dan bijvoorbeeld ("Jan", "Nijenoord 1", "Utrecht"), maar een andere verzameling kan zijn: ("Wim", "Oudenoord 340", "Utrecht"). Hoewel dus de waarden kunnen verschillen, worden van elke klant dezelfde eigenschappen vastgelegd.

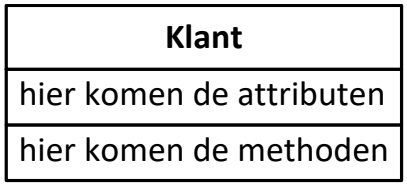
We hanteren de genoemde aanpak om de klasse Klant te ontwerpen. Het ontwerpproces is programmeertaal-onafhankelijk. Voor het ontwerpen gebruiken we de ontwerptaal **UML** (Unified Modelling Language). We beginnen met voorbeelden die slechts één klasse hebben. Later worden het er twee of meer. Uiteindelijk is het de bedoeling om een complete applicatie te kunnen modelleren en te programmeren. De overstap van één klasse naar twee klassen is lastig, maar de overstap naar meer dan twee klassen is daarna relatief eenvoudig.

Feitelijk heeft een klasse in een **UML-klassendiagram** altijd 3 onderdelen:

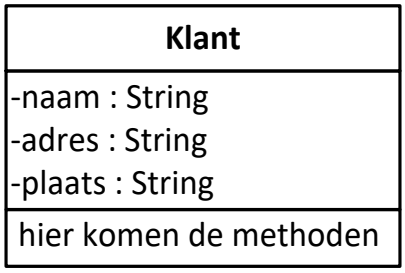
1. De naam van de klasse.
2. De gegevensvelden van (elk object van) de klasse.
3. De bewerkingen die kunnen worden uitgevoerd door (elk object van) de klasse.

ⓘ Een UML-diagram kun je maken in Microsoft Visio. Er zijn echter meer programma's waarmee dat kan. Je voor de programma's in deze eerste les ook volstaan met een schets op papier.

Een UML-klasse is een rechthoek met drie vakken. In het bovenste vak komt de naam van de klasse te staan. Voor de naam van de klasse is gekozen voor 'Klant' (de naam van een klasse begint altijd met een hoofdletter). In het middelste vak komen de gegevensvelden (of attributen) van de klasse te staan. In het onderste vak komen bewerkingen (of methoden):



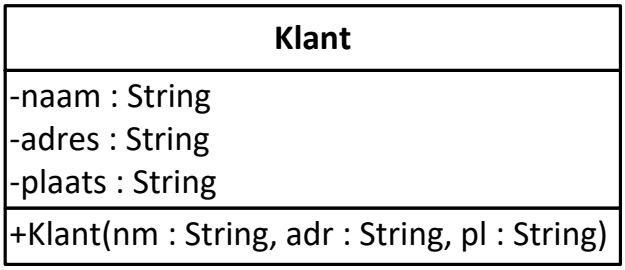
Uit de casus blijkt dat van elke klant de **naam**, het **adres** (straat & huisnummer) en de **woonplaats** moet worden vastgelegd. De ***attributen*** van de klasse zijn dus "**naam**", "**adres**" en "**plaats**". De attributen zijn van het type String.



In het UML-klassendiagram staat voor de attributen een **min**-teken. Dat is om aan te geven dat een attribuut '**private**' is. Zo'n attribuut kan alleen binnen die klasse gebruikt worden.

Als we concrete gegevens van een klant willen vastleggen met deze klasse, creëren we een **object**. Een object is te beschouwen als het "bouwwerk" dat wordt gebouwd op basis van de klasse (de "bouwtekening"). Eén object correspondeert met één set gegevens (gegevensverzameling) van een ding uit de werkelijkheid (dat kan zijn: gegevens van één klant, of van één student, of van één boek, of van één koe, of van één rekening, etc.).

Om objecten van het type 'Klant' te kunnen maken heb je in Java een **constructor** nodig. Een constructor is een bijzondere methode die exact dezelfde naam heeft als de klasse zelf. Verder moet een constructor parameters hebben voor alle gegevens die nodig zijn om een object van die klasse te maken! Daarom voegen we deze ook aan het klassendiagram toe:



Merk op dat voor deze constructor een **plus**-teken staat! De constructor is daarmee **public** gemaakt en ook beschikbaar buiten de klasse Klant. De parameters van de constructor zijn hier afkortingen van de attributen. Dat hoeft niet per se, maar het is gedaan om duidelijk te maken welke informatie je als eerste, tweede en derde parameter moet meegeven.

We kunnen nu het ontwerp omzetten naar Java-code! Daarvoor is het nodig om een bestand te maken met de naam 'Klant.java'. Stap voor stap moeten nu dan de onderdelen uit het ontwerp omgezet worden naar Java-code. Zie hieronder vast de code:

```
public class Klant                    // deze Java-klasse heet Klant
{
  private String naam;                // en heeft een attribuut "naam"
  private String adres;               // en een attribuut "adres"
  private String plaats;              // en een attribuut "plaats"

  public Klant(String nm, String adr, String pl)
  {                                   // en een contructor met 3 parameters
    naam = nm;                        // die in attributen naam,
    adres = adr;                      // adres en
    plaats = pl;                      // plaats worden opgeslagen!
  }
}
```

Listing 3: Klasse Klant, met alleen attributen

De eerste regel in de code van deze listing is weer de standaard manier om een klasse aan te maken. Daarachter staat een enkele regel commentaar, te herkennen aan een dubbele slash: //. Vervolgens staat de klasse beschreven tussen 2 accolades: { en }. Klasse Klant bestaat hier dus uit een drietal attributen en een constructor. Elk attribuut wordt voorafgegaan door het keyword '**private**' omdat in het ontwerp een **min**-teken voor de attributen staat. Let op dat de Java-notatie van attributen precies omgedraaid is ten opzichte van het UML-klassendiagram (dus eerst het type, en dan de naam van het attribuut).

De **constructor** wordt voorafgegaan door het keyword '**public**' omdat in het ontwerp een **plus**-teken staat. Ook de parameters van een constructor zijn weer precies omgedraaid ten opzichte van het UML-klassendiagram.

In de constructor zelf (ook weer tussen accolades) worden de parameters 'nm', 'adr' en 'pl' allen opgeslagen in de attributen van de klasse. Dat is noodzakelijk omdat deze parameters uit het geheugen verdwijnen zodra de constructor is uitgevoerd. Attributen blijven echter bestaan zolang het betreffende object in het geheugen van je computer staat. Helaas kan deze code nog niets, ook hier hebben we eerst een methode main nodig om ons programma te testen:

```
public class Main {
  public static void main(String[] arg) {
    Klant k1 = new Klant("Jan", "Nijenoord 1", "Utrecht");
    Klant k2 = new Klant("Wim", "Oudenoord 340", "Utrecht");

    // maak eventueel nog meer objecten aan van klasse Klant:
    Klant k3 =    . . .
  }
}
```

Listing 4: klasse Main maakt gebruik van klasse Klant uit listing 3

De Java-opdracht (binnen klasse Main):

```
Klant k1 = new Klant("Jan", "Nijenoord 1", "Utrecht");
```

wordt als volgt uitgevoerd:

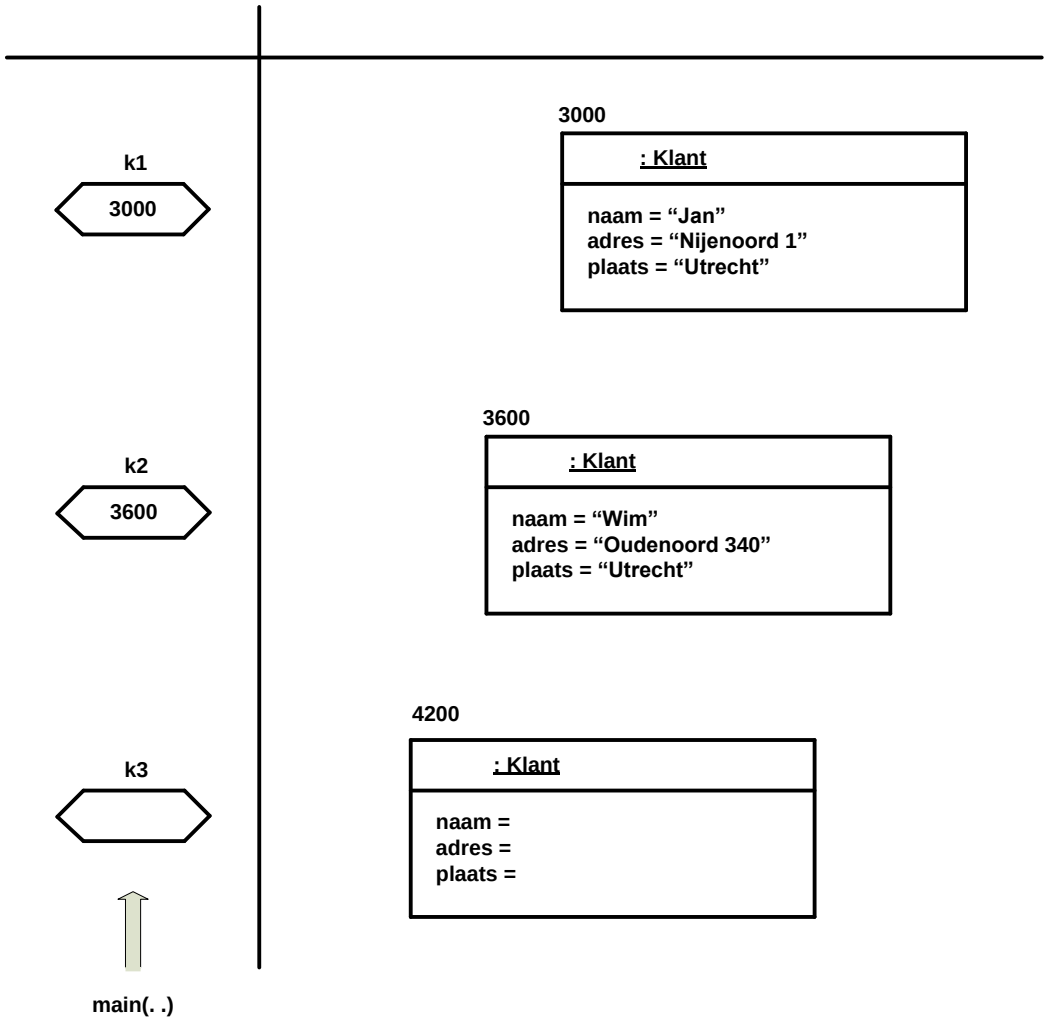
1. Er wordt in Java-klasse Klant gezocht naar de constructor (door het keyword '**new**').
2. Klasse Main "weet" dat de klant de naam "Jan" moet krijgen. Vanuit klasse Main wordt die naam "Jan" doorgegeven aan de constructor van klasse Klant. Dat gebeurt via de parameter met de naam nm en type String. De opdracht

   ```
   new Klant("Jan", "Nijenoord 1", "Utrecht");
   ```

   staat dus in klasse Main, maar deze opdracht zorgt er voor dat binnen constructor Klant() parameter nm de waarde "Jan" krijgt, dat parameter adr de waarde "Nijenoord 1" krijgt, en parameter pl de waarde "Utrecht".
3. Ten behoeve van dat ene object van klasse Klant wordt de constructor nu eenmalig uitgevoerd met nm="Jan", adr="Nijenoord 1" en pl="Utrecht". Daardoor ontstaat een **nieuw object met zijn eigen unieke attribuut-waarden**! Van dat object heeft attribuut *naam* de waarde van parameter nm ("Jan"); attribuut *adres* krijgt de waarde van parameter adr ("Nijenoord 1"); en attribuut *plaats* krijgt de waarde van parameter pl ("Utrecht");

   Zodra de sluit-accolade van de constructor -}- is bereikt, worden de waardes van de parameters nm, adr en pl weggegooid. Maar de waardes van de 3 attributen staan opgeslagen in het **nieuwe object** dat aanspreekbaar is **via de reference-variabele k1**. Zolang er in jouw programma een referentie is naar dit object, blijft het bestaan.

Het tweede object kan op vergelijkbare wijze aangemaakt worden. De toestand van het geheugen tijdens het uitvoeren van dit programma is op de volgende pagina weergegeven:



Figuur 2: het geheugen tijdens uitvoeren van het programma in listing 4

ⓘ We programmeren in dit voorbeeld nog niet het opslaan van de gegevens op (bijvoorbeeld) de harde schijf. Ook programmeren we nog geen GUI (Graphical User Interface) met Java zoals je dat met TKinter in Python zou kunnen doen. In een later stadium wil je de klasse Klant daarvoor natuurlijk wel gebruiken. Om nu toch de klasse Klant alvast te kunnen testen schrijven we nu dus eigenlijk kleine testprogramma’s in de klasse Main.

➔ MAAK NU EERST OPDRACHT 1_3 UIT HET WERKBOEK!

4. KLASSE KLANT UITGEBREID MET GETTERS EN METHODE TOSTRING()

De klasse Klant kun je nu gebruiken om objecten mee te maken, maar helaas kunnen we nu niet meer bij de gegevens die erin zijn gestopt. We kunnen de gegevens van een klant zelfs niet eens uitprinten. Als we bijvoorbeeld onder listing 4 de volgende code toevoegen;

```
System.out.println(k1.naam);
```

krijgen we een foutmelding omdat deze variabele **private** is! Het uitprinten van k1 kan wel:

```
System.out.println(k1);
```

maar zelfs dat helpt niet want de uitvoer zou dan ongeveer zijn:

```
Klant@659e0bfd
```

Dat is niet erg handig. Daarom krijgt de klasse Klant enkele methoden waarmee we het mogelijk maken om apart de naam, het adres of de woonplaats op te vragen van een klant. Je zou natuurlijk ook de attributen public kunnen maken, maar dat is niet de bedoeling. Binnen Object Oriëntatie speelt het begrip **information hiding** namelijk een belangrijke rol. De klasse Klant beschrijft welke gegevens (attributen) van iedere klant zullen worden vast gelegd. Om de vitale gegevens te beschermen, worden de attributen private (**hiding**) gemaakt. Dat betekent: uitsluitend toegankelijk en te wijzigen binnen de eigen klasse. Als we die attributen public zouden maken kan iedereen deze gegevens **lezen**, maar ook **wijzigen**!

Door aparte methoden te maken om informatie alleen op te halen (lezen), bescherm je dus je gegevens. Deze methoden noemen we '**getters**'. Een getter toont de inhoud van één attribuut. Wanneer je meerdere attributen tegelijk van een object wilt tonen, gebruik je methode **toString()**. Die methode gebruikt Java namelijk als je een Klant-object print.

Klant
-naam : String -adres : String -plaats : String
+Klant(nm : String, adr : String, pl : String) +getNaam() : String +getAdres() : String +getPlaats() : String +toString() : String

Als eerste gaan we bovenstaande klassendiagram omzetten naar code. Daarvoor breiden we de eerder gemaakte klasse Klant als volgt uit:

```
public class Klant {
  private String naam;
  private String adres;                    // de 3 attributen
  private String plaats;

  public Klant(String nm, String adr, String pl) {
    naam = nm;
    adres = adr;
    plaats = pl;
  }

  public String getNaam() {                // getter voor attribuut naam
    return naam;
  }

  public String getAdres() {               // getter voor attribuut adres
    return adres;
  }

  public String getPlaats() {              // getter voor attribuut plaats
    return plaats;
  }

  // toString() toont alle 3 de attributen van een Klant-object:
  public String toString() {
    String s = naam + " woont op " + adres + " in " + plaats;
    return s;
  }
}
```

Listing 5: Klasse Klant, na uitbreiding met getters en methode toString()

We schrijven ook een bijbehorende klasse Main die de nieuwe code test:

```
public class Main {
  public static void main(String[] arg){
    Klant k1 = new Klant("Jan", "Nijenoord 1", "Utrecht");
    System.out.println(k1.getNaam());                // publieke getter
    System.out.println(k1.getAdres());               // publieke getter
    System.out.println(k1.getPlaats());              // publieke getter
    System.out.println(k1.toString());               // toString()

    Klant k2 = new Klant("Wim", "Oudenoord 340", "Utrecht");
    System.out.println("Naam: " + k2.getNaam());
    System.out.println("Adres: " +k2.getAdres());
    System.out.println("Woonplaats: " + k2.getPlaats());
    System.out.println("klant nr 2: " + k2.toString());
  }
}
```

Listing 6: Klasse Main om de nieuwe klasse Klant te testen

Als we deze code uitvoeren levert dat een leesbare uitvoer op:

```
Jan
Nijenoord 1
Utrecht
Jan woont op Nijenoord 1 in Utrecht
Naam: Wim
Adres: Oudenoord 340
Woonplaats: Utrecht
klant nr 2: Wim woont op Oudenoord 340 in Utrecht
```

Let op: in methode toString() staat

```
String s = naam + " woont op " + adres + " in " + plaats;
```

De exacte waarde van s is afhankelijk van het object waarop toString() wordt aangeroepen:

voor **k1**.toString() wordt s: `"Jan woont op Nijenoord 1 in Utrecht"`
maar voor **k2**.toString() wordt s: `"Wim woont op Oudenoord 340 in Utrecht"`

Dit principe is geldig voor alle methoden van een object. Getters leveren dus ook alleen de waarde van het betreffende attribuut voor het object waarop ze worden aangeroepen:

voor **k1**.getAdres() is de return-waarde: `"Nijenoord 1"`
maar voor **k2**.getAdres() is de return-waarde: `"Oudenoord 340"`

➔ MAAK NU EERST OPDRACHT 1_4 UIT HET WERKBOEK!

5. KLASSE KLANT UITGEBREID MET SETTERS

Klanten kunnen verhuizen. En dan moet je dat nieuwe adres kunnen opslaan. Voor variabelen die naderhand nog kunnen wijzigen kan je een set-methode aanmaken, ook wel 'setter' genoemd. Een setter is methode waarmee je een attribuut een nieuwe waarde kunt geven via een parameter.

Klant
-naam : String -adres : String -plaats : String
+Klant(nm : String, adr : String, pl : String) +getNaam() : String +getAdres() : String +getPlaats() : String **+setAdres(nwAdr : String) : void** **+setPlaats(nwPl : String) : void** +toString() : String

Methode **setAdres(String nwAdr)** zorgt ervoor dat attribuut adres een nieuwe waarde krijgt, via parameter **nwAdr**. Deze methode doet iets (wijzigen van de variabele), maar hij heeft geen return-waarde. Ook methode **setPlaats(String nwPl)** heeft een parameter (**nwPl**) voor het instellen van de nieuwe woonplaats. In listing 7 is dit klassendiagram omgezet naar Javacode.

```
public class Klant {
  private String naam;
  private String adres;                          // de 3 attributen
  private String plaats;

  public Klant(String nm, String adr, String pl) {
    naam = nm;
    adres = adr;
    plaats = pl;
  }

  public String getNaam()   { return naam; }
  public String getAdres()  { return adres; }
  public String getPlaats() { return plaats; }

  public void setAdres(String nwAdr) {               // publieke setter
    adres = nwAdr;
  }

  public void setPlaats(String nwPl) {               // publieke setter
    plaats = nwPl;
  }

  public String toString() {
    String s = naam + " woont op " + adres + " in " + plaats;
    return s;
  }
}
```

Listing 7: Klasse Klant, nu met setters voor adres en plaats

Merk op dat de setters **geen return-statement** bevatten! En in plaats van een return-type staat er nu in de methode-declaratie **void**. Hierdoor weet de Java-compiler dat gecontroleerd moet worden of deze methode inderdaad geen return-waarde oplevert!

ⓘ *Om ruimte in het dictaat te besparen zijn de getters op 1 regel geschreven. Dit mag in Java. Maar voor de leesbaarheid van je code kan je de code beter gewoon uitschrijven!*

De werking van de nieuwe setters wordt in listing 8 gedemonstreerd:

```
public class Main {
  public static void main(String[] arg){
    Klant k1 = new Klant("Jan", "Nijenoord 1", "Utrecht");
    k1.setAdres("Vreeburg 38");
    System.out.println("klant nr 1:" + k1.toString());
    Klant k2 = new Klant("Wim", "Oudenoord 340", "Utrecht");
    k2.setPlaats("Amsterdam");
    System.out.println("klant nr 2: " + k2.toString());
  }
}
```

Listing 8: Klasse Main maakt een klant aan met het oude adres en wijzigt dit dan. De uitvoer bewijst de correcte werking van de getters.

Uitvoeren van de klasse Main levert de volgende uitvoer op, waarbij de adressen succesvol zijn gewijzigd:

```
klant nr 1: Jan woont op Vreeburg 38 in Utrecht
klant nr 2: Wim woont op Oudenoord 340 in Amsterdam
```

ⓘ *Je kunt je natuurlijk afvragen waarom je een attribuut niet gewoon publiek toegankelijk maakt als je zowel een set- als get-methode hebt voor een attribuut. Het is echter vaak zo dat je de waarde van een attribuut wel wilt kunnen instellen, maar dat je ook wilt afdwingen dat sommige waarden niet mogelijk zijn. Een variabele voor het bijhouden van een leeftijd mag bijvoorbeeld geen negatieve waarden bevatten. Als een attribuut publiek toegankelijk is, kan je dat niet meer afdwingen. In methoden kan je die controles wel inbouwen, maar dat is hier achterwege gelaten om de voorbeelden eenvoudig te houden.*

➔ MAAK NU EERST OPDRACHT 1_5 UIT HET WERKBOEK!

6. RICHTLIJNEN BIJ HET ONTWERPEN VAN KLASSEN

- De naam van iedere klasse is een zelfstandig naamwoord-in-het-enkelvoud, en deze naam begint met een hoofdletter. De klassenaam staat in het midden van het bovenste vakje. Let op: "zelfstandig naamwoord" betekent in het bijzonder dat de klassenaam GEEN WERKWOORD MAG bevatten.
- De naam van ieder attribuut begint altijd met een kleine letter. Bij langere namen begint iedere volgend deel met een hoofdletter. De attribuutnamen staan in het middelste vakje van het klassendiagram. Voorbeelden van correcte attribuutnamen zijn nummer, naam, reisNaam, reisCode, boekingsDatum, beginDatum, eindDatum etc.
- Attributen zijn bij ons vrijwel altijd "private". Dat is vanwege de "information hiding"; niet iedereen mag zomaar bij de object-informatie. Notatie van "private" is een - teken.
- De naam van iedere methode begint altijd met een kleine letter. Er is slechts één uitzondering: de naam van **constructoren** MOET hetzelfde zijn als de klassenaam, en DUS moet de naam van iedere constructor beginnen met een hoofdletter. Bij langere methode-namen begint iedere volgend deel met een hoofdletter, net zoals bij attribuut-namen het geval is. Iedere methode-naam MOET ronde haakjes () bevatten, en binnen die ronde haakjes staan de parameters vermeld. De methode-namen staan in het onderste vakje van het klassendiagram. Voorbeelden van correcte methode-namen zijn getNummer(), getNaam(), setNaam(String), getReisNaam(), getReisCode(), getBoekingsDatum(), setBoekingsDatum(Date), getBeginDatum(), getEindDatum() etc.

LES 2 – KLASSEN, OBJECTEN EN OPERATIES

1. INLEIDING

In les 1 heb je kunnen zien hoe je een UML-klassendiagram maakt en hoe je deze omzet naar Javacode. De klassen die we in dat hoofdstuk maakten waren echter nog niet zo geavanceerd. Het enige wat je ermee kon was er informatie in opslaan en die informatie middels getters, setters en methode toString() opvragen en wijzigen.

Les 2 is een vervolg van les 1, maar we maken naast eenvoudige setters en getters ook meer intelligente methoden. Dit zijn methoden waarin je ook bewerkingen en controles met en op attributen of parameters kunt uitvoeren.

2. VAN UML NAAR CODE

Net als in les 1 beginnen we met een UML-klassendiagram. Het eerste voorbeeld betreft klasse Rekening. Klasse Rekening kun je gebruiken om Rekening-objecten mee te maken.

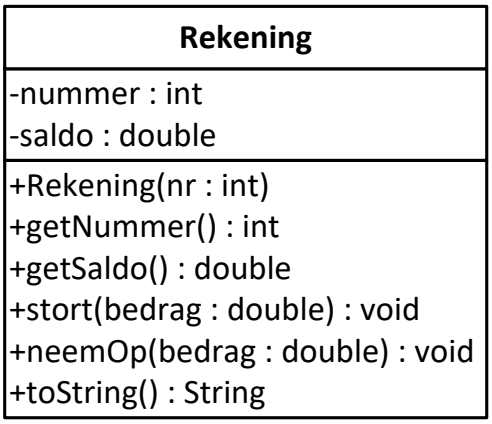
Klaas Knot werkt op de IT-afdeling van Knab, een relatief jonge bank. De bank wil een nieuwe rekening introduceren en Klaas is gevraagd om dit te implementeren in het centrale systeem van de bank.

De nieuwe rekening is een betaalrekening en heeft de volgende kenmerken:

- De klant kan zelf een nummer kiezen voor de nieuwe rekening.
- Je kunt onbeperkt rood staan op de rekening.
- Je kunt storten en opnemen op de rekening.
- Je moet een overzicht van de rekening kunnen opvragen.

Klaas begint met het ontwerpen van een klassendiagram. Hij heeft de klasse zo eenvoudig mogelijk gehouden. Elk Rekening-object heeft een nummer (een int-getal) en een saldo (een double-getal: een floating-point ofwel in goed Nederlands een komma-getal).

Verder heeft de constructor één parameter: voor het rekeningnummer. Er is dus geen parameter voor het saldo! Wanneer een nieuwe rekening wordt geopend, wordt het begin-saldo gelijk aan 0.0.



Met deze constructor kunnen we dus nieuwe rekeningen maken:

```
Rekening r1 = new Rekening(12345678);
Rekening r2 = new Rekening(13578642);
Rekening r3 = new Rekening(33444555);
```

Deze rekeningen beginnen dan met een saldo van 0 euro. Je kunt echter ook geld storten op een rekening; en je moet geld kunnen opnemen van een rekening. Dus worden de volgende bewerkingen mogelijk:

```
r1.neemOp(45.88);
r2.stort(250.00);
r2.neemOp(210.00);
r2.neemOp(50.00);
r3.stort(5987654.98);
```

Door geld te storten verander je de waarde van attribuut saldo. Een setter wijzigt alleen de waarde van attribuut naar een nieuwe waarde, maar de methoden stort(.) en neemOp(.) moeten de nieuwe waarde van saldo eerst uitrekenen. Dit is dus meer dan alleen een set-methode, daarom heeft de methode ook geen 'set' in de methodenaam.

In les 1 was te zien dat elke methode (behalve de constructor) bestaat uit drie onderdelen:

1. De methodenaam (vaak een werkwoord): public void **stort**(double bedrag) {
2. 0 of meer parameters: public void stort(**double bedrag**) {
3. Het return-type: public **void** stort(double bedrag) {

ⓘ *Parameters gebruik je als er invoer nodig is van buitenaf, in dit geval het 'bedrag'.*
ⓘ *Return-type 'void' gebruik je wanneer de methode geen uitvoer heeft.*
ⓘ *Als er wel uitvoer is, dan is het return-type het type van de uitvoer (String, int, double).*
ⓘ *Constructoren hebben geen returntype (dus ook niet void).*

Het type **int** gebruik je bij gehele getallen, zoals 0, 1, 1765 en -8745. Maar als er sprake is van gebroken getallen, zoals 0.0 of 34.0192 of -21987.665593 gebruik je het type **double**. In Javacode schrijf je doubles **altijd** met een punt, zoals in Engelstalige landen gebruikelijk is.

Het type **String** gebruik je voor één of meer tekens van het toetsenbord met dubbele aanhalingstekens ervoor en erachter zoals "A", "Jan", "12345678", "Laan 19". Als je enkele aanhalingstekens gebruikt is dat binnen Java een **char**! Deze kan precies **één** teken bevatten zoals 'A', '8', '$' etc.

We kunnen nu de klasse Rekening programmeren. Zie listing 9 voor het resultaat.

```
public class Rekening {
  private int nummer;                    // de 2 attributen
  private double saldo;

  // de constructor
  public Rekening(int nr) {
    nummer = nr;
  }

  public int getNummer()    { return nummer; }
  public double getSaldo() { return saldo; }

  // "echte" methoden:
  public void stort(double bedrag) {
    saldo = saldo + bedrag;              // saldonieuw = saldooud + bedrag
  }

  public void neemOp(double bedrag) {
    saldo = saldo - bedrag;              // saldonieuw = saldooud - bedrag
  }

  public String toString() {
    String s = "Op rekening " + nummer + " staat " + saldo;
    return s;
  }
}
```

Listing 9: De klasse Rekening in Javacode

Bij storten en opnemen wordt eerst uitgerekend wat rechts van het = teken staat. Daarna wordt het resultaat van die berekening opgeslagen in attribuut **saldo**. Dit kunnen we testen:

```
public class Main {
  public static void main(String[] arg){
    Rekening r1 = new Rekening(12345678);
    Rekening r2 = new Rekening(13578642);
    Rekening r3 = new Rekening(33444555);

    r1.neemOp(45.88);
    r2.stort(250.00);
    r2.neemOp(210.00);
    r2.neemOp(50.00);
    r3.stort(5987654.98);

    System.out.println(r1.toString());
    System.out.println(r2.toString());
    System.out.println(r3);
  }
}
```

Listing 10: De klasse Main om de klasse Rekening te testen

Bij uitvoeren krijgen we het volgende resultaat:

```
Op rekening 12345678 staat -45.88
Op rekening 13578642 staat -10.00
Op rekening 33444555 staat 5987654.98
```

De klasse Main van listing 10 bevat drie keer een print-statement:

```
    System.out.println(r1.toString());
    System.out.println(r2.toString());
    System.out.println(r3);
```

De derde regel print alleen het object. Java controleert dan zelf of er een methode toString() is en print deze uit. Zo niet, dan krijg je geen foutmelding maar print Java een standaard waarde uit. De andere regels printen altijd het resultaat van de methode toString().

In de uitvoer maakt dat geen verschil, maar de korte versie is beter dan de lange, omdat de programmeur bij de korte versie niet hoeft te beveiligen tegen een evt. NullPointerException. Bekijk deze code:

```
    Rekening r4 = null;
    System.out.println(r4.toString());
```

Hier heeft variabele r4 geen waarde (null in Java). Wanneer we deze code zouden uitvoeren krijgen we een foutmelding omdat Java zoekt naar methode toString() in een niet-bestaand object. Veiliger is dan:

```
    Rekening r4 = null;
    System.out.println(r4);
```

Op deze manier krijgen we geen foutmelding maar zal er 'null' geprint worden.

3. VAN CODE NAAR UML

We hebben in de voorgaande voorbeelden steeds gekeken hoe je een klassendiagram naar Javacode kunt omzetten. In de praktijk is het echter ook belangrijk om uit Javacode het model te kunnen begrijpen.

Henk Stevens is sinds enkele weken gedetacheerd bij Software.com. Zijn leidinggevende, Chris, gaat echter niet zo gestructureerd te werk. Als hij een plan heeft programmeert hij een (incompleet) voorbeeld van wat hij ongeveer wil hebben. Andere medewerkers moeten dan het idee van Chris uitwerken. Chris noemt dat Test Driven Design. Henk heeft de opdracht om de bestaande software te documenteren. Hij begint met een van de testscripts van Henk.

Henk krijgt de tests in de vorm van een klasse Main:

```
public class Main {
  public static void main(String[] arg) {
    Product pr1 = new Product("iPad", "1234AB", 614.00);
    pr1.setBTW(19.0);
    pr1.verhoogPrijsMet(2.5);
    System.out.println("Eerste product: " + pr1);

    Product pr2 = new Product("Paracetamol", "5678CD");
    pr2.setPrijs(1.90);
    pr2.setBTW(6.0);
    pr2.verhoogPrijsMet(-10);
    System.out.println("Tweede product: " + pr2);
    System.out.println();
    System.out.println("BTW percentage: " + pr2.getBTW());
    System.out.println("Betaalde btw: " + pr2.btwBedrag());
    System.out.println("Prijs: " + pr2.getPrijs());
  }
}
```

Listing 11: De klasse Main die Henk heeft geschreven om zijn klasse Product te demonstreren.

De meegeleverde uitvoer waaraan Henk moet voldoen is ook gegeven:

```
Eerste product: iPad heeft code 1234AB en kost 629.35; de btw is 19.0%
Tweede product: Paracetamol heeft code 5678CD en kost 1.71; de btw is 6.0%

BTW percentage: 6.0
Betaalde btw: 0.1026
Prijs: 1.71
```

In dit geval moet je de werking van klasse Product kunnen doorgronden door de uitvoer en de code van klasse Main te bestuderen. Dat doen we door de volgende vragen te stellen:

1. Welke constructors worden gebruikt om Product-objecten te maken?
2. Welke attributen bevat een Product-object minimaal?
3. Welke methoden worden er op de Product-objecten aangeroepen?
4. Welke informatie hebben deze methoden nodig of leveren ze op?
5. Welke extra methoden of attributen zijn afleidbaar uit de code of uitvoer?

Een deel van deze vragen kun je beantwoorden aan de hand van de klasse Main, terwijl je voor andere vragen ook de uitvoer nodig hebt. We behandelen elke vraag afzonderlijk en komen na het beantwoorden van deze vragen met een UML-klassendiagram van de klasse Product.

1. **Welke constructors worden gebruikt om Product-objecten te maken?**
   Voor een antwoord op deze vraag moeten we op zoek gaan waar in klasse Main de opdracht '**new Product(…)**' gegeven is. We vinden deze opdracht in de volgende vormen terug:

```
Product pr1 = new Product("iPad", "1234AB", 614.00);
Product pr2 = new Product("Paracetamol", "5678CD");
```

   Er zijn dus 2 Product-objecten. Maar de eerste aanroep is anders dan de tweede. Bij de eerste opdracht worden 3 parameters meegegeven (string, string, double), terwijl de tweede opdracht alleen 2 string-parameters heeft. We onderscheiden dus deze constructors:

   Product(String, String, double)
   Product(String, String)

2. **Welke attributen bevat een Product-object minimaal?**
   Bij de vorige vraag hebben we gezien dat er 2 Product-objecten werden gemaakt, maar het is nog niet duidelijk wat de betekenis is van de parameters die werd meegegeven. We kunnen uit de uitvoer al wel opmaken dat "ipad" en "paracetamol" kennelijk productnamen zijn. En "1234AB" en "5678CD" zijn productcodes. De waarde 614.00 moet volgens de uitvoer de prijs van een product zijn. Aangezien deze gegevens geprint kunnen worden nadat deze objecten zijn aangemaakt, moet de conclusie zijn dat klasse Product minimaal 3 attributen moet bevatten om deze gegevens op te slaan:

   - naam : String
   - code : String
   - prijs : double

3. **Welke methoden worden er op de Product-objecten aangeroepen?**
   De twee producten die we bij vraag 1 hebben ontdekt zijn ook hier het uitgangspunt om de vraag te kunnen beantwoorden. We moeten voor de methoden op zoek gaan naar alle voorkomens van "**pr1.**" en "**pr2.**". Iedere keer dat je pr1 gevolgd door een punt in de code ziet staan zal een methode worden aangeroepen. We vinden dan de volgende regels code:

```
pr1.setBTW(19.0);
pr1.verhoogPrijsMet(2.5);
pr2.setPrijs(1.90);
pr2.setBTW(6.0);
pr2.verhoogPrijsMet(-10);
System.out.println("BTW percentage: " + pr2.getBTW());
System.out.println("Betaalde btw: " + pr2.btwBedrag());
System.out.println("Prijs: " + pr2.getPrijs());
```

Deze methoden kunnen aangeroepen worden op een Product-object, dus die moeten in de klasse Product aanwezig zijn!

4. **Welke informatie hebben deze methoden nodig of leveren ze op?**
   Nu we weten welke methoden er zijn, kunnen we aan de manier waarop ze zijn gebruikt ook afleiden welke invoer (aantal parameters en type) een methode heeft en wat het resultaat is (return-type):

setBTW(**double**)	: **void** want het is een setter
verhoogPrijsMet(**double**)	: **void** geen type afleidbaar uit code/uitvoer
setPrijs(**double**)	: **void** want het is een setter
getBTW()	: **double** want de BTW-setter accepteert doubles
btwBedrag()	: **double** want de uitvoer is een double
getPrijs()	: **double** want de uitvoer is een double

5. **Welke extra methoden of attributen zijn afleidbaar uit de code of uitvoer?**
   Uit de methoden van vraag 3 en 4 blijkt dat er ook sprake is van een extra attribuut btw! Ook de uitvoer laat zien dat er per Product-object een verschillend BTW-percentage ingesteld kan worden. Dit leidt tot een extra attribuut: **btw**! Het type van dit attribuut is **double** omdat de parameter van methode setBTW(…) een double is.

   Tot slot kan een Product-object uitgeprint worden, wat dus wil zeggen dat er een **toString()** methode in klasse Product aanwezig MOET zijn.

Nu alle attributen, constructors en methoden bij elkaar worden gevoegd kan het UML-klassendiagram opgesteld worden. Deze kan vervolgens omgezet worden naar Javacode.

Product
-naam : String -code : String -prijs : double -btw : double
+Product(nm : String, cd : String, pr : double) +Product(nm : String, cd : String) +setBTW(bt : double) : void +setPrijs(pr : double) : void +getBTW() : double +getPrijs() : double +verhoogPrijsMet(extra : double) : void +btwBedrag() : double +toString() : String

Let op: methode '**verhoogPrijsMet(…)**' heeft double-parameter, maar uit de uitvoer valt op te maken dit een verhogings**percentage** betreft (geen verhogingsbedrag).

Evenzo is in de uitvoer zichtbaar dat methode '**btwBedrag()**' het bedrag oplevert je aan BTW kwijt bent. De Product-prijs is dus al **inclusief** BTW!

Zie listing 12 voor klasse Product.

```java
public class Product {
  private String naam;
  private String code;
  private double prijs;
  private double btw;

  public Product(String nm, String cd, double pr) {
    naam = nm;
    code = cd;
    prijs = pr;
  }

  public Product(String nm, String cd) {
    naam = nm;
    code = cd;
  }

  public void setBTW(double bt)   { btw = bt; }
  public void setPrijs(double pr) { prijs = pr; }
  public double getBTW()   { return btw;   }
  public double getPrijs() { return prijs; }

  public void verhoogPrijsMet(double extra) {
    prijs = prijs + prijs * (extra/100);
  }

  public double btwBedrag() {
    return prijs * (btw/100);
  }

  public String toString() {
    String s = naam + " heeft code " +code+ " en kost ";
    s = s + prijs+ "; de btw is " +btw+ "%";
    return s;
  }
}
```

Listing 12: De uiteindelijke klasse Product die hoort bij listing 11

➔ MAAK NU EERST OPDRACHT 2_1 UIT HET WERKBOEK!

4. RICHTLIJNEN BIJ HET SCHRIJVEN VAN METHODEN

- Elke concrete methode heeft een methodenaam, een return-type (dit geldt niet voor constructoren), 0, 1, 2, .. parameters en een implementatie of body.
- Iedere methode heeft een return-type (een typenaam of void). Er is slechts één uitzondering: **een constructor** (eigenlijk een speciale methode) heeft **geen** return-type.
- Return-type void betekent dat de methode iets doet: setters hebben altijd return-type void: een setter wijzigt de waarde van een attribuut ("doet iets"), maar geeft niets terug.
- Als het return-type een typenaam is, moet de methode een waarde teruggeven via het return-statement. Voor een methode met een typenaam als return-type wordt de volgende strategie aanbevolen:

```
public double inhoud() {
  double antw;
  //later te programmeren hoe de inhoud wordt berekend
  return antw;
}
```

- Getters hebben altijd een typenaam als return-type, omdat bijvoorbeeld getter getNummer() de waarde teruggeeft van nummer, en dat is een int.
- Wanneer een methode geen invoer-van-buiten nodig heeft, dan heeft hij geen parameters. Getters tonen de inhoud van een een attribuut, dus geen parameter
- Als een methode wel invoer-van-buiten nodig heeft, dan heeft hij parameter(s). Setters wijzigen de inhoud van een attribuut, en hebben dus een parameter, want de nieuwe attribuut-waarde moet worden binnengebracht.
- System.out.println(van iets) hoort (meestal) niet thuis in de body van methoden van een domeinklasse. Afdrukken op scherm is een taak van Main. **Dus niet:**

```
public void speak() {
  System.out.println("kwaak kwaak");
}
```

Maar:

```
public String speak() {
  return "kwaak kwaak";
}
```

Als je programmeert streef je naar hergebruik van code. Als je print in een methode, kan je deze methode niet meer (her)gebruiken in een GUI. Daarom printen we alleen maar in de methode main, waar het programma alleen gericht is op de console. Om dezelfde reden liever niet een "\n" (newline) of "\t" (tab) in een toString() methode.

LES 3 – SOFTWARE TESTEN & EXCEPTION HANDLING

1. INLEIDING

We hebben nu de afgelopen twee lessen gekeken naar het bouwen van software. We hebben ook steeds een main-methode geschreven waarin we wat functionaliteiten van de geschreven klassen hebben gebruikt. Daarmee kan je controleren of de klassen die je geschreven hebt, ook doen wat je bedacht had.

In de praktijk is deze manier van testen niet de handigste. Daarom kijken we deze les hoe je dat op een gestructureerdere wijze kunt aanpakken. De meest gebruikelijke manier is om daar in je code een test-framework voor in te zetten. Wij zullen daar JUnit voor gebruiken.

Voordat we naar de technische kant van het verhaal kijken, bespreken we eerst wat testen nu precies is, en waarom het belangrijk is om dit op een gestructureerde manier toe te passen in je programma.

Sommige (beginnende en gevorderde) ontwikkelaars vinden het testen niet interessant en besteden hier niet veel tijd aan. In dat licht is het goed om kennis te nemen van de situaties in de kaders in dit hoofdstuk waarin het goed mis ging.

2. TESTEN: WAT EN WAAROM?

Als je software test, dan controleer je of het systeem voldoet aan de eisen (requirements) die aan het programma gesteld zijn. Eisen kan je op meerdere niveau's stellen. Je kunt bijvoorbeeld eisen dat het programma een bepaalde reactiesnelheid heeft. Dit noemen we de **niet-functionele** requirements.

Daarnaast worden er eisen gesteld op het gebied van de functionaliteit van het programma. Bij het belastingaangifteprogramma moet een gebruiker bijvoorbeeld zijn of haar inkomstenbronnen kunnen opgeven. Als daar niets voor geprogrammeerd is, dan voldoet het programma niet aan alle **functionele requirements**.

Binnen een bepaalde groep requirements kan je ook weer onderscheid maken waarop je precies gaat controleren. De ISO 25010 standaard voor software kwaliteit schaart onder functionele geschiktheid de volgende categoriën:

1. Functionele compleetheid (is aan alle requirements voldaan)
2. Functionele correctheid (wordt daarmee het beoogde doel bereikt)
3. Functionele toepasselijkheid (helpt het ook in de praktijk of is het omslachtig)

Het door onderzoek controleren (testen) of het programma voldoet aan de gegeven requirements noemen we de **verificatieslag**. Deze slag heeft diverse doelstellingen:

1. Verifiëren dat voldaan is aan alle requirements.

2. Voorkomen van fouten in of door het product.
3. Controleren of de software ‘af’ is.
4. Creëren van vertrouwen in het product.
5. Het verhogen van de kwaliteit van de software.
6. Het geven van inzicht in de kwaliteit van de software.
7. Voldoen aan standaarden die gelden voor software.

Je ziet hier dat om aan deze doelstellingen te kunnen voldoen, ook de rapportage van de resultaten van deze verifcatieslag belangrijk is. Testen is dus niet een soort ‘debuggen’ van je code. Een test toont op een gestructureerde en herleidbare wijze aan dat er een fout in de software zit. Debuggen kan je vervolgens gebruiken om de oorzaak op te sporen.

Overigens hoeven niet tijdens elke verificatieslag alle doelstellingen een even grote rol te spelen. In de eindfase van een project kan het bijvoorbeeld heel belangrijk zijn om te kunnen aangeven hoe goed het product scoort tijdens testen, om potentiële klanten te kunnen overhalen het product te kopen. In de beginfase (van een lang ontwikkeltraject) kan dat minder relevant zijn.

De eerste testvlucht van de Ariane 5 raket vond plaats op 4 juni 1996. Bij lancering trad er een integer-overflow op, wat (via een kettingreactie) leidde tot het in werking stellen van het zelfvernietigingsmechanisme.

De directe oorzaak van de crash bleek het niet passen van een 64 bits floating point getal in een 16 bits integer. Achteraf bleek dat tijdens het testen niet altijd met de daadwerkelijke hardware getest werd, maar met simulaties.

Natuurlijk kan je nooit alle software-failures voorkomen, omdat de systemen dermate complex zijn, dat niet alle mogelijke situaties voorzien kunnen worden. Maar het toepassen van tests en testtechnieken kan wel helpen om de frequentie waarin deze problemin zich voordoen, te verminderen.

Best practices zijn bijvoorbeeld om softwaretesters nauw te betrekken bij het reviewen van requirements. Zodoende kunnen fouten en/of onduidelijkheden in een vroeg stadium worden verholpen. Dit is natuurlijk ook van toepassing op andere fases in een project. Als een tester meekijkt met de softwarearchitecten en softwareontwikkelaars, begrijpt hij beter hoe een systeem in elkaar zit, en kan hij ook beter testen.

Een andere best practice is dat het systeem grondig getest moet worden VOORDAT de software gereleased kan worden. Dit lijkt logisch, maar door tijdsdruk wil het nog weleens gebeuren dat dit erbij inschiet, met soms vervelende gevolgen!

Als je software test, houd dan de volgende uitgangspunten in je achterhoofd;

1. Een test kan alleen de **aanwezigheid** van fouten aan tonen, niet hun **afwezigheid**!
2. Een test kan nooit 100% alle voorkomende situaties dekken, stel prioriteiten.
3. Begin zo vroeg mogelijk in het ontwikkeltraject met testen, dit scheelt tijd + geld.
4. Fouten clusteren vaak samen. Detecteer deze componenten en geef ze prioriteit!

Een illustratie te geven bij uitgangspunt 1; stel dat een stuk code een fout bevat, dan hoeft het niet altijd zo te zijn dat er elke keer een error optreedt als de code uitgevoerd wordt. Het kan zijn dat de waarden van variabelen zodanig zijn dat het de error niet voorkomt.

Stel dat je bijvoorbeeld jongeren onder 18 jaar korting wilt geven. Onderstaande code geeft echter personen van exact 18 jaar ook nog korting. Dat zul je niet detecteren als je alleen controleert op een leeftijd van 17 en 25:

```
double bedrag = 100;
if (leeftijd <= 18)
    bedrag /= 2;
```

Anderzijds (uitgangspunt 2) heeft het ook geen nut als we hier leeftijden 0 t/m17 allemaal zouden controleren. Als het werkt voor 15, dan vast ook voor 14. Je moet dus prioriteiten stellen, en goed nadenken over bijvoorbeeld grenswaarden.

Het is goed om te bedenken dat fouten in de code allerlei oorzaken kunnen hebben. Zojuist hebben we tijdsdruk al genoemd, maar onervaren of incompetente teamleden kunnen ook een bron van fouten zijn. En als de teamleden individueel wel competent zijn, kan gebrekkige onderlinge communicatie alsnog roet in het eten gooien.

Naast direct menselijke oorzaken is in veel gevallen complex ontworpen code, of onbekende techniek ook een reden waardoor er fouten in de code kunnen sluipen. Hoe complexer iets is, hoe minder inzicht je vaak hebt in de gevolgen van de code die je schrijft. En als je niet precies doorgrondt met welke techniek je werkt, is een fout natuurlijk ook gauw gemaakt.

3. TESTEN: HOE?

In de voorgaande paragraaf hebben we gekeken naar wat testen is, en waarom je het zou moeten doen. Daarbij is nog niet aan de orde gekomen **hoe** we dat dan gaan doen. Dat zullen we nu nader gaan bekijken. We zullen ons richten op het testen van software-componenten. In ons geval zullen dat steeds individuele klassen zijn. Je kunt een testprogramma natuurlijk helemaal zelf schrijven, maar daar komt nog best wat bij kijken. We gebruiken daarom het JUnit 5 framework. Dit is in het Java-ecosysteem het meest gebruikte testframework.

Voor elk software-component dat we willen testen schrijven we een bijbehorende JUnit-testklasse. Een JUnit-testklasse bevat diverse testmethoden voor de verschillende situaties die we willen testen. Stel dat we de klasse Rekening van les 2 willen testen. We beginnen met een test voor de methode stort():

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class RekeningTest {

    @Test
    void stortenPositiefBedragWijzigtSaldo() {
        Rekening r = new Rekening(13245698);

        r.stort(100);
        assertEquals(100, r.getSaldo());

        r.stort(1000);
        assertEquals(1100, r.getSaldo());
    }
}
```

Het is niet verplicht om je testklasse een bepaalde naam te geven, maar een algemeen geaccepteerde conventie is om de test te noemen naar de klasse die je test, gevolgd door 'Test'. Hier dus **RekeningTest**. Voor een testmethode geldt dat uit de naam op te maken moet zijn wat de test doet.

Iets anders dat opvalt is de @Test annotatie. Een annotatie is te herkennen aan het @-teken. Het is onderdeel van het JUnit framework (1e import), en we gebruiken het hier om aan te geven dat de geschreven methode een testmethode is. Anders dan bij een gewone applicatie bevat deze klasse namelijk geen main-methode. Willen we deze test uitvoeren, dan wordt JUnit gestart, en deze voert dan de tests uit. Of beter gezegd; de methoden waar een annotatie voor staat!

Het tweede opvallende is dat deze testmethode een aantal methode-aanroepen kent die beginnen met **'assert'**. Dergelijke methoden (2e import[2]) gebruik je om een specfieke testcase te controleren. De eerste aanroep:

assertEquals(100, r.getSaldo());

controleert of de waarde 100 overeenkomt met het getal dat methode getSaldo() returnt. Is dat **niet** het geval, dat faalt hier onze test **direct**! Andere asserts in **dezelfde** methode worden niet meer uitgevoerd. Hoewel onze test als geheel dus al gefaald heeft, worden de overige testmethoden nog wel gewoon afgehandeld, en kunnen individueel dus nog slagen!

Er staat een waaier aan verschillende assertmethoden tot je beschikking. Een greep hieruit:

- assertEquals(int expected, int actual)
- assertEquals(int expected, int actual, String message)

[2] Voor nu mag je volstaan om je IDE deze methoden te laten importeren. Statics behandelen we later deze cursus.

- assertEquals(long expected, long actual)
- assertEquals(boolean expected, boolean actual)
- assertFalse(boolean condition)
- assertNotNull(Object actual)
- assertSame(Object expected, Object actual)
- assertTrue(boolean condition)

De tweede assert die hier is genoemd, bevat ook een message-parameter. Dit is handig als je een duidelijk leesbaar bericht wilt geven aan de gebruiker van de test. Gebruik je dat niet, dan krijg je standaard alleen het bericht dat methode X gefaald is op regel Y. Als je meerdere asserts in je testmethode hebt opgenomen, kan het handig zijn als daar direct bij staat wat precies het probleem is. Voor de meeste genoemde asserts is er ook een variant met message-parameter, hoewel die hier dus niet allemaal zijn genoemd.

Overigens zijn de meningen verdeeld over de vraag of een testmethode meer dan één assert zou moeten bevatten. Strikt genomen test je dan meer dan 1 testcase. Als de eerste assert fout gaat, worden de andere asserts in die methode niet meer gecontroleerd, en je krijgt daar dus ook geen informatie meer over. Daarnaast zou je beide controles in de naam van je methode moeten opnemen, wat de duidelijkheid en leesbaarheid van je code niet ten goede komt. Anderzijds zou ons voorbeeld op de vorige bladzijde niet per se leesbaarder worden als er nog een extra methode zou worden toegevoegd voor die ene assert.

De regel is dat je probeert het aantal asserts te beperken, maar als je er toch meer dan één opneemt, deze dan ook echt betrekking moeten hebben op dezelfde testcase. Niemand kan ons bijvoorbeeld weerhouden om ook nog een assert toe te voegen waarin we het storten van negatieve bedragen testen, maar dat komt a) niet overeen met de naam van de methode, en b) test eigenlijk een andere requirement ("negatieve bedragen storten is niet toegestaan"). Dit zou je dus moeten voorkomen!

Op 25 februari 1991 faalde een Amerikaanse Patriot luchtafweerbatterij om een Iraakse Scud-raket te onderscheppen. Oorzaak: de tijd in seconden werd berekend door de up-time van het systeem te vermenigvuldigen met 1/10. Het getal 1/10 is binair oneindig (vergelijkbaar met 1/3 decimaal), maar werd opgeslagen in een 24-bits register, wat dus een afwijking in accuraatheid opleverde.

Dit resulteerde na 100 uur in tijdsafwijking van 0,34 seconden. De plaatsbepaling waar de Scud-raket en onderscheppingsraket elkaar zouden treffen was daardoor incorrect. Gevolg: 28 Amerikaanse doden.

JUNIT ANNOTATIES

We hebben nu één test geschreven. Stel, we maken er een test bij:

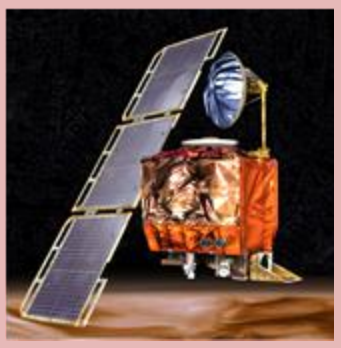
```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class RekeningTest {
    @Test
    void stortenPositiefBedragWijzigtSaldo() {
        Rekening r = new Rekening(13245698);

        r.stort(1000);
        assertEquals(1000, r.getSaldo());
    }

    @Test
    void stortenNegatiefBedragWijzigtSaldoNiet() {
        Rekening r = new Rekening(13245698);

        r.stort(-100);
        assertEquals(0, r.getSaldo());
    }
}
```

Je ziet dat nu in beide methoden hetzelfde Rekening-object aangemaakt moet worden om voor elke test met een schone lei te beginnen. Een belangrijke stelregel voor developers is echter Don't Repeat Yourself (DRY). Eén van de gedachten daarachter is dat wanneer zo'n stuk dubbele code ooit aangepast moet worden, je dat op alle plaatsen moet doen waar die code voorkomt. Het is bijna een wetmatigheid dat je het dan ergens vergeet, en op die manier introduceer je dus weer bugs in een programma.

ⓘ *Overigens zou de 2e test 'falen',want methode stort(.)accepteert negatieve bedragen!*

De Mars Climate Orbiter werd in 1998 naar Mars gestuurd om het klimaat op Mars te bestuderen.

Helaas werd de sonde vernietigd (vermoedelijk in de atmosfeer van Mars). Oorzaak: de software om de sonde vanaf de aarde aan te sturen, gebruikte parameters in het Britse imperiale stelsel, terwijl de software van de sonde uitging van het metrische stelsel… Kosten: 327,6 miljoen dollar.

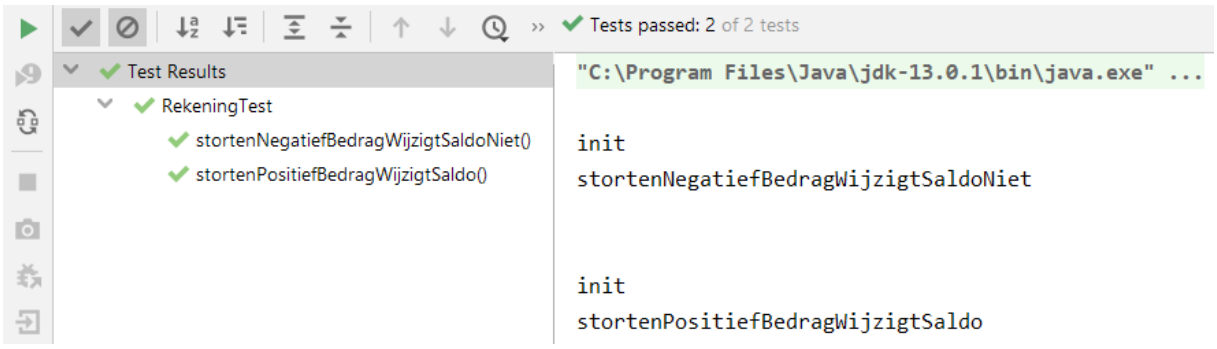


En natuurlijk is het gewoon vervelend als je iets twee keer moet coderen als het ook één keer kan… JUnit helpt ons daar een handje bij met de @BeforeEach annotatie:

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class RekeningTest {
    private Rekening r;

    @BeforeEach
    public void init() {
        System.out.println("init");
        r = new Rekening(13245698);
    }
    @Test
    void stortenPositiefBedragWijzigtSaldo() {
        System.out.println("stortenPositiefBedragWijzigtSaldo");
        r.stort(1000);
        assertEquals(1000, r.getSaldo());
    }
    @Test
    void stortenNegatiefBedragWijzigtSaldoNiet() {
        System.out.println("stortenNegatiefBedragWijzigtSaldoNiet");
        r.stort(-100);
        assertEquals(0, r.getSaldo());
    }
}
```

Methoden geannoteerd met @BeforeEach worden voor iedere testmethode uitgevoerd. Het Rekening-object is nu wel een attribuut geworden om ervoor te zorgen dat elke testmethode bij de referentie kan. De init-methode zorgt ervoor dat de referentie (het attribuut) net voorafgaande aan een testmethode naar een nieuw (schoon) Rekening-object gaat wijzen.

We kunnen dit in werking zien als we deze testklasse in z'n geheel laten uitvoeren. Vanwege de printouts in elke methode kunnen we in de console exact zien in welke volgorde de methoden worden uitgevoerd. Zie het volgende resultaat (in IntelliJ[3]):



Figuur 3: Links de JUnit testresultaten, rechts de console-output (System.out)

[3] Zie ook https://www.jetbrains.com/help/idea/configuring-testing-libraries.html

Naast @BeforeEach zijn er nog meer handige annotaties. We zetten er enkele op een rijtje:

@Test	**Methode** is een testmethode.
@BeforeEach	**Methode** wordt uitgevoerd **voor** elke testmethode.
@AfterEach	**Methode** wordt uitgevoerd **na** elke testmethode.
@BeforeAll	**Methode** wordt eenmalig uitgevoerd **voorafgaand aan alle tests**.
@AfterAll	**Methode** wordt eenmalig uitgevoerd **na afloop van alle tests**.
@Disabled	**Klasse of methode** is uitgeschakeld.

4. TESTONTWERP

We zijn nu technisch in staat om een test te bouwen. Hoewel een testframework allerlei randzaken regelt, geeft het je weinig hulp bij het schrijven van de test zelf. De test(code) die jij bedenkt wordt uitgevoerd, of dat nu een goede of slechte test is. Het is daarom belangrijk om je test goed te ontwerpen voordat je deze in code omzet. Bij andere cursussen komen diverse ontwerptechnieken aan de orde, maar we willen hier toch vast wat zaken aanstippen zodat je aan de slag kunt met unittesten.

Het begin van een goede test wordt gevormd door de **testbasis**: informatie over het (beoogde) gedrag van je systeem. Dit kan je halen uit bijvoorbeeld de requirements en andere functionele en/of technische documentatie. Aan de hand van de testbasis kan je de **condities** en **acties** van je systeem vaststellen. Een conditie is een situatie die zich wel of niet voordoet in het systeem, als gevolg van de acties van (bijvoorbeeld) een gebruiker.

Voorbeeld; de eis voor een programma is: **"De gebruiker krijgt korting als hij/zij 67 jaar of ouder is, of een museumpas heeft"**. Er is dan sprake van twee mogelijke condities:

a. De leeftijd is hoger of gelijk aan 67
b. De klant heeft een museumpas

De acties die het systeem kan ondernemen zijn:

a. Wel korting toekennen
b. Geen korting toekennen

Nu we een overzicht hebben van de condities en acties van je systeem, kan je **logische testgevallen** opstellen. Een logisch testgeval kent nog geen concrete input- of outputwaarden, maar beschrijft wel de verschillende situaties die zich in het programma kunnen voordoen, en die dus getest zouden moeten worden:

Testgeval	Leeftijd >= 67	In bezit van museumpas	Resultaat
1	Ja	Ja	Korting toekennen
2	Nee	Ja	Korting toekennen
3	Ja	Nee	Korting toekennen
4	Nee	Nee	Geen korting

Van een logisch testgeval kan je **fysieke testgevallen** afleiden. Voor een fysiek testgeval vervang je de condities door concrete waarden. Welke waarden je het beste kunt gebruiken, kan je bijvoorbeeld achterhalen met behulp van een **grenswaardenanalyse**.

Voor de leeftijd is de grenswaarde bijvoorbeeld 67. Interessant is dan vaak om net onder, op, en net over de grenswaarde te testen. In dit voorbeeld dus 66, 67 en 68. Voor het al dan niet hebben van een museumpas zijn er maar twee opties: true of false. Willen we alle combinaties testen, dan krijg je de volgende fysieke testgevallen:

Testgeval	Invoeren leeftijd:	Invoeren museumpas:	Resultaat
1	67	True	Korting toekennen
	68	True	Korting toekennen
2	66	True	Korting toekennen
3	67	False	Korting toekennen
	68	False	Korting toekennen
4	66	False	Geen korting

Met deze termen en technieken hebben we een tipje van de testsluier opgelicht. Andere cursussen zullen hier nog verder mee aan de slag gaan. Voor nu zou je echter genoeg bagage moeten hebben om voor beperkte domeinmodellen tests te kunnen gaan schrijven.

➔ MAAK NU EERST OPDRACHT 3_1 UIT HET WERKBOEK!

5. EXCEPTION-HANDLING

In Java, en veel andere talen, worden exceptions gebruikt om fouten en andere buitengewone gebeurtenissen af te handelen. Er zijn veel oorzaken waardoor er een fout in je programma kan optreden. Zo veroorzaakt het delen van een getal door 0 een **ArithmeticException**. Of als de netwerkverbinding wegvalt als je net iets probeert te downloaden: **IOException**. Of als je methoden aanroept op een null-variabele (**NullPointerException**), zoals in het onderstaande voorbeeld het geval is. Laten we eens kijken naar een, op zichzelf normale, klasse Student:

```
public class Student {
  private String naam;
  private String email;

  public Student(String nm, String em) {
    naam = nm;

    if (em.endsWith("@hu.nl")) {
      email = em;
    } else {
      email = "";
    }
  }
}
```

Listing 13: Klasse Student, met op het oog een prima constructor

Uit de constructor is af te leiden dat een Student-object alleen maar een HU e-mailadres mag hebben. Als dat niet zo is, zal de parameter-waarde genegeerd worden, en is het e-mailadres standaard een lege string. Dit lijkt een prima constructor, maar als we de volgende main-methode uitvoeren, komen we toch voor verrassingen te staan:

```
public class Main {
  public static void main(String[] args) {
    Student s1 = new Student("Kees", null);
    System.out.println(s1);
  }
}
```

Listing 14: Bij uitvoeren van deze main-methode onstaat een exception

Als je dit programma zou uitvoeren, dan geeft de JVM de volgende output:

```
Exception in thread "main" java.lang.NullPointerException
        at Student.<init>(Student.java:7)
        at Main.main(Main.java:3)
```

De eerste regel van de uitvoer toont **welke** fout er is opgetreden; een **NullPointerException**. De navolgende informatie toont de **stacktrace**; ofwel welke code werd uitgevoerd op het moment dat de fout optrad. De **stack** is een 'stapel' van methoden, en is het beste te begrijpen als je 'm van onder naar boven leest. De onderste regel;

```
at Main.main(Main.java:3)
```

vertelt ons dat de methode main van klasse Main werd uitgevoerd. En, om precies te zijn, regel 3 van bestand Main.java. We kunnen die regel bekijken, maar we hebben de top van de stacktrace nog niet bereikt. Daaruit is op te maken dat op deze regel de fout nog niet is opgetreden, maar dat deze code weer een andere methode heeft aangeroepen. Dat is:

```
at Student.<init>(Student.java:7)
```

Kennelijk is dat een methode in klasse Student, de aanduiding <init> geeft aan dat het om een constructor gaat. De exacte locatie is regel 7 van het bestand Student.java. Daar staat:

```
if (em.endsWith("@hu.nl")) {
```

Een NullPointerException wil zeggen dat je een variabele probeert te gebruiken, die naar geen enkel (null) object wijst. Het is dus zaak om te bepalen welke variabele op deze regel daarvoor in aanmerking komt. Omdat er maar 1 variabele is (`em`), is dat ook meteen de boosdoener. De **oorzaak** hiervan ligt echter in de klasse Main, waar een **null-waarde** is doorgegeven als parameter.

We kunnen natuurlijk de klasse Main aanpassen, zodat methode main een string-waarde doorgeeft. Maar dat is geen echte oplossing, want andere programmeurs die de klasse Student gebruiken, kunnen vervolgens alsnog tegen dit probleem aanlopen. Daarom moet het

probleem in de klasse Student worden opgelost. De meest effectieve oplossing is om te controleren of de parameter null is, of niet:

```
public class Student {
  private String naam;
  private String email;

  public Student(String nm, String em) {
    naam = nm;

    if (em != null && em.endsWith("@hu.nl")) {
      email = em;
    } else {
      email = "";
    }
  }
}
```

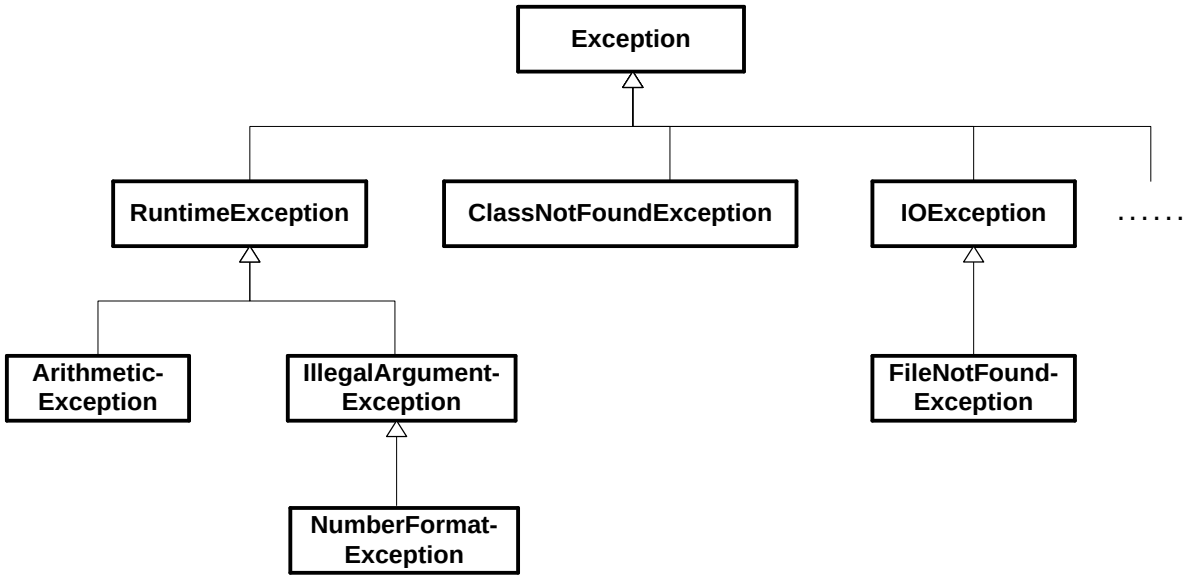
Listing 15: De verbeterde versie van klasse Student; er is nu een null-check toegevoegd

Let op; de check em != null moet als **eerste** worden uitgevoerd! Komt hier **false** uit, dan zal de tweede check niet worden uitgevoerd (dit zou ook geen zin meer hebben, want het eindresultaat van een AND-operatie kan dan nooit meer **true** worden). De &&-operator noem je daarom ook wel een short-evaluation operator. Zie ook onderwerp 'Logische operatoren'.

Deze oplossing is natuurlijk prima, maar niet altijd is zo'n eenvoudige oplossing mogelijk. We onderscheiden namelijk een aantal verschillende typen fouten:

1. **Checked exceptions**; een uitzonderlijke toestand in het programma, waarvan een goed programma zou moeten kunnen herstellen. Bijvoorbeeld wanneer je een bestand wilt openen, waarvan de gebruiker de naam heeft ingevoerd. Als de ingevoerde naam niet klopt, zal de Java-library voor file-bewerkingen een exception opleveren. Hiermee moet je rekening houden. Dit soort fouten kan je vaak afhandelen met speciale exception-handling constructies.
2. **Runtime exceptions**; fouten die optreden door programmeer*bugs*; fouten in de logica van het programma. Deze kunnen *meestal* voorkomen worden door de code aan te passen. Ons voorbeeld van zojuist valt in deze categorie.
3. **Errors**; fouten die optreden door oorzaken die buiten het programma liggen. Bijvoorbeeld het niet goed functioneren van hardware. Hier kan je meestal geen rekening mee houden in je programma, simpelweg omdat je niet alle mogelijke problemen kunt voorzien, en de oorzaak meestal niet kunt wegnemen.

We zullen ons alleen richten op exceptions. De ontwikkelaars van de taal Java hebben al een groot aantal soorten exceptions in de taal opgenomen. Daarin is het onderscheid tussen checked exceptions en runtime exceptions ook terug te zien in Figuur 15. Alle exceptions die niet onder RuntimeException vallen, zijn checked exceptions.



Figuur 4: Een aantal van de ingebouwde Java-exceptions

Als het voorkomt dat er exceptions optreden in je programma, dan wil je daar natuurlijk op in kunnen spelen. Dat kan bijvoorbeeld met een try-catch blok.

6. TRY-CATCH

Een try-catch blok is een constructie, waarmee je bepaalde code in een 'gecontroleerde' omgeving kunt uitvoeren: het '**try**-block'. Vervolgens specificeer je in het '**catch**-block' wat er gedaan moet worden als er daadwerkelijk iets misgaat in het programma. We nemen als voorbeeld een programma waarin een deling wordt uitgevoerd:

```
import java.util.Random;

public class Main {
  public static void main(String[] args) {
    int getal1 = new Random().nextInt(); // willekeurig integer getal
    int getal2 = new Random().nextInt(); // willekeurig integer getal

    try {
      int uitkomst = getal1 / getal2;
      System.out.println(uitkomst);

    } catch (ArithmeticException ae) {
      System.out.println("Fout opgetreden: " + ae.getMessage());
    }
  }
}
```

Listing 16: Deling met twee willekeurige getallen kan misgaan als getal2 == 0

Het zou natuurlijk kunnen dat getal2 de waarde 0 toegekend krijgt. De deling:

```
int uitkomst = getal1 / getal2;
```

zou dan resulteren in een exception. Doordat de code in een try-blok staat, zal direct de uitvoering van dat try-blok stoppen, en wordt het catch-blok uitgevoerd. Deze code print dan netjes dat er een fout is opgetreden, en daarachter komt het exception-bericht te staan. Exception-informatie kan je opvragen aan het ArithmeticException-object. Deze heeft de naam **ae** (afkorting van het exception-type) gekregen, maar die naam mag je zelf verzinnen.

```
import java.util.Random;

public class Main {
  public static void main(String[] args) {
    int getal1 = new Random().nextInt(); // willekeurig integer getal
    int getal2 = new Random().nextInt(); // willekeurig integer getal

    try {
      int uitkomst = getal1 / getal2;
      System.out.println(uitkomst);
    } catch (ArithmeticException ae) {
      System.out.println("Fout opgetreden: " + ae.getMessage());
    } catch (Exception e) {
      System.out.println("Algemene fout: " + e.getMessage());
    }
  }
}
```

Listing 17: Bijna hetzelfde programma als Listing 16, maar met extra catch-blok

Het catch-blok van listing 16 **vangt alleen** exceptions van het type ArithmeticException af. Alle andere exceptions zorgen alsnog voor een abrupt einde van het programma. Wil je **alle** andere fouten ook afvangen, dan kan dat door een extra catch-blok toe te voegen, zoals in listing 17 op de vorige pagina is gedaan.

ⓘ *Wil je meerdere catch-blokken toepassen, dan moet je de meest specifieke exception bovenaan plaatsen. Een ArithmeticException is bijvoorbeeld specifieker dan de algemene Exception, omdat het een subtype daarvan is. Dit is te zien in Figuur 15. In latere lessen zullen we verder ingaan op subtypen.*

➔ MAAK NU EERST OPDRACHT 3_2 UIT HET WERKBOEK!

7. EXCEPTIONS CREËREN & THROWS-CLAUSULE

Soms schrijf je code voor iemand anders, en wil je in een methode aangeven dat er iets is misgegaan. Je kunt dan zelf een exception creëren. Stel dat klasse Student ook een methode zou hebben om een e-mailadres op een later moment te wijzigen. We kunnen in de methode een exception laten creëren op het moment dat er geen @ in het e-mailadres voorkomt:

```
public void wijzigEmail(String em) throws Exception {
  if (!em.contains("@")) {
    throw new Exception("@ ontbreekt!");
  }
```

```
        email = em;
    }
```

Als jouw methode een exception kan *gooien*, **moet je ook een throws-clausule opnemen, tenzij het een (subtype van) RuntimeException betreft!** Dat is omdat de compiler op deze manier kan controleren of andere code die deze methode gebruikt, wel iets doet om de fout af te handelen. Bijvoorbeeld met een try-catch blok.

ⓘ *De gedachte is dat runtime exceptions niet voor zouden moeten komen als het programma goed is geschreven. Daarom hoeven die niet per se te worden afgehandeld, en dus ook niet te worden gemeld met een throws-clausule.*

Je kunt deze methode vanuit de klasse Main als volgt gebruiken:

```
public class Main {
  public static void main(String[] args) {
    Student s1 = new Student("Kees", null);

    try {
      s1.wijzigEmail("kees@hu.nl");
    } catch (Exception e) {
      System.out.println("Wijzigen niet geslaagd!");
    }
  }
}
```

Listing 18: Methode wijzigEmail(.) kan een Exception opleveren

Wil je de exception niet afhandelen in de main-methode, dan kan je er eventueel voor kiezen om de exception **door te gooien** door ook aan deze methode een throws-clausule toe te voegen:

```
public class Main {
  public static void main(String[] args) throws Exception {
    Student s1 = new Student("Kees", null);

    s1.wijzigEmail("kees@hu.nl");
    System.out.println("Wijzigen geslaagd!");
  }
}
```

Listing 19: Je kunt ervoor kiezen de exception niet af te handelen, maar 'door te gooien'

Het is de vraag of dat handig is, want de main-methode is het laatste vangnet. Een exception komt daarna terecht bij de JVM, en deze zal het programma abrupt stoppen, en de fout (inclusief stacktrace) uitprinten.

Naast een algemene exception kan je ook een specifiekere exception gooien (throwen). Java kent diverse exceptions die voor speciale situaties te gebruiken zijn, zoals ook in Figuur 4 is getoond. Is dat niet genoeg, dan kan je zelf een exception-type maken. Zie hiervoor bijlage 1 (de bijlage is geen tentamenstof).

LES 4 – RELATIES TUSSEN KLASSEN (ASSOCIATIES)

1. INLEIDING

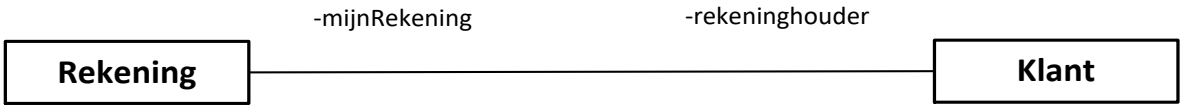
Tot deze les hebben we alleen met programma's gewerkt waarin 1 klasse werd geschreven. Maar in bijna alle situaties waarin je de werkelijke wereld in een model wilt vangen is sprake van meer dan 1 entiteit. We hebben bijvoorbeeld in les 2 de klasse Rekening gezien. Maar in de werkelijkheid is er natuurlijk altijd ook een klant die de bezitter is van een rekening. In dat geval gaat het klassendiagram bestaan uit meer dan 1 klasse die onderling bij elkaar betrokken zijn.

Wanneer er wordt gewerkt met meerdere klassen bestaan er tussen die klassen relaties. Tussen twee klassen kunnen drie soorten relaties bestaan: een **associatie**-relatie, een **generalisatie**-/**specialisatie**-relatie of een **aggregatie**-/**compositie**-relatie. We kijken in deze les alleen naar de **associatie**-relatie.

Een associatie-relatie heet ook wel een **knows-relation**. De ene klasse kent de andere klasse. Bij deze soort relatie hebben de twee klassen ook onafhankelijk van elkaar bestaansrecht. Je zegt in het bovenstaande geval dan: "de rekening kent zijn rekeninghouder" of juist "de rekeninghouder kent zijn rekening(en)", afhankelijk van welke relatie er tussen de twee klassen bestaat.

2. ASSOCIATIES

In de voorbeelden van deze les is sprake van een associatie tussen de klassen Rekening en Klant. In een (eenvoudig) UML klassendiagram geef je een associatie aan met een lijn of pijl:

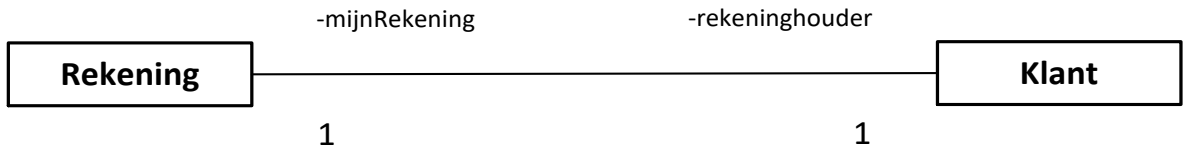


Je ziet dat er in dit voorbeeld niet alleen een klasse-naam is opgenomen, maar ook een **rol-naam**. Dit geeft je informatie over de relatie (rol) die de klassen onderling hebben. Omdat er geen pijlen tussen de beide klassen staan gaat de relatie beide kanten op. In dit geval moet je het diagram dan ook lezen als: "een rekening **kent** zijn rekeninghouder **èn** de rekeninghouder **kent** zijn rekening". De relatie is dan **bi-directioneel** (beide kanten op).

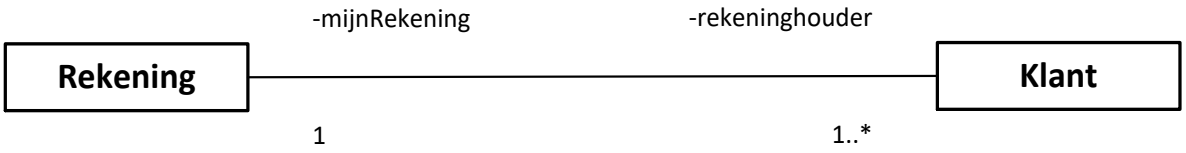
Het woord '**kennen**' duidt op het feit dat een Rekening-object een **koppeling/link** heeft naar een Klant-object. Andersom is dat ook het geval: Een Klant-object heeft ook een koppeling/link naar een Rekening-object.

ⓘ *Een koppeling kun je vergelijken met een telefoonnummer. Als je iemands telefoonnummer hebt kun je diegene dus per telefoon benaderen. Maar als die andere persoon jouw nummer **niet** heeft kan diegene jou dus **niet** per telefoon bereiken! Pas als die andere persoon jouw nummer ook heeft is er sprake van een **bi-directionele** relatie!*

In dit eerste voorbeeld is stilzwijgend aangenomen dat een klant maar 1 rekening heeft en dat een rekening bij slechts 1 klant hoort! Je zegt dan dat de **multipliciteit** 1 is. De multipliciteit geeft aan hoeveel objecten bij elkaar betrokken kunnen zijn. Maar eigenlijk moet je dat in een klassendiagram er expliciet bijzetten:



Multipliciteiten zijn belangrijk om de relatie te verduidelijken. Zo is het bij sommige banken bijvoorbeeld mogelijk om met 1 of meerdere (hieronder weergegeven met *****) personen een gezamenlijke rekening kunnen openen:



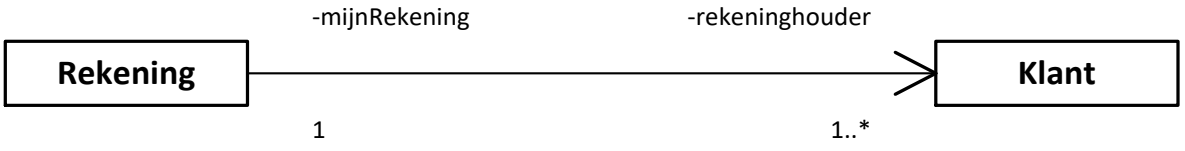
Multipliciteiten zijn er vele soorten en maten:

- 0..1: nul of één: de relatie is optioneel
- 0..*: nul of meer: optioneel, maar tegelijk onbegrensd, ook weer te geven als '*****'
- 1..1: altijd één: de relatie is verplicht, ook weer te geven als '**1**'
- 1..*: minimaal één: verplicht, maar tegelijk onbegrensd
- m..n: minimaal **m**, maar niet meer dan **n** links naar objecten

In voorgaande voorbeelden is er steeds een **bi-directionele** associatie: objecten van beide klassen kennen elkaar. Maar in veel gevallen is het voldoende als de koppeling/link tussen twee objecten maar 1 kant op ligt. Dan spreek je van een **uni-directionele** relatie.

ⓘ *Een uni-directionele associatie kun je vergelijken met een situatie waarin jij wel het telefoonnummer van een ander hebt, maar de ander niet jouw nummer heeft. Jij bent dan ten allen tijde verantwoordelijk voor het onderhouden van de relatie!*

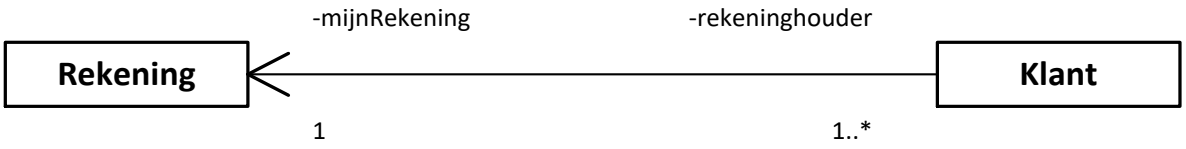
Een uni-directionele associatie kan je in een klassendiagram weergeven met een pijl. Dit noem je de navigeerbaarheid van een associatie. De kant waar de pijl **begint** is dan **verantwoordelijk** voor de relatie:



In bovenstaande voorbeeld is klasse **Rekening** dus de **verantwoordelijke**! Klasse Klant weet helemaal **niets** van de relatie met Rekening! Wie verantwoordelijk is, hangt af van het

omgeving (domein) waarin je werkt. Wanneer je een programma voor een bank moet programmeren staat vaak de rekening centraal omdat een rekening uniek is binnen de bank. Een klantnaam is dat niet. Het is dan handig om via een rekening de klantgegevens erbij te zoeken. Klasse Rekening is dan verantwoordelijk voor de relatie.

Als je echter een boekhoudprogramma voor thuisgebruikers aan het ontwikkelen bent zou het kunnen zijn dat je per gebruiker een lijst met rekeningen wilt bijhouden. Een klantnaam of gebruikersnaam is dan vaak uniek en gebruik je om in te loggen. Via die klantnaam is het handig om de rekeninggegevens van de klant op te zoeken. In dat geval staat de klant centraal en is die klasse verantwoordelijk. Dan ziet het UML-klassendiagram er als volgt uit:



Tot slot kun je door middel van **associatienamen** nog meer verduidelijken hoe een associatie in elkaar steekt. Dat kan de overzichtelijkheid soms in de weg zitten, dus in deze reader laten we de associatienaam steeds achterwege. Met als uitzondering onderstaande afbeelding:

Nog even alle de eigenschappen van associaties op een rij:

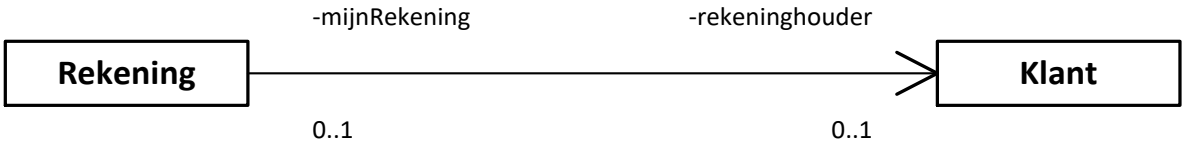
- Een associatie is een structurele relatie tussen twee klassen (op klasse-niveau).
- Een associatie wordt vertaald naar een link tussen twee objecten (op object-niveau).
- Een associatie heeft een richting, aangegeven door een pijl.
- De richting van een associatie heet: de navigeerbaarheid. Let op de pijl !!
- Een associatie heeft rolnamen. In het vb. heeft klasse Rekening de rolnaam ‘mijnRekening’ en klasse Klant heeft de rolnaam ‘rekeninghouder’.
- Een associatie heeft multipliciteiten.
- Bij een associatie van Rekening **naar** Klant heet klasse Rekening de **verantwoordelijke** en klasse Persoon de **afhankelijke** klasse. Dan zeg je: Rekening "kent" Klant. Omgekeerd kent Klant Rekening niet.

3. KLASSE REKENING "KENT" KLASSE KLANT

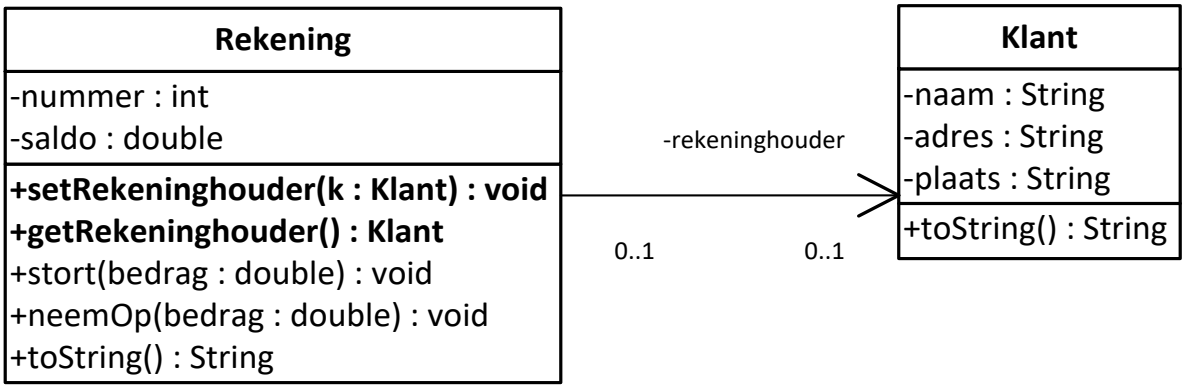
We hebben nu gezien hoe je een relatie kunt aangeven in een UML klassendiagram. De besproken diagrammen zijn eenvoudige diagrammen waarin alle methoden en attributen achterwege zijn gelaten. Maar je hebt als je een ontwerp maakt meestal de intentie om dit ook daadwerkelijk in code om te zetten. Dat is het doel van deze paragraaf. We beginnen met een korte casus waaruit duidelijk moet worden in welke reële situatie (domein) dit voorbeeld zou kunnen voorkomen.

> Klaas Knot (van Knab bank, zie les 2) heeft als opdracht gekregen om voor de telefonische helpdesk een GUI te ontwikkelen waarin medewerkers van de helpdesk klantgegevens kunnen zien. Als klanten met de bank bellen moeten zij eerst hun rekeningnummer invoeren voordat ze een medewerker aan de lijn krijgen. Als de medewerker de klant aan de lijn krijgt ziet hij (of zij) direct de gegevens van de klant voor zich op het scherm.

Omdat een rekening voor Klaas het belangrijkste startpunt is, neemt hij klasse **Rekening** als uitgangspunt. Daarvandaan moeten de gegevens van de klant worden opgezocht. Klasse Rekening is daarom de **verantwoordelijke klasse**:



Het eenvoudige klassendiagram hierboven geeft de essentie van de associatie weer. Maar het is daarbij helaas niet duidelijk welke attributen en methoden deze klassen hebben. Daarvoor hebben we een uitgebreider diagram nodig. De klassen Klant en Rekening zijn gebaseerd op de gelijknamige klassen uit les 1 en 2. Echter, om de **koppeling** tussen een Rekening en Klant object te kunnen leggen zijn **extra** methoden nodig:



Let op: Omdat het teveel ruimte in beslag zou nemen om alle getters, setters en constructors op te nemen, zijn hierboven **alleen attributen** en **belangrijke methoden** opgenomen!

Klasse **Rekening** is **verantwoordelijk** voor de relatie, dus die klasse krijgt **extra methoden** om de **koppeling** te **maken** of te **raadplegen**! De extra methoden voor het leggen van de koppeling tussen Rekening en Klant zijn:

setRekeninghouder(k : **Klant**) : void
getRekeninghouder() : **Klant**

De **set-methode** kan gebruikt worden om de **koppeling** te **maken**. De **get-methode** kan je aanroepen als je van een Rekening-object het bijbehorende **Klant-object** wilt **opvragen**.

Merk op dat de setter een parameter heeft van het type Klant!! Dit **type** heb je dus **zelf ontworpen**. De getter heeft als return-type nu geen double, int of String maar ook het zelfgemaakte type **Klant**!

We zijn nu zover dat we de code kunnen maken die bij dit model hoort. Zoals al eerder aangegeven is dit een uni-directionele associatie. Klasse Klant is niet verantwoordelijk en weet dus helemaal **niets** van de associatie met klasse Rekening. Dat is terug te zien in de code van klasse Klant, die niet gewijzigd is ten opzichte van les 1:

```
public class Klant {
  private String naam;
  private String adres;
  private String plaats;

  public Klant(String nm, String ad, String pl) {
    naam = nm;
    adres = ad;
    plaats = pl;
  }

  public void setAdres(String ad)  { adres = ad;  }
  public void setPlaats(String pl) { plaats = pl; }

  public String getNaam()   { return naam; }
  public String getAdres()  { return adres; }
  public String getPlaats() { return plaats; }

  public String toString() {
    return naam + " woont op " + adres + " in " + plaats;
  }
}
```

Listing 20: Klasse Klant is niet verantwoordelijk en is dus gelijk aan de klasse van les 1, listing 7

Vervolgens kan de klasse Rekening geprogrammeerd worden. Deze klasse is **wel** verantwoordelijk. De klasse bevat daarom onder andere de methoden om de koppeling te leggen.

```
public class Rekening {
  private int nummer;
  private double saldo = 0.0;
  private Klant rekeninghouder;                    // associatie naar Klant

  public Rekening(int nr) {
    nummer = nr;
  }

  public void setRekeninghouder(Klant k) {
    rekeninghouder = k;
  }

  public Klant getRekeninghouder() {
    return rekeninghouder;
  }

  public double getSaldo() { return saldo; }
  public int getNummer()   { return nummer; }

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

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

  public String toString() {
    String s = "Op rekening " + nummer + " staat " + saldo;
    if (rekeninghouder == null) {
      s = s + " en de rekeninghouder is onbekend";
    }
    else {
      s = s + ";\nen de rekeninghouder is: " + rekeninghouder.toString();
    }
    return s;
  }
}
```

Listing 21: Klasse Rekening, verantwoordelijk voor de relatie met Klant dus inclusief attribuut, getter en setter

Let op: de klasse heeft nu naast de getter en setter **ook** een extra attribuut 'rekeninghouder' gekregen. Dit attribuut staat **niet expliciet** in het klassendiagram, maar de **pijl** van Rekening **naar** Klant in het klassendiagram resulteert in een **attribuut** in **klasse Rekening**! Eigenlijk is de pijl dus een impliciet attribuut.

Merk op dat de koppeling van Rekening naar Klant **optioneel** is want de multipliciteit is 0..1. Het kan dus zijn dat aan de rekening geen klant is gekoppeld. Daarom moet methode toString() controleren of attribuut rekeninghouder **null** is voordat van de rekeninghouder informatie

opgevraagd kan worden! De rekeninghouder is van het type Klant, dus methode toString() van klasse Klant wordt aangeroepen! Zie ook de uitvoer van de klasse Main die hieronder volgt.

Nu beide klassen zijn geschreven, kunnen we een applicatie schrijven die deze twee klassen gebruikt en test. Hierbij maken we eerst twee rekeningen aan, nog zonder dat er een klant aan gekoppeld is. Na uitprinten van deze twee rekeningen zal aan beide rekeningen een klant gekoppeld worden. Vervolgens printen we de rekeningen en rekeninghouders:

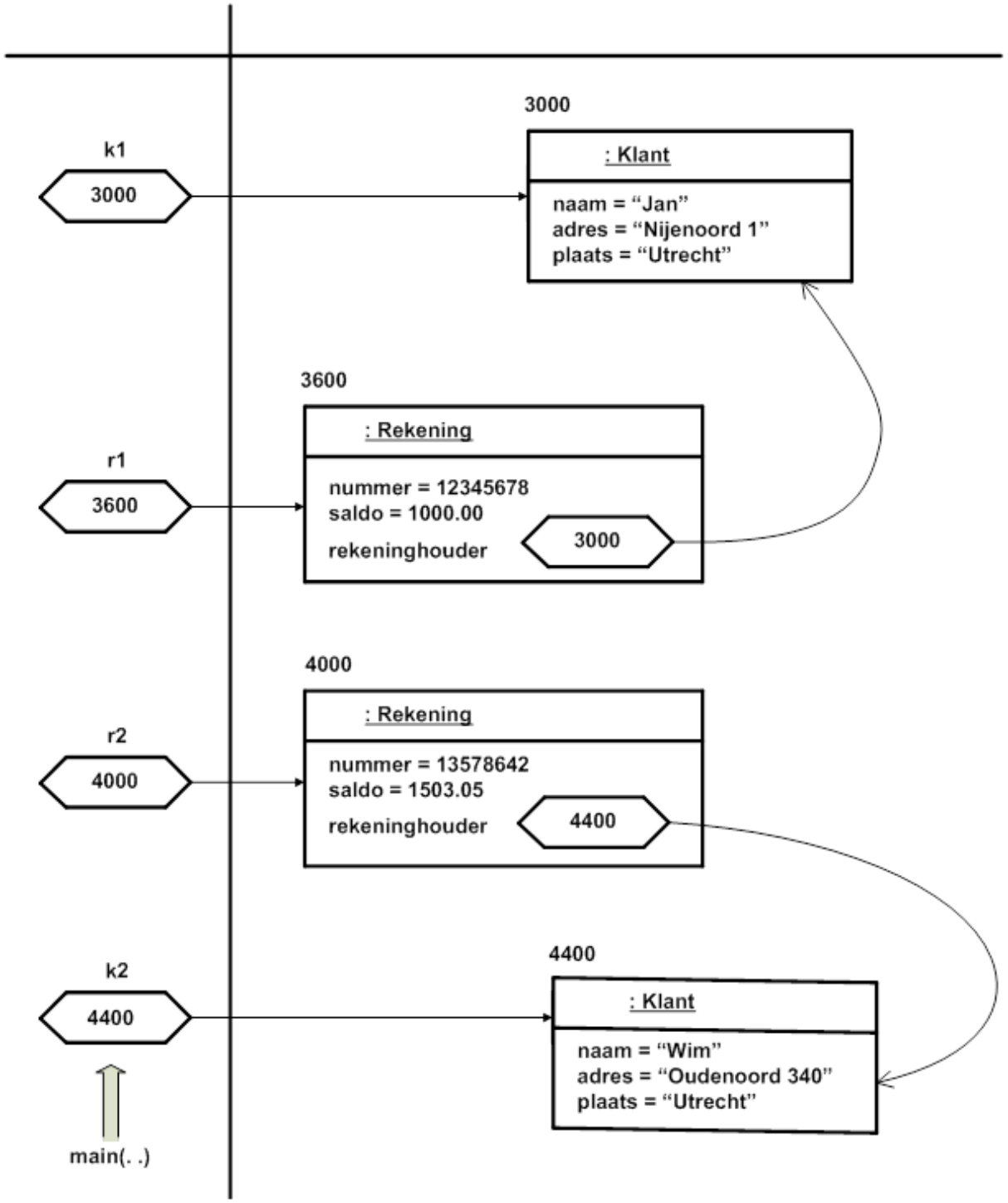
```
public class Main {
  public static void main(String[] arg) {
    // maak 2 objecten aan van klasse Rekening; en stort:
    Rekening r1 = new Rekening(12345678);
    Rekening r2 = new Rekening(13578642);
    r1.stort(1000);
    r2.stort(1503.05);
    System.out.println(r1);
    System.out.println(r2);
    System.out.println();

    // we maken nu voor elk Rekening-object een Klant -object aan:
    Klant k1 = new Klant("Jan", "Nijenoord 1", "Utrecht");
    Klant k2 = new Klant("Wim", "Oudenoord 340", "Utrecht");

    // en maken vervolgens de koppeling:
    r1.setRekeninghouder(k1);
    r2.setRekeninghouder(k2);

    System.out.println(r1);
    System.out.println(r1.getRekeninghouder());       // Klant-object
    System.out.println();
    System.out.println(r2);
    System.out.println(r2.getRekeninghouder());       // Klant-object
  }
}
```

Listing 22: Klasse Main maakt de koppeling tussen Rekening- en Klant-objecten

De bijbehorende uitvoer:

```
Op rekening 12345678 staat 1000.0 en de rekeninghouder is onbekend
Op rekening 13578642 staat 1503.05 en de rekeninghouder is onbekend

Op rekening 12345678 staat 1000.0;
en de rekeninghouder is: Jan woont op Nijenoord 1 in Utrecht
Jan woont op Nijenoord 1 in Utrecht

Op rekening 13578642 staat 1503.05;
en de rekeninghouder is: Wim woont op Oudenoord 340 in Utrecht
Wim woont op Oudenoord 340 in Utrecht
```

Het geheugen van je computer zal er bij uitvoeren van klasse Main er dan zo uitzien:



Figuur 5: het interne geheugen van de computer tijdens het runnen van methode 'main' uit listing 22

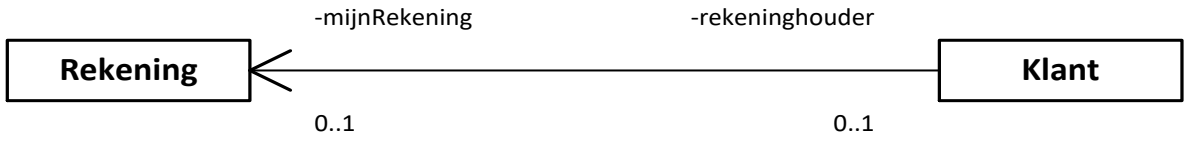
➔ MAAK NU EERST OPDRACHT 3_1 UIT HET WERKBOEK!

4. KLASSE KLANT "KENT" KLASSE REKENING

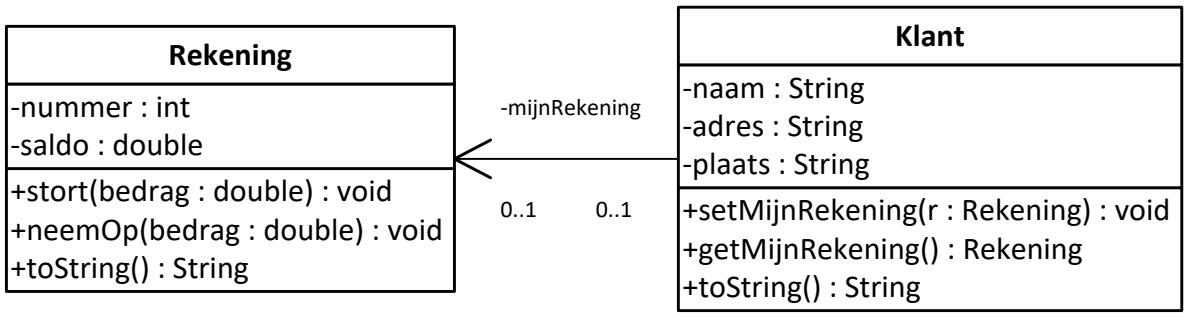
In de vorige paragraaf zagen we dat Rekening verantwoordelijk was voor de relatie met Klant. Maar niet in alle situaties is dat de gewenste oplossing. Deze paragraaf voert dezelfde aanpak uit, maar dan met klasse **Klant** als **verantwoordelijke**. De casus hierbij:

> Rosa Dinat doet haar boekhouding in Excel. Hierin import ze rekeningafschriften via internetbankieren. Helaas moet ze steeds handmatig aangeven welke afschrijvingen onder welke kostenpost vallen. Ze wil daarom een applicatie ontwikkelen waarin die afschriften automatisch worden gedownload en ondergebracht bij de juiste kostenpost. De code plaatst ze in GitHub zodat ook andere mensen de applicatie kunnen gebruiken.

Rosa maakt een generieke opzet zodat elke gebruiker een account kan aanmaken en hier een rekening aan kan koppelen. Daarom neemt zij juist klasse Klant als uitgangspunt:



Dit vereenvoudigde klassendiagram lijkt natuurlijk heel veel op de situatie in de vorige paragraaf. Maar let op; de **pijl** staat hier de **andere kant** op! Dat zorgt ervoor je moet lezen: "een klant **kent** zijn rekening". Nu wordt dus ineens klasse **Klant** de **verantwoordelijke** klasse! **Extra** methoden voor het onderhouden van relatie komen dus in die klasse terecht. Het uitgebreidere klassendiagram ziet er dan zo uit:



Let op: Omdat het teveel ruimte in beslag zou nemen om alle getters, setters en constructors op te nemen, zijn hierboven **alleen attributen** en **belangrijke methoden** opgenomen! Ook is de rolnaam van de **niet-verantwoordelijke** klasse weggelaten omdat deze in dit domein minder relevant is.

Klasse **Klant** is **verantwoordelijk** voor de relatie, dus die klasse krijgt **extra methoden** om de **koppeling** te **maken** of te **raadplegen**! Een setter en een getter dus. De extra methoden voor het leggen van de koppeling tussen Klant en Rekening zijn:

setMijnRekening(r : **Rekening**) : void
getMijnRekening() : **Rekening**

De **set-methode** kan gebruikt worden om de **koppeling** te **maken**. De **get-methode** kan je aanroepen als je van een Klant-object het bijbehorende **Rekening-object** wilt **opvragen**.

Merk op dat de setter een parameter heeft van het type Rekening!! Dit **type** heb je dus **zelf ontworpen**. De getter heeft als return-type nu geen double, int of String maar ook het zelfgemaakte type **Rekening**!

We zijn nu zover dat we de code kunnen maken die bij dit model hoort. Ook deze relatie is een uni-directionele associatie. Klasse Rekening is niet verantwoordelijk en weet dus helemaal **niets** van de associatie met klasse Klant. Dat is terug te zien in de code van klasse Rekening, die nu weer terug bij af is en dus niet gewijzigd is ten opzichte van les 1:

```
public class Rekening {
  private int nummer;
  private double saldo = 0.0;

  public Rekening(int nr) {
    nummer = nr;
  }

  public double getSaldo() {return saldo; }
  public int getNummer()   {return nummer;}

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

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

  public String toString() {
    return "Op rekening " + nummer + " staat " + saldo;
  }
}
```

Listing 23: Klasse Rekening is niet verantwoordelijk en is dus gelijk aan de klasse van les 2, listing 9

Vervolgens kan de klasse Klant geprogrammeerd worden. Deze klasse is **wel** verantwoordelijk. De klasse bevat daarom onder andere de methoden om de koppeling te leggen.

```
public class Klant {
  private String naam;
  private String adres;
  private String plaats;
  private Rekening mijnRekening;                // associatie naar Rekening

  public Klant(String nm, String ad, String pl) {
    naam = nm;
    adres = ad;
    plaats = pl;
  }

  public void setMijnRekening(Rekening r) {
    mijnRekening = r;
  }

  public Rekening getMijnRekening() {
    return mijnRekening;
  }

  public void setAdres(String ad) { adres = ad; }
  public void setPlaats(String pl) { plaats = pl; }

  public String getNaam() { return naam; }
  public String getAdres() { return adres; }
  public String getPlaats() { return plaats; }

  public String toString() {
    String s = naam +", woont op "+ adres +", in "+ plaats;
    if (mijnRekening == null) {
      s = s + "; er is nog geen rekening bekend";
    }
    else {
      s = s + ";\n" + mijnRekening.toString();
    }
    return s;
  }
}
```

Listing 24: Klasse Klant, verantwoordelijk voor de relatie met Rekening dus inclusief attribuut, getter en setter

Let op: de klasse heeft nu naast de getter en setter **ook** een extra attribuut 'mijnRekening' gekregen. Dit attribuut staat **niet expliciet** in het klassendiagram, maar de **pijl** van Klant **naar** Rekening in het klassendiagram resulteert in een **attribuut** in **klasse Klant**! Eigenlijk is de pijl dus een impliciet attribuut.

Merk op dat de koppeling van Klant naar Rekening **optioneel** is want de multipliciteit is 0..1. Het kan dus zijn dat aan de klant geen rekening is gekoppeld. Daarom moet methode toString() controleren of attribuut mijnRekening **null** is voordat van de informatie van de rekening opgevraagd kan worden! Het attribuut mijnRekening is van het type Rekening, dus methode **toString()** van **klasse Rekening** wordt aangeroepen!

Nu beide klassen zijn geschreven, kunnen we een applicatie schrijven die deze twee klassen gebruikt en test. Hierbij maken we eerst twee klanten aan, nog zonder dat er een rekening aan gekoppeld is. Na uitprinten van deze twee objecten zal aan beide klanten een rekening gekoppeld worden. Vervolgens printen we de klanten en hun rekeningen:

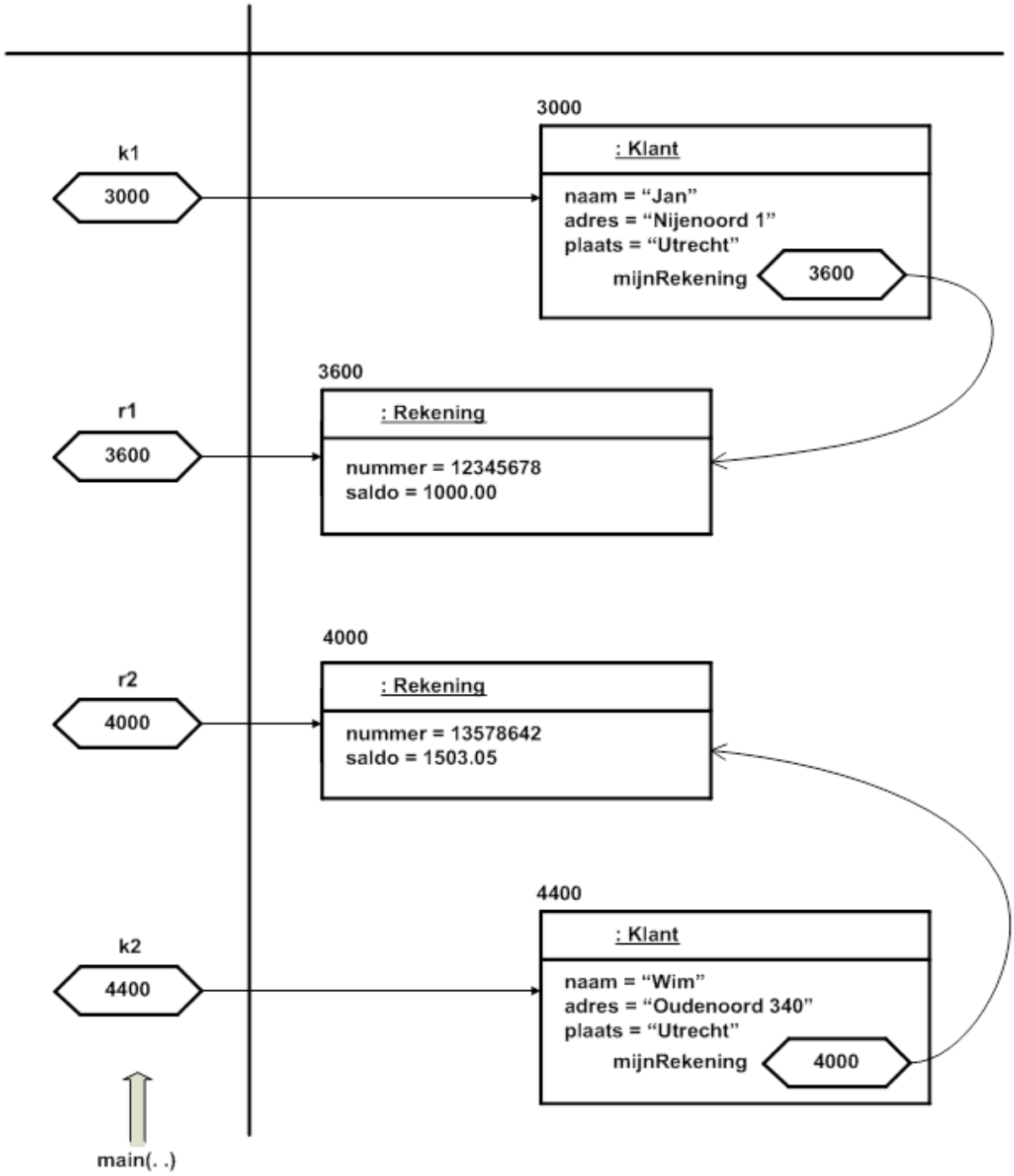
```
public class Main {
  public static void main(String[] arg) {
    // maak 2 objecten aan van klasse Klant:
    Klant k1 = new Klant("Jan", "Nijenoord 1", "Utrecht");
    Klant k2 = new Klant("Wim", "Oudenoord 340", "Utrecht");
    System.out.println(k1);
    System.out.println(k2);
    System.out.println();

    // we maken nu voor elk Klant-object een Rekening-object aan:
    Rekening r1 = new Rekening(12345678);
    Rekening r2 = new Rekening(13578642);

    // en maken vervolgens de koppeling:
    k1.setMijnRekening(r1);
    k2.setMijnRekening(r2);

    // storten kan nu alleen via de rekeninghouder
    k1.getMijnRekening().stort(1000);
    k2.getMijnRekening().stort(1503.05);

    System.out.println(k1);
    System.out.println(k1.getMijnRekening());        // Rekening-object
    System.out.println();
    System.out.println(k2);
    System.out.println(k2.getMijnRekening());        // Rekening-object
  }
}
```

Listing 25: Klasse Main maakt de koppeling tussen Klant- en Rekening-objecten

De bijbehorende uitvoer:

```
Jan, woont op Nijenoord 1, in Utrecht; er is nog geen rekening bekend
Wim, woont op Oudenoord 340, in Utrecht; er is nog geen rekening bekend

Jan, woont op Nijenoord 1, in Utrecht;
Op rekening 12345678 staat 1000.0
Op rekening 12345678 staat 1000.0

Wim, woont op Oudenoord 340, in Utrecht;
Op rekening 13578642 staat 1503.05
Op rekening 13578642 staat 1503.05
```

Het geheugen van je computer zal er bij uitvoeren van klasse Main er dan zo uitzien:



Figuur 6: het interne geheugen van de computer tijdens het runnen van methode 'main' uit listing 25

➔ MAAK NU EERST OPDRACHT 3_2 UIT HET WERKBOEK!

LES 5 – RELATIES TUSSEN KLASSEN (MEERVOUDIGE ASSOCIATIES)

1. INLEIDING

We hebben in de vorige les gezien dat er tussen klassen associaties kunnen bestaan. Die associaties geven dat er een relatie bestaat of kan bestaan tussen een object van klasse X en een object van klasse Y. In les 3 was dat concreet tussen klasse Rekening en Klant. We hebben daarbij alleen gekeken naar de uitwerking van associaties waar sprake was van een multipliciteit van maximaal 1. Bij zo'n koppeling zijn dus maximaal 2 objecten betrokken die aan elkaar verbonden zijn.

In de werkelijke wereld zijn er echter veel situaties te bedenken waar er veel meer objecten bij elkaar betrokken zijn. Je kunt denken aan een bedrijf waar tientallen werknemers werken. Of aan een student die meerdere cursussen per schooljaar volgt. Of aan een fabrikant die honderden verschillende producten produceert. Deze associaties worden in Java veelal geïmplementeerd met collecties. Een collectie is een verzameling van objecten. Eén van de collecties die binnen veelgebruikt is, is de lijst. De lijst ken je al van Python, maar in Java werkt dat syntactisch net weer iets anders.

Om goed met lijsten om te kunnen gaan hebben we een beter begrip nodig van typen, variabelen, declaraties en scopes dan tot nu toe is behandeld. In les 4 kijken we daarom eerst nog eens goed naar variabelen en typen in Java en behandelen daarna de lijst-structuur en de toepassing daarvan in associaties.

2. TYPEN, VARIABELEN, DECLARATIES EN SCOPES

Tot nu toe maakten we variabelen van type String, int en double. Een variabele in Java heeft:

- een naam.
- een type.
- een scope
- een waarde.

ⓘ *Anders dan in Python moet je in Java **altijd** het variabele-type opgeven bij declaratie!*

DE **NAAM** VAN EEN VARIABELE:

- bestaat uit letters (A..Z, a..z), cijfers (0..9), underscore (_) of dollarteken ($).
- begint niet met een cijfer.
- bevat geen andere tekens dan hierboven vermeld staan, ook geen spatie.
- Mag niet een keyword zijn (int, extends, public, etc).

Voorbeelden:

kostenPerPersoonPerDag	Correct

aantalPersonen	Correct
aantalDagen	Correct
totaleKosten	Correct
3appcld	Fout, begint met een cijfer
hoogte_van_het_blok	Toegestaan maar gebruikelijk is: 'hoogteVanHetBlok'
MAXIMUM	Toegestaan, hoofdletters duiden op een final variabele
naam2	Correct
wim@work.nl	Fout, @ is niet toegestaan
_lengte	Toegestaan maar gebruikelijk is 'lengte'

HET **TYPE** VAN EEN VARIABELE:

We hebben al gezien dat er binnen Java 2 soorten getallen zijn; **gehele getallen** zoals 34, -7 en 284838745 maar ook **floating point getallen** (kommagetallen) zoals 48.85639, 1.0 en 2.99792458e8. Maar binnen deze twee categorieën zijn er ook weer verschillende typen:

Gehele getallen:

Type	Aantal bytes	Minimum-waarde	Maximum-waarde
byte	1	-128	127
short	2	-32768	32767
int	4	-2147483648	2147483647
long	8	-9223372036854775808	9223372036854775807

Floating point getallen

Type	Aantal bytes	Minimum-waarde	Maximum-waarde	precisie
float	4	-3.4e38	3.4e38	6 (7 cijfers)
double	8	-1.79e308	1.79e308	15

De typen int en double worden het meest gebruikt. Naast deze typen zijn er ook nog **boolean** en **char**. Samen met de bovenstaande typen zijn dit de **primitieve typen** van Java. Primitieve typen zijn als het ware de basisbouwstenen van de taal Java. Klasse String is bijvoorbeeld gebaseerd op het primitieve type 'char'. Een string is namelijk opgebouwd uit een reeks van karakters. Primitieve typen kun je herkennen aan het feit dat je ze geheel met **kleine letters** schrijft.

DE **SCOPE** VAN EEN VARIABELE

Naast een naam heeft een variabele **impliciet** ook altijd een '**scope**'. Dat wil zeggen dat de variabele binnen een bepaalde omgeving gebruikt kan worden. De scope kan bijvoorbeeld zijn dat een variabele alleen geldig is binnen een 'klasse', 'object', 'methode' of soms alleen het code-blok waarin ze zijn gedeclareerd zoals een 'if-statement'. De plaats waar je een variabele declareert bepaalt wat de scope van een variabele is.

We onderscheiden vier soorten **scopes**:

- Een **klasse variabele** is **static** en wordt gedeeld door de klasse en alle objecten. We behandelen dit type variabele later in deze reader.
- Een **instantie** (object) **variabele** (oftewel: **attribuut**) begint te bestaan zodra het object waar hij bij hoort wordt aangemaakt, en wordt opgeruimd zodra dat object wordt opgeruimd.
- **Lokale variabele**(**n**) en **parameter**(**s**) van een methode: beginnen te bestaan zodra een methode wordt aangeroepen, en worden opgeruimd zodra de return is uitgevoerd.
- **Blok-variabelen** van een blok beginnen te bestaan zodra het blok wordt uitgevoerd, en worden opgeruimd zodra de sluit-accolade van hun blok is bereikt. Een blok kan zijn; een if-statement, switch, try-catch, for-loop, while-loop etc.

Java zoekt een variabele bij uitvoeren altijd eerst in de blok-scope, dan lokaal, instantie en als laatste in de klasse-scope. Als een scope niet bestaat, zal deze overgeslagen worden.

DE **WAARDE** VAN EEN VARIABELE:

Je kunt aan een variabele een waarde toekennen met een toekennings-statement. Een toekennings-statement kun je herkennen aan een = -teken met links en rechts 'iets' van hetzelfde type (het moet compatibel zijn). Er hoeft niet letterlijk hetzelfde type te staan.

Voorbeelden:

int i	=	7;	**Correct**, links en rechts zijn van het type int.
double d	=	i;	**Correct**, rechts is van type int, maar dat past ook in een double! Variabele d krijgt de waarde **7.0**
double d	=	3.7;	**Correct**, links en rechts zijn van het type double.
int i	=	d;	**Fout**, de compiler weigert dit. De variabele d verliest hierdoor mogelijk informatie achter de komma.
int i	=	**(int)**d;	**Correct**, variabele d verliest hier informatie achter de komma, maar met (int) geef je expliciet aan dat dit niet erg is. Deze conversie noemen we '**casten**'.
float f	=	4.3;	**Fout**, Java ziet kommagetallen standaard als double!
float f	=	**(float)**4.3;	**Correct**, met (float) geef je expliciet aan dat de double naar een float omgezet moet worden.
float f	=	4.3f;	**Correct**, met 'f' achter het getal geef je aan dat het om een float gaat en geen double is.

Vaak is het mogelijk een waarde van het ene type om te zetten naar een waarde van een ander type. Dit noemt men (type-)conversie. Er bestaat impliciete type-conversie, dat gaat vanzelf, en expliciete type-conversie d.m.v. een cast-opdracht. In de tabel hierboven is bijvoorbeeld de waarde van variabele 'd' expliciet van type double naar int omgezet:

```
double d = 3.7;
int i = (int)d;
System.out.println(i);        // geeft als uitvoer: 3
```

Type-conversie speelt ook een rol bij berekeningen met +, -, *, / en %. Voorbeelden:

7 * 3 = 21	int * int = int	
7 / 3 = 2	int / int = int	7 / 3 = 2.333 maar decimalen verdwijnen
7.0 / 3 = 2.333333	double / int = double	
7 / 3.0 = 2.333333	int / double = double	
(double)7 / 3 = 2.333333	double / int = double	Cast heeft voorrang t.o.v. berekeningen
143 / 60 = 2	int / int = int	
143 % 60 = 23	int % int = int	De rest bij deling van 143 door 60 is 23
14 % 5 = 4	int % int = int	De rest bij deling van 14 door 5 is 4
14 % 7.0 = 0.0	int % double = double	De rest bij deling van 14 door 7.0 is 0

Vermenigvuldigen (*), delen (/) en rest (%) hebben ook binnen Java voorrang op optellen en aftrekken. Bij bewerkingen met variabelen van verschillend type vindt impliciete type-conversie plaats. De meeste bewerkingen associëren van **links naar rechts**. Bijvoorbeeld:

```
    3 * 4 / 6                      1 + 3 / 4.0 - 3 * 9
=   (3 * 4) / 6                =   1 + (3 / 4.0) - (3 * 9)
=   12 / 6                     =   1 + 0.75 - 27
=   2 (int / int = int)        =   -25.25 (int+double-int=double)
```

Het = -teken associeert echter van **rechts naar links**. Bijvoorbeeld:

```
        int x = 1;
        int y = 2;
        int z = 3;

        x = y = z;
=       x = (y = z);        gelijk aan:   y = z; (1e stap)
                                          x = y; (2e stap)
```

➔ MAAK NU EERST OPDRACHT 4_1 UIT HET WERKBOEK!

3. BOOLEAN EXPRESSIES

Een boolean expressie heeft als uitkomst true of false. Voorbeelden van boolean expressies zijn hieronder **vetgedrukt** (let op het hele grote verschil tussen een = en een ==):

```
int nr = 12345;
nr == 12345                    boolean expressie met uitkomst true

int leeftijd = 20;
leeftijd >= 18                 boolean expressie met uitkomst true
leeftijd < 18                  boolean expressie met uitkomst false

String s1 = "tekst";
s1.equals("andereTekst")       boolean expressie met uitkomst false
```

ⓘ *Methode equals gebruik je om 2 strings met elkaar te vergelijken, zie ook les 6.*

Bij een if-statement moet je **altijd** een boolean expressie gebruiken:

```
if (boolean expressie) {
      doeIets;                  // als de boolean expressie == true
}                               // je doet niets, als de boolean expressie == false
```

Omdat uit een boolean expressie altijd true **of** false moet komen hoef je bij een if-else-statement ook maar één boolean expressie te gebruiken:

```
if (boolean expressie) {
      doeIets;                  // uitgevoerd als expressie == true
} else {
      doeIetsAnders;            // uitgevoerd als expressie == false
}
```

Voorbeeld:

```
int leeftijd = 20;
if (leeftijd < 18) {            // als (leeftijd < 18) == true
      System.out.println("minderjarig");
} else {                        // als (leeftijd < 18) == false
      System.out.println("meerderjarig");
}
```

Een boolean expressie kan worden gemaakt met vergelijkings operatoren.

>	is groter dan	**ezelsbruggetje**: kun je geen K(leiner) van maken
>=	is groter dan of gelijk aan	
<	is kleiner dan	**ezelsbruggetje**: kun je een K(leiner) van maken
<=	is kleiner dan of gelijk aan	
==	is gelijk aan	
!=	is niet gelijk aan	

Voorbeeld:

(leeftijd >= 18) betekent: (leeftijd > 18) of (leeftijd == 18)

LOGISCHE OPERATOREN

Meer ingewikkelde boolean expressies kunnen worden gemaakt met logische operatoren. Dit zijn de logische and, or en not:

&&	and / en
||	or / of
!	not / niet

Voorbeelden

a. `!(leeftijd >= 18)` : leeftijd is niet groter en ook niet gelijk aan 18
b. `!(leeftijd >= 18)` : dwz (leeftijd < 18)
c. `(x >= 10 && x <= 20)` : x ligt tussen 10 en 20 (wiskundig: 10 <= x <= 20)
d. `!(x >= 10 && x <= 20)` : x ligt niet tussen 10 en 20 (dwz x < 10 of x > 20)
e. `!(x >= 10 && x <= 20)` : (x < 10 || x > 20)
f. `(ch == '0' || ch == '1')` : ch is een `'0'` of ch is een '1'
g. `!(ch == '0' || ch == '1')` : `(ch != '0' && ch != '1')`

Bovenstaande voorbeelden zijn eigenlijk boolean expressies die op zichzelf weer uit twee losse boolean expressies samengesteld zijn: (x >= 10 && x <= 20) bestaat uit (x >= 10) en (x <= 20). Beiden leveren een true of false op. Je zou de eerste expressie 'A' kunnen noemen en de tweede 'B'. Dan kun je de onderstaande waarheidstabel gebruiken om te bepalen wat de mogelijke resultaten kunnen zijn:

A	**B**	**A && B**	**A || B**	**!A**
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Let op: de '&&' en '||' operator in Java zijn **short evaluation** operators. De uitkomst van de voorwaarde aan de linkerkant van de operator is bepalend of de voorwaarde rechts van de operator zal worden uitgevoerd. In de boolean expressie (x != 0 && y = 1/x) zal y bijvoorbeeld niet worden uitgerekend als x nul is. Dat gebeurt niet, omdat eerst x != 0 wordt uitgerekend. En 0 != 0 is false. Dan kan er nooit meer uit de hele boolean expressie true uitkomen. Daarom stopt de berekening in dat geval. De && stopt meteen als de voorwaarde links false is. De || stopt meteen als de voorwaarde links gelijk is aan true.

➔ MAAK NU EERST OPDRACHT 4_2 UIT HET WERKBOEK!

4. KLASSE ARRAYLIST

In Java zijn er verschillende soorten lijsten; iedere soort heeft z'n eigen specifieke kenmerken. We gebruiken in deze les uitsluitend de **ArrayList**. Dit is een lijst waarbij elke item dat je toevoegt in principe aan het einde van de lijst erbij komt, tenzij je anders aangeeft. De volgorde van de lijst is ook gelijk aan de volgorde waarin je items aan de lijst hebt toegevoegd!

Een lijst kun je, net als in Python, ook gewoon gebruiken om allerlei gegevens in op te slaan. Je zou bijvoorbeeld een lijst kunnen maken waarin je de cijfers van de afgelopen tentamens in opslaat. We maken voor dit eenvoudige programma gewoon een simpele klasse met methode main(..):

```java
import java.util.ArrayList;   // import nodig om een lijst te gebruiken!

public class Voorbeeld1 {
  public static void main(String[] args) {
    ArrayList cijferlijst = new ArrayList();
    cijferlijst.add(6.9);
    cijferlijst.add(8.1);
    cijferlijst.add(5.2);
  }
}
```

Listing 26: Een eenvoudige lijst (ArrayList) met doubles

Voordat je een lijst kunt gebruiken moet je de betreffende soort eerst importeren, dat is de eerste regel in het voorbeeld. De **package** 'java.util' bevat allerlei handige klassen, waaronder diverse soorten collecties. We zullen deze package nog wel vaker tegenkomen, dus het is handig om deze package eens te bestuderen. Dat kan via de Java API documentatie, https://docs.oracle.com/javase/9/docs/api/. Klik links bovenin het scherm op de package java.util om een overzicht te krijgen van de klassen die erin zitten!

Als we vervolgens weer willen uitprinten wat er in de lijst zit kunnen we een for-loop of een foreach-loop gebruiken. Beide loops (of lussen) heb je al eens voorbij zien komen in Python. We doorlopen de lijst op beide manieren en printen alle lijst-items uit:

```java
import java.util.ArrayList;

public class Voorbeeld2 {
  public static void main(String[] args) {
    ArrayList cijferlijst = new ArrayList();
    cijferlijst.add(6.9);
    cijferlijst.add(8.1);
    cijferlijst.add(5.2);

    for (int i=0; i < cijferlijst.size(); i++) {      // for-loop
       System.out.print(cijferlijst.get(i) + "   ");
    }
    System.out.println();

    for (Object item : cijferlijst) {                 // foreach-loop
      System.out.print(item + " * 2 = ");
      System.out.println((double)item * 2);           // voor rekenen cast nodig!
    }
  }
}
```

Listing 27: Printen van een lijst met for-loop en foreach-loop

De eerste for-lus blijft herhalen zolang variabele 'i', die begint bij 0, kleiner is dan de grootte van de cijferlijst (size). Aangezien dat exact 3 keer het geval is (voor i = 0, 1 en 2), zal de body van de for-lus ook precies 3 keer uitgevoerd worden. Omdat het eerste item in de lijst net als

in Python altijd op index '0' staat kunnen we variabele 'i' meteen ook gebruiken voor het opvragen van een lijst-item. Dat is met het statement cijferlijst.get(i); gedaan.

De foreach-lus is syntactisch eenvoudiger. Variabele 'item' krijgt eerst de waarde van het eerste cijfer in de lijst (6.9). Dan wordt 'item' geprint. Als er nog een cijfer in de lijst zit (in dit geval wel; 8.1) krijgt variabele 'item' de waarde van dit volgende getal. Hier hoef je niets voor te doen, dat gaat automatisch. Dit proces herhaalt zich zolang er cijfers in de lijst zitten. Variabele 'item' is van het type **Object** om aan te geven dat er van alles in de lijst kan zitten!

Let op: De lijst kan allerlei soorten informatie bevatten (doubles, strings, ints en andere objecten). Je kunt elk soort object printen, maar je bijvoorbeeld kunt niet met elk soort variabele rekenen! Als je dus een specifieke bewerking wilt uitvoeren op een variabele moet je 'm eerst naar het juiste type casten. Een voorbeeld hiervan is opgenomen in de foreach-lus. Alles bij elkaar resulteert het voorbeeld in deze uitvoer:

```
6.9   8.1   5.2
6.9 * 2 = 13.8
8.1 * 2 = 16.2
5.2 * 2 = 10.4
```

De foreach-lus is eenvoudiger maar tegelijk ook minder krachtig dan de eerste variant omdat je geen invloed hebt op de volgorde (altijd voorwaarts door de lijst) of op de stapgrootte (altijd één voor één alle lijst-items langs).

5. GENERICS

De lijst die we zojuist hebben gevuld met cijfers kan uitgebreid worden met allerlei objecten die we er maar in zouden willen stoppen. Dat kan soms handig zijn, maar in de praktijk kan dat ook voor ongewenste situaties zorgen:

```
import java.util.ArrayList;

public class Voorbeeld3 {
  public static void main(String[] args) {
    ArrayList cijferlijst = new ArrayList();
    cijferlijst.add(6.9);
    cijferlijst.add(8.1);
    cijferlijst.add("dit is geen cijfer");
    cijferlijst.add(new Klant("Henk", "Nijenoord 1", "Utrecht"));

    for (int i=0; i < cijferlijst.size(); i++) {
      System.out.print(cijferlijst.get(i) + "   ");
    }
  }
}
```

Listing 28: Een lijst kan (soms ongewenst) van alles en nog wat bevatten

De uitvoer is dus ook een mengelmoes:

```
6.9  8.1  dit is geen cijfer  Henk woont op Nijenoord 1 in Utrecht
```

Als dit precies is wat je wilt, dan is er niets aan de hand. Maar als je wilt voorkomen dat iemand objecten toe kan voegen die je niet in de lijst wilt hebben kun je de lijst **parametriseren**. Dit is mogelijk omdat ArrayList een **generieke** (generic) klasse is: ArrayList<E>, waarbij de E in de Java-documentatie altijd staat voor een willekeurig type. Je mag dus de E vervangen door een type van jouw keuze. Stel dat we een lijst willen hebben waar alleen maar doubles in mogen, dan kan je dat als volgt doen:

```
ArrayList<Double> cijferlijst = new ArrayList<Double>();
```

Let op dat de lijst hier gemaakt is voor het type '**Double**' (met een hoofdletter). Dat komt omdat je in Java (nog) geen lijsten met primitieve-typen kunt maken. Dus ArrayList<**double**> is **niet toegestaan!** Omdat je vaak wel getallen in een lijst wilt kunnen opslaan heeft men voor elk primitieve type een **wrapper klasse** gemaakt.

Voor byte, short, int, long, float, double, boolean en char zijn respectievelijk Byte, Short, Integer, Long, Float, Double, Boolean en Character als **wrapper klasse** aanwezig. Deze klassen hebben allemaal handige methoden voor het betreffende type. Klasse Integer heeft bijvoorbeeld onder andere:

```
public static int parseInt(String s)
public static Integer valueOf(String s)
public int intValue()
public static String toHexString(int i)
```

Als je een lijst aanmaakt van primitieve typen, dan hoef je de waarde niet zelf in- of uit te pakken (wrappen), dat doet Java voor je. Dat proces heet **auto-(un)boxing**:

```
import java.util.ArrayList;
public class Voorbeeld4 {
  public static void main(String[] args) {
    ArrayList<Double> cijferlijst = new ArrayList<Double>();
    cijferlijst.add(6.9);                          // auto-boxing (geen wrap nodig)
    cijferlijst.add(8.1);

    double index1 = cijferlijst.get(1);            // auto-unboxing (geen cast nodig)
    System.out.println(index1);

    for (Double cijfer : cijferlijst) {
      System.out.print(cijfer + "  ");
    }
  }
}
```

Listing 29: Bij een for-each lus kun je dus het juiste type variabele gebruiken (Double) zonder zelf te hoeven casten

In het voorbeeld is ook te zien dat het return-type van een methode ook verandert! Eerst had methode **get(..)** als return-type **Object**. Maar variabele **index1** is van het type double en dit gaat nu gewoon goed. Dat komt omdat deze methode eigenlijk als return-type het type E heeft. Als je niets opgeeft is dat altijd **Object**.

Naast lijsten van doubles, strings en ints kun je op dezelfde wijze ook lijsten van objecten maken. Bijvoorbeeld een lijst met meerdere Klant-objecten:

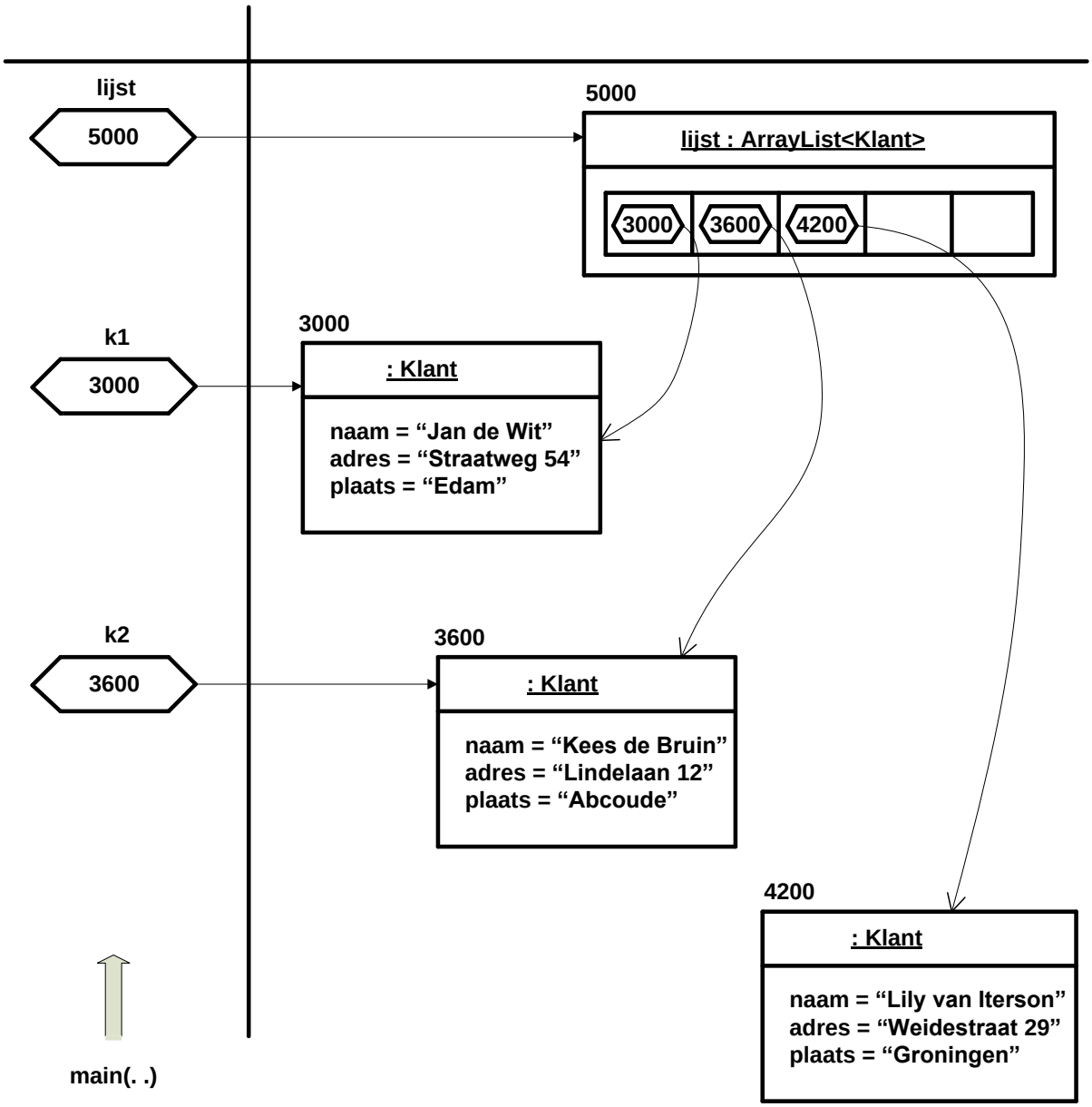
```
import java.util.ArrayList;

public class Voorbeeld5 {
  public static void main(String[] arg) {
    ArrayList<Klant> lijst = new ArrayList<Klant>();

    Klant k1 = new Klant("Jan de Wit","Straatweg 54","Edam");
    Klant k2 = new Klant("Kees de Bruin","Lindelaan 12", "Abcoude");

    lijst.add(k1);
    lijst.add(k2);
    lijst.add(new Klant("Lily van Iterson","Weidestraat 29","Groningen"));

    int aantal = lijst.size();
    System.out.println("De lijst bevat "+aantal+" klanten");
    for (Klant k : lijst) {
      System.out.println(k);    // dus: k.toString()
    }
  }
}
```

Listing 30: Een lijst met Klant-objecten

De code van Voorbeeld5 heeft eigenlijk dezelfde werking als Voorbeeld4, maar de lijst is nu geheel voor Klant-objecten gereserveerd. De compiler zou het niet toestaan om in deze lijst een ander soort variabele toe te voegen. Voor de volledigheid is hieronder de uitvoer opgenomen:

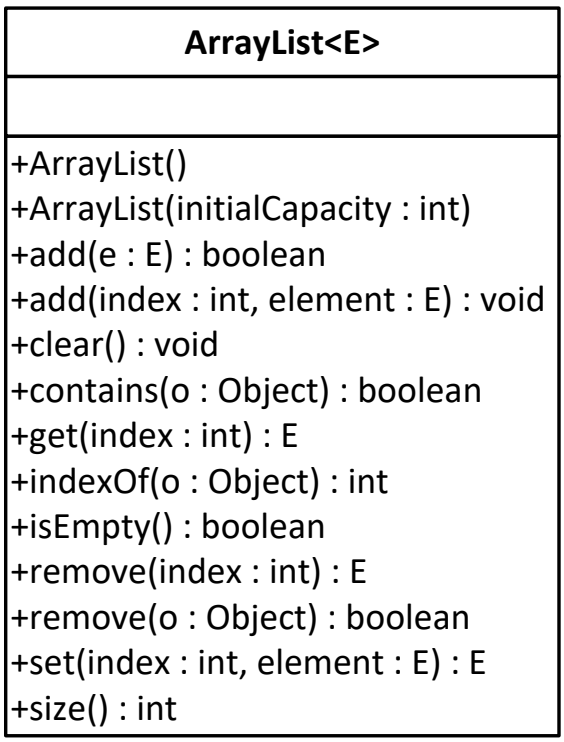
```
De lijst bevat 3 klanten
Jan de Wit, woont op Straatweg 54, in Edam
Kees de Bruin, woont op Lindelaan 12, in Abcoude
Lily van Iterson, woont op Weidestraat 29, in Groningen
```

Besef wel dat bij dit voorbeeld sprake is van lijst met begin-adressen. Dus de lijst zelf bevat referenties naar de verschillende Klant-objecten. Deze objecten staan ergens in het geheugen. Zie ter verduidelijking ook de geheugen-afbeelding op de volgende pagina.



Figuur 7: Het geheugen van de computer na toevoegen van de Klant-objecten aan de lijst in Voorbeeld5

Inmiddels hebben we al een aantal methoden van ArrayList voorbij zien komen. Klasse ArrayList heeft echter nog veel meer methoden in de aanbieding voor het manipuleren van een lijst:



Je kunt ook de documentatie bekijken voor een volledig overzicht van deze klasse! Lees voor 'E' in dit UML-diagram dus het type wat jij in de lijst wilt stoppen. Wil je geen specifiek type opgeven? Vervang dan 'E' door 'Object'.

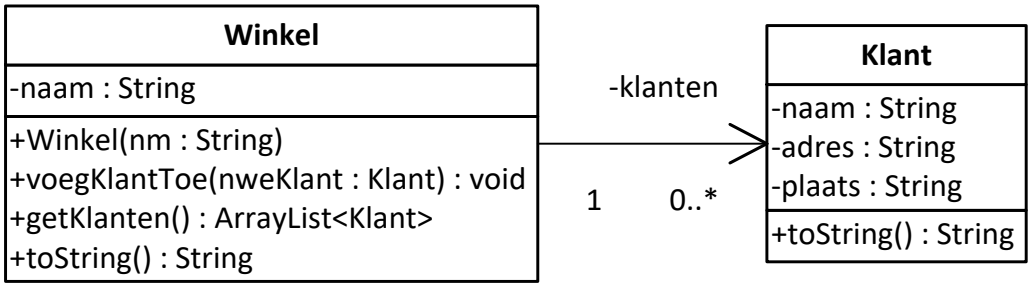
➔ MAAK NU EERST OPDRACHT 4_3 UIT HET WERKBOEK!

6. MEERVOUDIGE ASSOCIATIES

Tot nu toe hebben we alleen **enkelvoudige** associaties behandeld. Dat zijn associaties waarbij een relatie aan beide kanten uit 1 object bestaat. Eén object **heeft** dus een relatie met **maximaal** één ander object. Maar in veel gevallen situaties heb je ook **meervoudige** associaties nodig. Bij een meervoudige associatie heeft een object een koppeling met 0 of meer andere objecten. Denk bijvoorbeeld aan een klas die uit meerdere studenten bestaat. Of een auto die (meestal) vier wielen heeft. Of een supermarkt die veel verschillende producten verkoopt.

Voor het bijhouden van zo'n koppeling naar meerdere objecten van hetzelfde gebruikt men vaak een lijst. Wij passen daarvoor de veelgebruikte ArrayList toe. We doen dat in een voorbeeld van een winkel die meerdere klanten heeft.

De klasse Winkel is in dit voorbeeld alleen geschikt om Klant-objecten toe te voegen. Het verwijderen van objecten uit een lijst is wat complexer en behandelen we van les 6.



Klasse Klant is inmiddels bekend uit eerdere voorbeelden. De klasse is in deze associatie ook niet verantwoordelijk, want de pijl vertrekt vanuit klasse Winkel. Het UML-diagram van klasse Klant is daarom beperkt gehouden, wat wil zeggen dat niet alle getters, setters en constructors zijn opgenomen. Klasse **Winkel** is **wel verantwoordelijk** en is hieronder opgenomen.

```
import java.util.*;

public class Winkel {
  private String naam;
  private ArrayList<Klant> klanten;

  public Winkel(String nm) {
    naam = nm;
    klanten = new ArrayList<Klant>();
  }

  public void voegKlantToe(Klant nweKlant) {
    klanten.add(nweKlant);
  }

  public ArrayList<Klant> getKlanten() {
    return klanten;
  }

  public String toString() {
    return "winkel " +naam+ " heeft " +klanten.size()+ " klanten\n";
  }
}
```

Listing 31: Klasse Winkel

Bij enkelvoudige associaties gebruikten we steeds een **set**- en **get**-methode om de **koppeling** tussen twee objecten te maken. Hier is de set-methode **vervangen** door een **voegKlantToe(Klant)** methode, en methode **getKlanten()** waarmee de lijst van klanten van een winkel kan worden opgevraagd.

Merk op dat het in deze winkel mogelijk is dat er klanten voorkomen met dezelfde naam, adres en woonplaats! De winkel is hier niet tegen beveiligd. Voorbeeld6 test klasse Winkel:

```
import java.util.ArrayList;

public class Voorbeeld6 {
  public static void main(String[] arg) {
    Winkel w = new Winkel("Jumbo Utrecht Parkwijk");

    w.voegKlantToe(new Klant("Jan de Wit", "Straatweg 54", "Edam"));
    w.voegKlantToe(new Klant("Kees de Bruin", "Lindelaan 12", "Abcoude"));

    System.out.println(w + "\n"); // dus: w.toString() + "\n" (newline)

    System.out.println("De winkellijst bevat de volgende klanten:");

    ArrayList<Klant> klantenLijst = w.getKlanten();
    for (Klant klant : klantenLijst) {
      System.out.println(klant);
    }
  }
}
```

Listing 32: Voorbeeld6 toont de klantenlijst van de winkel

De uitvoer van dit voorbeeld toont de toString() van Winkel en print de lijst van de winkel:

```
winkel Jumbo Utrecht Parkwijk heeft 2 klanten

De winkellijst bevat de volgende klanten:
Jan de Wit, woont op Straatweg 54, in Edam
Kees de Bruin, woont op Lindelaan 12, in Abcoude
winkel Jumbo Utrecht Parkwijk heeft 2 klanten
```

Overigens kan iemand anders (in dit geval klasse Voorbeeld6) op deze manier wel de controle over de lijst met Klant-objecten overnemen! Voorbeeld6 zou bijvoorbeeld de lijst kunnen leegmaken (methode **clear()**), wat niet erg wenselijk is. In een Object Oriented omgeving is een object zelf eigenaar en beheerder (**verantwoordelijke**) voor de koppeling.

Dit kan opgelost worden door methode getKlanten() een **unmodifiable list** te laten returnen! Omdat we nog niet alle theorie besproken hebben is dat nu nog een beetje omslachtig:

```
public ArrayList<Klant> getKlanten() {
  return new ArrayList<Klant>(Collections.unmodifiableList(klanten));
}
```

Voor deze les is dat echter nog niet noodzakelijk, dus je mag het eventueel voor nu ook weer vergeten.

LES 6 – STRINGS, EQUALS, ARRAYLIST EN CONTAINS

1. INLEIDING

We hebben de afgelopen lessen gebruik gemaakt van strings, lijsten, primitieve typen en referentie typen. De termen zeggen je misschien niet allemaal evenveel, maar je hebt ze wel toe moeten passen in de diverse opdrachten. Deze les kijken we wat beter naar de technieken die daarbij op de achtergrond zijn toegepast.

2. HET PRIMITIEVE TYPE CHAR

Een char is in Java het variabele-type van 2 bytes waar je tekens (karakters) in kunt opslaan. Dat kunnen letters, cijfers en leestekens zijn, maar ook een spatie, enter etc. Er zijn wereldwijd natuurlijk zeer veel karakters. Een **character set** (karakterset) is een verzameling van karakters die in verschillende talen gebruikt worden zoals Nederlands, Engels, Chinees etc. Als je vervolgens aan elk karakter een nummer toekent, dan kun je spreken van een **coded character set**.

Omdat de computer alleen maar getallen begrijpt heb je om tekst in het geheugen van de computer op te slaan zo'n coded character set nodig. Bij de ontwikkeling van Java heeft men ervoor gekozen om de Unicode-karakterset te gebruiken om karakters mee te representeren. In de Unicode-karakterset heeft de kleine letter 'a' bijvoorbeeld het getal 97 gekregen! Dit getal noem je een **code point**. Dit kun je ook zien als je de letter 'a' niet uitprint als karakter maar als int:

```
char karakter = 'a';
System.out.println(karakter);       // geeft als uitvoer: a
System.out.println((int)karakter);  // geeft als uitvoer: 97
```

De truc hier is dat bij het printen een char automatisch omgezet wordt naar het **code point** van de **coded character set** van je besturingssyteem en (bijvoorbeeld) Windows dus netjes de letter 'a' op het scherm plaatst. Terwijl van een int alleen de tekstuele representatie wordt getoond.

Unicode was oorspronkelijk zo ontworpen dat elk karakter 16 bits besloeg omdat de meeste machines indertijd 16-bits PC's waren. Dat is de reden waarom een char in Java 2 bytes (16 bits) gebruikt. Hiermee kun je echter 'slechts' 65563 (2^16) karakters weergeven. Dat bleek niet genoeg en is de Unicode-karakterset verder uitgebreid. Er zijn dus nu karakters met een code point die niet in 16 bits past. Het is echter niet mogelijk om dan maar 'even' het char-type uit te breiden naar 4 bytes omdat dan wereldwijd miljoenen programma's zullen crashen…

Om toch alle Unicode karakters te kunnen representeren past Java nu intern de UTF-16 **character encoding** toe. Dit houdt in dat alle karakters **minimaal** 16 bits (1 char) beslaan. Maar als dat niet genoeg is kan dit uitgebreid worden naar 32 bits (2 chars). Java noemt deze

karakters '**supplementary characters**'. Het voert te ver om dit nu uitgebreid te bespreken, maar je kunt er hier meer over lezen. We beperken ons in deze cursus tot karakters die 1 char beslaan (char-literal)!

De meeste karaktersets -ook Unicode- hanteren voor de eerste 127 code points (nummers) karakters dezelfde tekens, gebaseerd op de ASCII-schema dat tot 2007 het meest werd toegepast.

Als je in je code char-literals wilt gebruiken doe je dat tussen enkele quotes: ' '

```
char ch1 = 'a';
char ch2 = '3';
char ch3 = '+';
char ch4 = 'A';
```

Een speciale char maak je met escape sequence: \

```
char ch5 = '\'';
char ch6 = '\\';
char ch7 = '\"';
char ch8 = '\n';          // newline
char ch9 = '\t';          // tab
char ch10= '\u20AC';      // Euro-teken in hexadecimale Unicode
char ch11= '€';
```

De klasse String in Java kun je gebruiken om een tekst oftewel een string-object van te maken. Een string is niets meer of minder dan een lijst van chars.

3. DE KLASSE STRING

De klasse String is één van de basisklassen van de taal Java. In variabelen van het type String kan tekst opgeslagen worden.

String
+length() : int +charAt() : char +compareTo(s : String) : int +equals(o : Object) : boolean +indexOf(c : char) : int +toUpperCase() : String +toLowerCase() : String +$valueOf(i : int) : String +$valueOf(d : double) : String

```
String s0 = null;
String s1 = "Hallo ";
String s2 = "Hans";
String s3 = "";
String s4 = new String("appeltaart");
```

Merk op dat string s0 en s3 nogal van elkaar verschillen. De tweede is namelijk een lege string. Dat wil zeggen: er staan geen karakters in, maar er bestaat in het geheugen **wel** een String-object. Dit in tegenstelling tot s0 die null is. Er is dus wel een variabele s0, maar deze verwijst naar geen enkel object in het geheugen. Als je daar methoden op aanroept krijg je dan ook een **NullPointerException** want je kunt geen bewerkingen uitvoeren op iets dat er niet is.

String **concatenatie** (plakken) van strings kan met string-expressies of het plus-teken (+):

```
s1 + s2                    = "Hallo Hans"
s1 + "\n" + s2             = "Hallo \nHans"
"appel" + "taart"          = "appeltaart"
"route" + 66               = "route66"
s3 + 2 + 6                 = "26"
"47" + 1 + 1               = "4711"
4 + 7 + "11"               = "1111"
s4 + "-met-room"           = new String("appeltaart-met-room")
```

In de bovenstaande voorbeelden zie je dat getallen ook aan een tekst geplakt kunnen worden. Het **resultaat** is dan in alle gevallen een **string**! Het komt echter ook vaak voor dat je een string wilt omzetten naar een getal. Dat noem je **string-conversie** en is iets complexer:

```
String s4 = "123"
String s5 = "45.6"
int i    = Integer.parseInt(s4);          // i is het getal 123
double d = Double.parseDouble(s5);        // d is het getal 45.6
```

Hier maak je gebruik van de wrapper-klassen Integer en Double waar de **static** methoden **parseInt(..)** en **parseDouble(..)** in staan. Static methoden zijn methoden die je kunt aanroepen zonder dat er een object van die klasse gemaakt hoeft te worden. We komen hier later nog op terug.

We bespreken een aantal voorbeelden van de **string-methoden**. Uitgangspunt is deze code:

```
String s1 = "amsterdam";
String s2 = "Rotterdam";
String s3 = "Den Haag";
```

length()	**s1**.length() is het aantal tekens van string **s1**, en heeft waarde 9. Ook de waarde van **s2**.length() is 9, terwijl de waarde van **s3**.length() gelijk is aan 8 (er zijn 7 letters en een spatie).
charAt()	**s3**.charAt(0) is het ***character*** dat staat op de voorste index (= index 0) en dus is de waarde van **s3**.charAt(0) gelijk aan 'D'. De waarde van **s3**.charAt(1) is 'e'. En van **s3**.charAt(3) is het ' ' (een spatie).
equals(Object)	**s1**.equals("Amsterdam") levert false als waarde op, omdat Java ***case-sensitive*** is (dat betekent dat Java verschil ziet tussen hoofdletters en kleine letters).
compareTo(String)	De waarde van **s1**.compareTo("Amsterdam") is een positief getal, omdat "amsterdam" lexicografisch groter is dan "Amsterdam". De waarde van **s2**.compareTo("rotterdam") is een negatief getal, omdat "Rotterdam" lexicografisch kleiner is dan "rotterdam". Hoofdletters hebben dus een kleinere Unicode/ASCII-waarde dan kleine letters!
toUpperCase()	De waarde van **s3**.toUpperCase() is "DEN HAAG" omdat alle kleine letters dan worden vervangen door hoofdletters.

Misschien is je in de voorbeeldcode opgevallen dat we strings aanmaken **zonder** een **constructor** te gebruiken. Misschien vind je dat logisch omdat dit bij getallen en chars ook mag, maar dat is het niet. Klasse string is namelijk geen primitief type, maar een klasse! Je maakt er dus **objecten** van aan! Klasse String is de enige klasse dit mogelijk is! Zo'n string heet een ***string-literal***. Je kunt ook String-objecten creëren met. een constructor. Deze twee soorten strings gedragen zich verschillend. Bestudeer onderstaande voorbeeld:

```
public class Voorbeeld1 {
  public static void main(String[] arg) {
    String s1 = new String("appeltaart");       // s1 via constructor
    String s2 = "appeltaart";                   // s2 is string-literal
    String s3 = new String("boterkoek");        // s3 != s2
    s3 = "appel" + "taart";                     // s3 == s2 en s3 != s1

    // Hier vraag je naar de adressen:
    if (s1 == s2)
      System.out.println("beginadres van s1 en s2 hetzelfde");
    else
      System.out.println("beginadres van s1 en s2 verschillend");

    // Hier vraag je naar de inhouden:
    if (s1.equals(s2) == true)
      System.out.println("inhoud van s1 en s2 hetzelfde");
    else
      System.out.println("inhoud van s1 en s2 verschillend");
  }
}
```

Listing 33: String, string-literal en strings

De uitvoer is kort maar krachtig:

```
beginadres van s1 en s2 verschillend
inhoud van s1 en s2 hetzelfde
```

Als je zonder het keyword **new** (dus zonder constructor) objecten maakt van klasse String, dan is dat object van de soort de ***string-literal***. Hiervan bestaat er altijd **slechts één** in het interne geheugen. **Let op:** string-literals met dezelfde tekst verwijzen dus allemaal naar hetzelfde object! Alle strings, die wel met **new** worden aangemaakt, hebben **elk hun eigen beginadres** in het interne geheugen.

Bij het aan elkaar plakken van strings wordt een nieuwe string aangemaakt. Het aan elkaar plakken van (uitsluitend) **string-literals** levert ook weer een **string-literal** op als resultaat. Daarom geldt dat s2 == s3. In alle andere gevallen wordt een nieuw object gemaakt.

De variabelen s1, s2, s3 zijn **references** (en dus begin-adressen). Met s1 == s2 worden de references vergeleken. Die zijn verschillend want ze wijzen naar (of: zijn het beginadres van) verschillende objecten. De uitvoer bewijst dit. Zie ook de afbeelding van het geheugen op de

hieronder! Echter, de inhoud is gelijk. Met **s1.equals(s2)** wordt de inhoud van de objecten vergeleken. Die is **hetzelfde**, zie opnieuw de uitvoer.

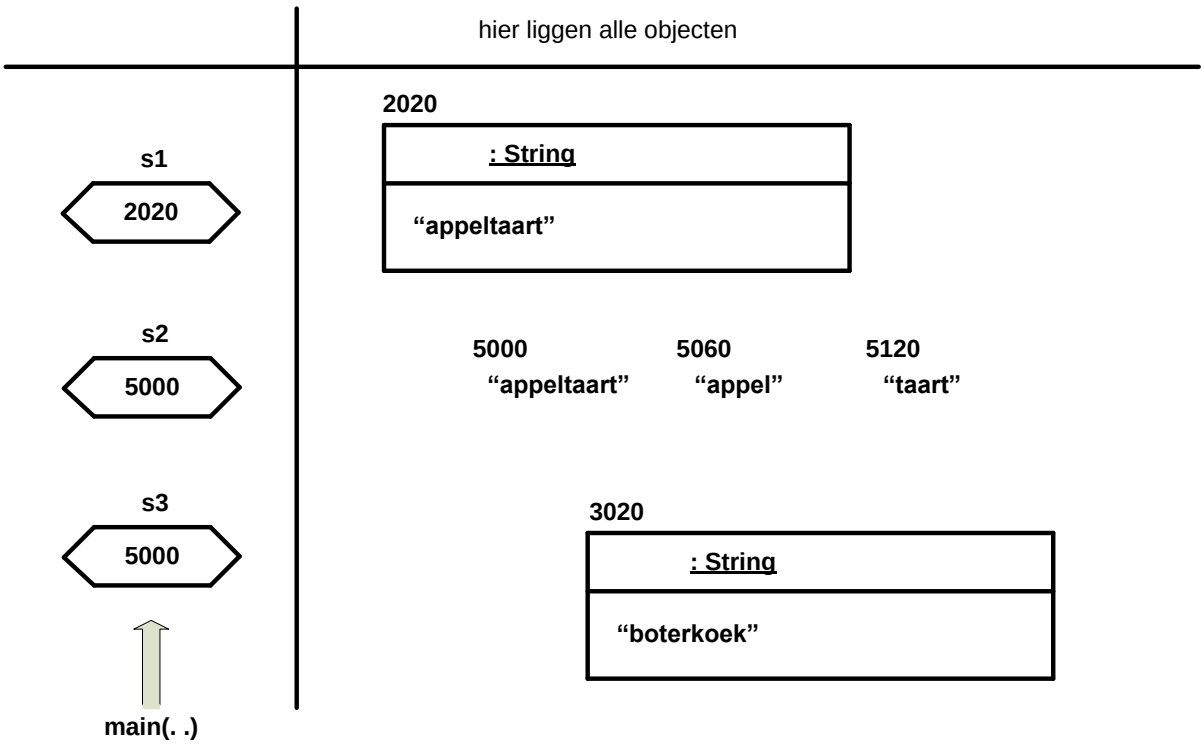
Uit dit alles kunnen we nog iets concluderen, namelijk dat strings **immutable** zijn. Dat wil zeggen dat een string, zodra deze eenmaal gemaakt is, niet meer kan wijzigen. Op deze manier kunnen strings eenvoudig gedeeld worden tussen meerdere references.

```
String s3 = new String("boterkoek");
s3 = "appel" + "taart";
```

De reference-variabele s3 krijgt eerst als waarde het begin-adres van het String-object "boterkoek" (in het plaatje: 3020). Vervolgens is er sprake van 3 string-literals; "appel", "taart" en "appeltaart". Door de concatenatie komt uit de eerste 2 **string-literals** de string "appeltaart" tevoorschijn. Maar deze bestaat al als string-literal! Dus eigenlijk vindt deze toekenning plaats:

```
s3 = "appeltaart";
```

Dat houdt in dat er **niets** wordt gewijzigd aan de string “boterkoek”. Er wordt overgegaan op een **nieuwe** string: "appeltaart", de al bestaande string-literal (in het plaatje: adres 5000):



Figuur 8: Het geheugen van de computer direct na uitvoeren van Voorbeeld1

Een **reference-variabele** van type String (zoals **s1**) bevat het (begin) adres van een **object** van klasse String. Het **object**, de rechthoek in het plaatje, bevat de tekst (“appeltaart”).

➔ MAAK NU EERST OPDRACHT 6_1 UIT HET WERKBOEK!

4. METHODE EQUALS

Voor twee strings kan worden getest op gelijkheid met methode equals. Dit is dus een inhoudelijke vergelijking. Ook voor iedere andere klasse kan worden getest of twee objecten dezelfde inhoud hebben. Daarvoor moet je de methode equals(Object) uitschrijven. Net als methode toString() kan je de methode equals(Object) in alle klassen toevoegen waar je dat wilt.

Ter illustratie gebruiken we klasse Klant, die we al eerder voorbij hebben zien komen. We nemen hier de meest eenvoudige vorm, zonder associaties:

```
public class Klant {
  private String naam;
  private String adres;
  private int leeftijd;

  public Klant(String nm, String adr, int lft) {
    naam = nm;
    adres = adr;
    leeftijd = lft;
  }

  /* setters & getters zijn in dit voorbeeld niet nodig */

  public String toString() { return naam+ ", " +adres+ ", " +leeftijd; }
}
```

Listing 34: Klasse Klant met alleen een constructor

Net als methode toString kun je ook de methode equals al aanroepen Zonder uitschrijven van de methode equals levert de volgende vergelijking false op:

```
Klant k1 = new Klant("Jan", "Nijenoord 1", 55);
Klant k2 = new Klant("Jan", "Nijenoord 1", 55);

boolean vergelijking1 = k1.equals(k2);
boolean vergelijking2 = k1 == k2;

System.out.println(vergelijking1);        // uitvoer: false
System.out.println(vergelijking2);        // uitvoer: false
```

Als je methode equals **niet** uitschrijft krijg je een **standaardvergelijking**. Daarbij kijkt Java of de references k1 en k2 naar hetzelfde object in het geheugen wijzen! Dat is **niet** het geval want er zijn **2** Klant-objecten gemaakt. In feite is het dus exact dezelfde vergelijking als vergelijking2. Vandaar dat de uitvoer van bovenstaande statements **false** is terwijl ze inhoudelijk wel gelijk zijn.

Om objecten te kunnen vergelijken moeten we dus zelf een methode equals toevoegen aan klasse Klant. Die methode doet in alle klassen hetzelfde soort werk. Er moeten namelijk altijd twee dingen gecontroleerd worden:

1. Is de inkomende parameter een object van dezelfde klasse als het object waarop de methode equals is aangeroepen?
2. Zo ja, hebben hun attributen exact dezelfde waarden?

Dat kan op de volgende manier:

```
public class Klant {
  private String naam;
  private String adres;
  private int leeftijd;

  public Klant(String nm, String adr, int lft) {
    naam = nm;
    adres = adr;
    leeftijd = lft;
  }

  public boolean equals(Object andereObject) {
    boolean gelijkeObjecten = false; // blijft false tenzij:

    if (andereObject instanceof Klant) {
      Klant andereKlant = (Klant) andereObject;

      if (this.naam.equals(andereKlant.naam) &&        // zie vooor 'this'
          this.adres.equals(andereKlant.adres) &&      // uitleg hieronder
          this.leeftijd == andereKlant.leeftijd) {

        gelijkeObjecten = true;
      }
    }

    return gelijkeObjecten;
  }

  public String toString() { return naam+ ", " +adres+ ", " +leeftijd; }
}
```

Listing 35: Klasse Klant uitgebreid met methode equals

De parameter is van het algemene type Object omdat je in Java **alle** objecten met elkaar kunt vergelijken. Daarom moet je eerst kijken of de parameter wel van het juiste type is. Dat doe je met het keyword **instanceof**. Als andereObject van het type Klant is, dan kan er veilig een cast plaatsvinden. Dat hoeft niet, maar maakt de rest van de methode korter van code.

ⓘ *De code '**this.**naam' wijst expliciet naar het naam-attribuut van **het object waarop de methode equals wordt aangeroepen** (en niet de naam van het object waarmee je een vergelijking wilt maken). De toevoeging **'this'** is hier gebruikt om de code zo duidelijk*

*mogelijk te laten zijn, maar is niet noodzakelijk. Soms zijn er echter parameters met dezelfde naam als een attribuut. Gebruik je die naam in je code, dan zal Java eerst de parameter 'zien' (zie ook les 4 voor 'scopes'). Wil je toch het attribuut benaderen, dan kan met '**this.**attribuutnaam' expliciet het attribuut aangewezen worden, in plaats van de gelijknamige parameter.*

Als de objecten van hetzelfde type zijn, en ook de attributen zijn gelijk, dan hebben we het over **inhoudelijk** gelijke objecten. Die attributen worden één voor één gecontroleerd. String attributen kun je natuurlijk met de equals van klasse String vergelijken, terwijl primitieve typen zoals int en double met == op gelijkheid kunt testen. Na toevoegen van methode `equals()` aan klasse `Klant`, krijgt de vergelijking een ander resultaat:

```
Klant k1 = new Klant("Jan", "Nijenoord 1", 55);
Klant k2 = new Klant("Jan", "Nijenoord 1", 55);

boolean vergelijking1 = k1.equals(k2);
boolean vergelijking2 = k1 == k2;

System.out.println(vergelijking1);        // uitvoer: true
System.out.println(vergelijking2);        // uitvoer: false
```

De methode doet zijn werk dus goed. In dit voorbeeld roepen we de methode equals aan op **k1**. Dat betekent dus voor die methode-aanroep in methode equals **this.naam** verwijst naar de naam van object k1! Variabele **andereKlant.naam** verwijst dan naar de naam van object **k2**. Maar dat hoeft niet altijd het geval te zijn. Gegeven is deze code:

```
Klant k1 = new Klant("Henk", "Nijenoord 1", 41);
Klant k2 = new Klant("Jan", "Oudenoord 10", 55);
```

In de onderstaande tabel kun je goed zien wat dit voor gevolgen heeft in de methode equals tijdens uitvoeren van deze methode:

In methode equals:	**this.naam**	**this.leeftijd**	**andereKlant.naam**	**andereKlant.leeftijd**
k1.equals(k2)	Henk	41	Jan	55
k2.equals(k1)	Jan	55	Henk	41

Maar wat nu als we ook nog andere objecten met onze klant zouden vergelijken?

```
Klant  k1 = new Klant("Jan", "Nijenoord 1", 55);
String s1 = "een willekeurige tekst";

boolean vergelijking1 = k1.equals(s1);
System.out.println(vergelijking1);        // uitvoer: false

boolean vergelijking2 = s1.equals(k1);
System.out.println(vergelijking2);        // uitvoer: false
```

Deze **eerste** vergelijking is mogelijk omdat we in de equals-methode van Klant de parameter van het type Object hebben opgenomen! De **tweede** vergelijking is toegestaan omdat klasse String een versie van methode equals heeft die dat ook heeft gedaan.

5. ARRAYLIST EN EQUALS

Klasse ArrayList heeft een aantal methoden die nauw verbonden zijn aan de methode equals. We behandelen daarvan de methoden contains(Object), indexOf(Object) en remove(Object). Methode **contains** zoekt in een lijst of er een bepaald object in voorkomt en geeft dan **true** of **false** als resultaat. Methode **indexOf** doet hetzelfde maar geeft de index van dat object als resultaat of **-1** als het object niet gevonden kon worden. Methode **remove** verwijdert een object uit de lijst en geeft met **true/false** aan of het is gelukt. De werking van die methoden is volledig afhankelijk van het feit of voor een object wel of niet de methode equals(Object) is uitgeschreven.

De beste manier om dit duidelijk te maken is aan de hand van een voorbeeld. Stel dat we een lijst maken van Klant-objecten terwijl klasse Klant **geen** methode equals is opgenomen.

```
import java.util.ArrayList;
public class Voorbeeld2 {
  public static void main(String[] args) {
    ArrayList<Klant> klanten = new ArrayList<Klant>();

    Klant k1 = new Klant("Piet", "Oudenoord 330", 55);
    Klant k2 = new Klant("Piet", "Oudenoord 330", 55);

    klanten.add(k1);

    if (klanten.contains(k2)) {
      System.out.println("k2 komt al voor in de lijst!");
    } else {
      klanten.add(k2);
    }

    klanten.forEach(klant -> System.out.println(klant)); // lambda functie!

    Klant k3 = new Klant("Piet", "Oudenoord 330", 55);

    System.out.println("Index van k3: " + klanten.indexOf(k3));
    System.out.println("Verwijderen k3 gelukt: " +klanten.remove(k3));

    klanten.forEach(klant -> System.out.println(klant));
  }
}
```

Listing 36: De uitvoer van deze code hangt sterk af van de aan- of afwezigheid van methode equals in klasse Klant!

ⓘ *De code print op verschillende plaatsen informatie uit, waaronder twee keer de lijst met klanten. Dat is hier gedaan met een zogenaamde '[lambda-functie]'. Klasse ArrayList heeft een methode **forEach** waaraan je een 'losse' methode mee kunt geven die voor elk item in de lijst moet worden uitgevoerd. In dit geval is dat het printen van een klant. De werking van deze code is vergelijkbaar met een for-each lus maar dan korter opgeschreven!*

Achtereenvolgens wordt in Voorbeeld2;

- Een lijst aangemaakt om Klant-objecten in te kunnen opslaan.
- Klant k1 toegevoegd aan de lijst.
- Klant k2 toegevoegd aan de lijst **mits** deze **niet in de lijst voorkomt!**
- De lijst uitgeprint.
- Klant k3 gemaakt maar **niet aan de lijst toegevoegd.**
- De positie van Klant k3 in de lijst opgevraagd (die nog niet is toegevoegd).
- Een poging gedaan om Klant k3 uit de lijst te verwijderen.
- De lijst uitgeprint.

De **uitvoer** is **zonder methode equals in klasse Klant** als volgt:

```
Piet, Oudenoord 330, 55
Piet, Oudenoord 330, 55
Index van k3: -1
Verwijderen k3 gelukt: false
Piet, Oudenoord 330, 55
Piet, Oudenoord 330, 55
```

Regel 1 en 2 van de uitvoer tonen aan dat methode **contains** niet detecteert dat k1 en k2 **inhoudelijk gelijke objecten** zijn. Daardoor is het mogelijk om twee Klant-objecten met dezelfde gegevens aan de lijst toe te voegen.

Vanuit het perspectief van Java is dit logisch gedrag, want in het geheugen zijn 2 verschillende objecten aangemaakt. Vanuit het perspectief van een klant is dit echter vaak **niet wenselijk**. Bijvoorbeeld: als je als klant registreert bij een webshop, dan wil je niet dat iemand anders met exact dezelfde ingevoerde gegevens zich ook kan registreren! Dat zou om te beginnen al problemen geven bij het inloggen, want hoe moet het systeem bepalen welke klant er nu eigenlijk in probeert te loggen?

Regel 3 van de uitvoer bewijst dat k3 niet gevonden is in de lijst, want het resultaat is **-1**. Het is dan ook begrijpelijk dat het verwijderen **niet** lukt. De lijst blijft dus zoals deze was, wat op regel 5 en 6 van de uitvoer te zien is.

Hoe anders ziet de **uitvoer** er uit **met methode equals in klasse Klant**:

```
k2 komt al voor in de lijst!
Piet, Oudenoord 330, 55
Index van k3: 0
Verwijderen k3 gelukt: true
```

Methode contains detecteert nu ineens **wel** dat k2 al voorkomt in de lijst. Zelfs zonder dat deze is toegevoegd! Dat komt omdat de methode **contains** nu een **inhoudelijke vergelijking** maakt.

Dat werkt zo: methode contains vergelijkt alle items in de lijst één-voor-één met de gezochte parameter (**k2**). Dat is gedaan met de **equals** methode in klasse Klant. Zodra er een object in de lijst is die inhoudelijk overeenkomt met k2 geeft de methode **true** terug. In de lijst is **k1** al aanwezig, en deze komt inhoudelijk overeen met **k2**. Daarom wordt k2 **niet** toegevoegd. Dat zie je in regel 1 van de uitvoer. Op regel 2 wordt het enige item in de lijst uitgeprint; dat is dus k1.

Methode **indexOf** werkt vergelijkbaar. Omdat **k1** inhoudelijk ook gelijk is aan **k3**, levert deze methode de index 0 op want op deze positie staat **k1**. Verwijderen van **k3** zorgt ervoor dat het enige, **inhoudelijk gelijke, object** uit de lijst verdwijnt. Uitprinten van de lijst levert dus geen uitvoer meer op!

Samengevat; bij deze code:

```
ArrayList<Klant> klanten = new ArrayList<Klant>();
Klant k1 = new Klant("Piet", "Oudenoord 330", 55);
Klant k2 = new Klant("Piet", "Oudenoord 330", 55);
klanten.add(k1);

System.out.println(/*STATEMENT*/);
```

zou vervangen van `/*STATEMENT*/` door één van onderstaande regels als resultaat hebben:

Uitvoertabel	**Zonder equals** in Klant	**Met equals** in Klant
`klanten.contains(k1)`	true	true
`klanten.contains(k2)`	false	true
`klanten.indexOf(k1)`	0	0
`klanten.indexOf(k2)`	**-1**	0
`klanten.remove(k1)`	**true**	true
`klanten.remove(k2)`	**false**	true

➔ MAAK NU EERST OPDRACHT 6_2 UIT HET WERKBOEK!

LES 7 – OBJECT ORIENTATION: INTERFACES & INHERITANCE

1. INLEIDING

We hebben gezien hoe klassen onderling verbonden kunnen worden middels associaties. Tot nu toe waren dat alleen **has-relaties**. Maar dit is niet altijd voldoende. Er bestaan namelijk ook **is-relaties**. Dit is een relatie waarbij **klasse B een subtype is van klasse A**. Dit is misschien wat vaag, maar in de wereld om ons heen komt dat veelvuldig voor. Zo kun je zeggen dat een **cabriolet** een subtype van **auto** is, terwijl auto op zichzelf weer een subtype is van **voertuig**. Naast auto zijn ook fiets, tram en trein subtypen van voertuig.

We gaan kijken naar 2 belangrijke vormen van de **is-relation:**

- **interfaces**
- **(class) inheritance** (**overerving** in fatsoenlijk Nederlands)**.**

2. INTERFACES

Een **interface** is een manier waarop twee systemen kunnen **communiceren** zonder elkaar inhoudelijk te kennen. Dat kunnen software-systemen zijn, maar ook twee apparaten die je op elkaar aan kunt sluiten. Je kunt ook denken aan een Grafische User Interface (GUI). Via een scherm kan een mens dan communiceren met een applicatie.

Als we binnen de Java-wereld over interfaces spreken, dan hebben we het alleen over de manier waarop objecten met elkaar kunnen communiceren zonder elkaar inhoudelijk te kennen! We kijken hiernaar aan de hand van een eenvoudige casus:

> Jos werkt bij een jong softwarebedrijf. Het bedrijf heeft enkele vaste ontwikkelaars in dienst, maar de meeste werknemers zijn studenten die parttime klussen uitvoeren. Jos bouwt aan de GUI voor een systeem voor een autohandelaar die daarmee de auto-voorraad wil bijhouden.
>
> Hij wil dat een aantal parttimers de backend gaan ontwikkelen (waar b.v. de auto voorraad in een database wordt opgeslagen) en heeft al op een rij gezet wat dat systeem precies moet kunnen. Jos heeft echter geen tijd om te wachten totdat deze backend klaar is, en wil zelf verder werken aan de voorkant (de GUI).

De backend moet het mogelijk maken om een auto aan de voorraad toe te voegen en om een overzicht van alle auto's op te vragen. Deze functionaliteit heeft hij als eerste nodig in de GUI. Dat beschrijft Jos in een interface:

«interface»**AutoService**
+voegAutoToe(a : Auto) : void *+getAutos() : List<Auto>*

De code van een interface lijkt op een klasse waarvan de methoden niet zijn uitgeschreven:

```
import java.util.List;

public interface AutoService {
  public void voegAutoToe(Auto a);
  public List<Auto> getAutos();
}
```

Listing 37: Een interface heeft abstracte methoden; dus zonder body

Methoden die niet uitgeschreven zijn, noem je **abstracte** methoden. Je kunt deze methoden niet aanroepen want ze doen niets (er wordt geen code uitgevoerd). Methoden die wel uitgeschreven zijn heten **concrete** methoden en die kun je aanroepen en voeren dan een actie of berekening uit zoals we tot nu toe gewend zijn. Een interface kan **alleen** abstracte methoden bevatten (dat is niet helemaal correct maar daarover later meer). Die abstracte methoden moeten natuurlijk nog wel ergens geprogrammeerd worden. Je noemt dit **implementeren**. Implementeren van een interface doe je in een gewone klasse. In die klasse maak je dus dezelfde methoden aan als in de interface zijn gedefinieerd, maar nu wel uitgeschreven (uitvoerbare code) en deze methoden kun je dus wél aanroepen en uitvoeren.

Een interface is eigenlijk een soort **contract**, waarin je alleen de methoden beschrijft, maar ze niet codeert. Een klasse die dit contract 'ondertekent' belooft om alle methoden in het contract te implementeren (zich te houden aan alle eisen van het contract).

Het voordeel van het definiëren van de 'AutoService' interface in ons voorbeeld is dat Jos alvast aan de GUI kan gaan werken zonder dat er al een backend is. Hij heeft met de 'AutoService' interface een contract gemaakt waar een toekomstige backend klasse zich aan moet gaan houden. Door de interface te definiëren weet Jos nu alvast welke methoden hij kan aanroepen in zijn GUI code.

Jos gaat nu z'n GUI programmeren en kan daarbij b.v. de methode getAutos() gebruiken om auto's uit de backend op te vragen. Echter, hij wil z'n GUI code ook kunnen testen. Daarvoor moet hij wel een soort backend hebben die hem ook daadwerkelijk iets teruggeeft als hij de methode getAutos() aanroept. Omdat er niet direct collega's beschikbaar zijn, maakt Jos zelf even een tijdelijke implementatie van de interface AutoService.
Jos noemt deze klasse MockAutoService. 'Mock' wijst er op dat hij snel even methoden programmeert om de eigenschappen van de uiteindelijke service te **simuleren**. Deze methoden doen dus niet echt iets met de voorraad zoals je hieronder kunt zien:

```java
import java.util.*;

public class MockAutoService implements AutoService {
  public void voegAutoToe(Auto a) {
    System.out.println("methode voegAutoToe");
  }

  public List<Auto> getAutos() {
    ArrayList<Auto> autos = new ArrayList<Auto>();
    autos.add(new Auto("Mercedes", "SLK 230", 2008));
    autos.add(new Auto("Suzuki", "Swift", 2007));
    return autos;
  }
}
```

Listing 38: Implementatie van de AutoService-interface

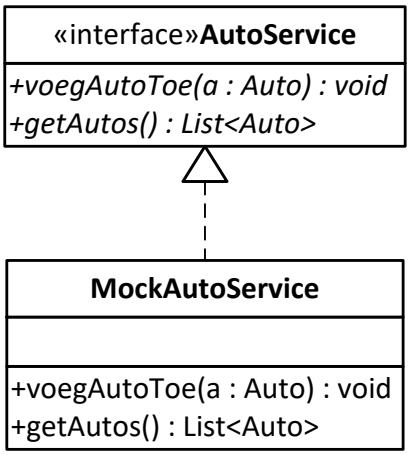
De methode voegAutoToe(Auto a) bijvoorbeeld verandert niets aan de voorraad, maar print alleen even dat de methode is aangeroepen.

De eerste regel van de klasse MockAutoService:

```java
public class MockAutoService implements AutoService {
```

geeft aan welke interface geïmplementeerd gaat worden (aan welk contract deze klasse zich gaat houden). De klasse moet op z'n minst **exact** dezelfde methoden hebben als de interface. Daarbij moeten per methode de methodenaam, het return-type, de parameters en de **access-modifier** (public) gelijk zijn. Als de klasse daar niet aan voldoet geeft de compiler een foutmelding dat de implementatie niet correct is (de klasse houdt zich niet aan het contract/ interface).

In UML ziet dat er als volgt uit:



De pijl met stippellijn en gesloten pijlpunt geeft aan dat de klasse MockAutoService de interface AutoService **implementeert**.

Laten we nu eens naar een stukje van de GUI code van Jos kijken om te zien hoe hij daar de AutoService interface gebruikt.

```
import java.util.List;
public class VoorraadGUI {
  private AutoService service;

  public VoorraadGUI(AutoService srv) {
    service = srv;
  }

  public void showVoorraad() {
    List<Auto> autos = service.getAutos();
    for (Auto a : autos) {
      /* Autos worden in het scherm getoond! */
    }
  }
}
```

Listing 39: Fictieve GUI ter illustratie van de AutoService

Zoals je ziet kan Jos zijn klasse VoorraadGUI al maken, terwijl de eigenlijke implementatie van de service, waarbij de auto's in een database, geheugen of andere datasource worden opgeslagen, nog niet is gemaakt! Jos heeft in zijn VoorraadGUI klasse een attribuut met de naam 'service' opgenomen. Dit attribuut is van het type AutoService (onze interface). En omdat we weten dat deze interface de methoden voegAutoToe() en getAutos() definieert kunnen we deze methoden aanroepen op dit 'service' attribuut. Dit zie je wat verderop in de code: service.getAutos();

Door de interface hebben we de backend (met evt. een database) losgekoppeld van de frontend (de GUI). Deze loskoppeling is een belangrijke reden waarom we een interface gebruiken.

Met een klasse Main start Jos de applicatie op en kan hij de GUI testen.

```
public class Main {
  public static void main(String[] args) {
    AutoService srv = new MockAutoService();
    VoorraadGUI gui = new VoorraadGUI(srv);
    gui.showVoorraad();
  }
}
```

Listing 40: Klasse Main maakt een AutoService-object, geeft deze door aan VoorraadGUI en start het scherm op

Hij gebruikt nu de snel in elkaar geprogrammeerde MockAutoService om de echte backend te simuleren. De MockAutoService implementeert de AutoService interface (zie Listing 38). De variabele 'srv' is een object van het type MockAutoService want we hebben de constructor van MockAutoService is gebruikt.

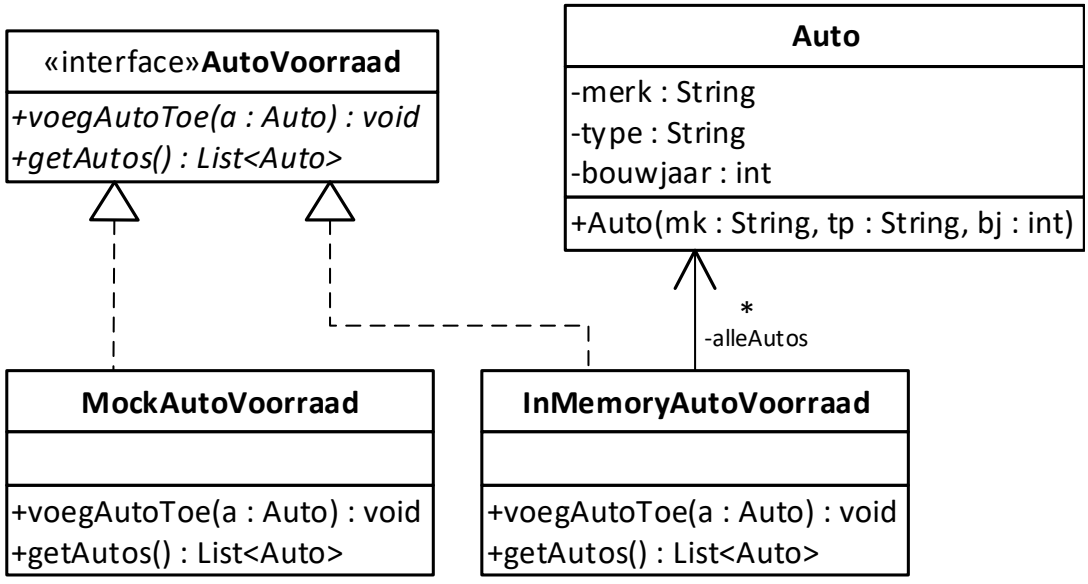
Toch gebruiken we hier in de code de naam van de interface (AutoService) om het type aan te duiden:

```
AutoService srv = new MockAutoService();
```

Waarom dit zomaar kan, daar gaan we in les 8 dieper op in. Voor nu is het van belang om te weten dat wanneer een klasse (bijvoorbeeld MockAutoService) een interface implementeert (AutoService), we ook kunnen zeggen dat de objecten van die klasse van het type AutoService zijn. Daarmee zeggen we eigenlijk dat die objecten voldoen aan het contract/interface AutoService. En daardoor kunnen we de methoden die in die interface zijn gedefinieerd (b.v. getAutos()) aanroepen op deze objecten. Mogelijk heeft zo'n object nog veel meer methoden dan die in de interface zijn gedefinieerd, maar daar zijn we nu niet in geïnteresseerd.

ⓘ *Merk op dat klasse MockAutoService ook op een andere manier gebruikmaakt van interfaces! Klasse ArrayList is namelijk een implementatie van de interface List. Daarom kun je in methode getAutos() als return-type* ***List<Auto>*** *opgeven terwijl je in de methode een ArrayList returnt.*

Terugkomend op de GUI van Jos: op een later moment kan een collega beginnen met de eigenlijke implementatie van de AutoService. Die implementatieklasse wordt hier **InMemoryAutoService** genoemd omdat de objecten voorlopig alleen nog in het geheugen worden opgeslagen. De klasse implementeert dezelfde interface als MockAutoService. Ze hebben dus minimaal dezelfde methoden, maar InMemoryAutoService houdt ook permanent een lijst met auto's bij, dus er bestaat een associatie met klasse Auto:



Hieronder staat de klasse InMemoryAutoService uitgeschreven. Om ervoor te zorgen dat niemand de oorspronkelijke lijst van auto's kan wijzigen geeft methode getAutos() een ***unmodifiable*** (onwijzigbare) lijst terug. Daarbij gebruikmakend van klasse Collections.

```java
import java.util.*;

public class InMemoryAutoService implements AutoService {
  private ArrayList<Auto> alleAutos;

  public InMemoryAutoService() {
    alleAutos = new ArrayList<>();
  }

  public void voegAutoToe(Auto a) {
    if (a == null)
      throw new IllegalArgumentException("a mag niet null zijn!");

    if (!alleAutos.contains(a)) {
      alleAutos.add(a);
    }
  }

  public List<Auto> getAutos() {
    return Collections.unmodifiableList(alleAutos);
  }
}
```

Listing 41: InMemoryAutoService implementeert dezelfde interface als MockAutoService

Toevoegen van een null-object levert een IllegalArgumentException op. Als de auto al voorkomt in de lijst wordt deze ook niet nogmaals toegevoegd (geen dubbelen). Omwille van de ruimte in deze reader is de klasse InMemoryAutoService beperkt van omvang gehouden. Voor het voorbeeld is het echter voldoende. Zodra de klasse InMemoryAutoService helemaal af is, hoeft Jos alleen zijn klasse Main te wijzigen:

```java
public class Main {
  public static void main(String[] args) {
    AutoService srv = new InMemoryAutoService(); // alleen deze regel verandert
    VoorraadGUI gui = new VoorraadGUI(srv);
    gui.showVoorraad();
  }
}
```

Listing 42: De wijzigingen om naar een andere AutoService-implementatie te gaan zijn minimaal

Omdat Jos in klasse VoorraadGUI alleen via de interface-methoden communiceert maakt het niet uit welk object daar eigenlijk achter zit (iedere klasse die de interface AutoService implementeert kan gebruikt worden). Klasse VoorraadGUI blijft dus volledig hetzelfde, terwijl 'onder de motorkap' een compleet andere klasse het werk doet!

Je ziet hier dus weer duidelijk dat de klassen zijn 'losgekoppeld' en dat de VoorraadGUI klasse alleen iets weet over de backend klassen door het contract/ de interface, maar niets over hoe die klassen de functionaliteit precies hebben geïmplementeerd. Dat is een belangrijk Object Orientatie principe.

ⓘ *Vanaf Java 8 is het toegestaan om in een interface ook default method-implementatie code op te nemen. Dit betekent dat we in de interface niet alleen maar de abstracte methode definiëren, maar ook meteen een (default) implementatie mee kunnen geven:*

```
public interface AutoService {
  public void voegAutoToe(Auto a);
  public List<Auto> getAutos();
  public default void verwijderAuto(Auto a){
    System.out.println("Auto verwijderd.");
  }
}
```

Listing 43: Default interface imlementation

De klasse die deze interface implementeert hoeft nu niet per se een implementatie voor de methode 'verwijderAuto()' op te nemen, want de interface heeft zelf al een default implementatie gegeven.
Dit is o.a. bedoeld om te voorkomen dat bestaande implementatie klassen van een interface niet meer werken als aan de interface een nieuwe methode wordt toegevoegd.

3. INHERITANCE (OVERERVING)

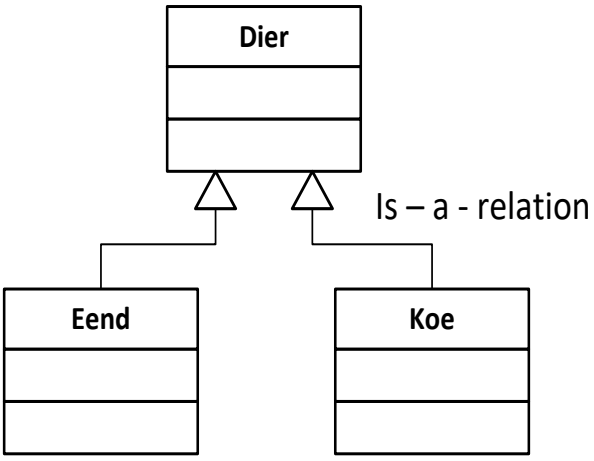
In de voorgaande paragraaf is al een **is-relation** aan de orde geweest. Zoals gezegd kun je in Java een klasse die een interface implementeert beschouwen als van het type van de interface zelf. Met andere woorden: klasse InMemoryAutoService **is** van het type AutoService. Of nauwkeuriger geformuleerd: Java beschouwt de klasse InMemoryAutoService als een **subtype** van AutoService.

De term **is-relation** en **subtype** kunnen we beter begrijpen bij een voorbeeld met 2 klassen (in plaats van met een interface):

> Jos heeft een nieuwe klus: hij moet een app ontwikkelen voor de lokale kinderboerderij. Een onderdeel van de app is dat kinderen een dier van de kinderboerderij kunnen opzoeken en er informatie over kunnen lezen.

Jos begint met het maken van een model. Naar verwachting zullen er veel dieren in de app opgenomen worden. Maar die dieren hebben ook gemeenschappelijke eigenschappen. Daarom ontwerpt hij een basisklasse voor de algemene eigenschappen. Dier-typen kunnen dan van deze basisklasse afgeleid worden.

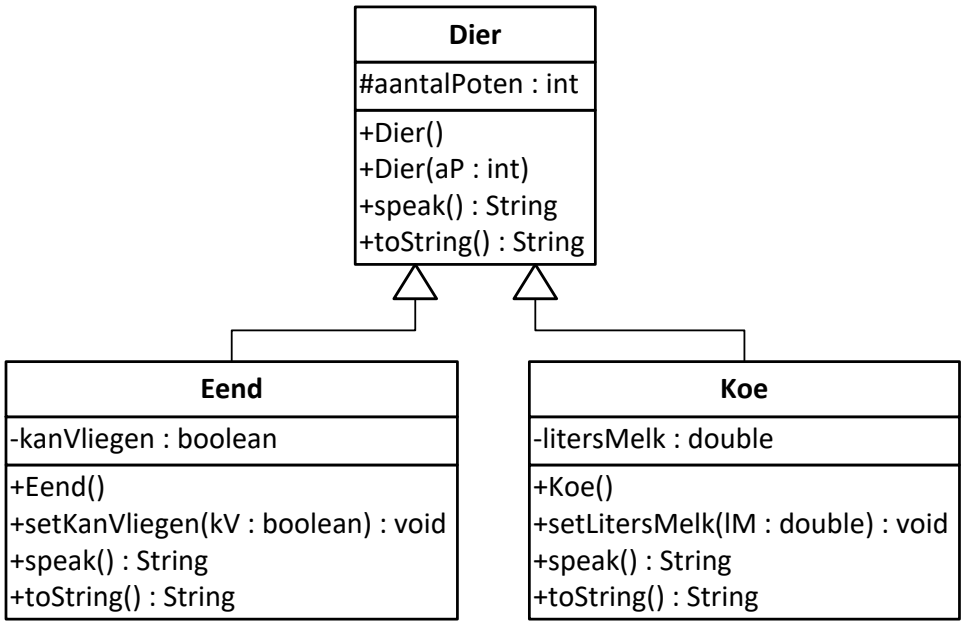
In het Dier-Eend-Koe voorbeeld hiernaast is klasse Dier de **superklasse** en zijn Koe en Eend **subklassen**. Een subklasse is een uitbreiding van de superklasse. Omgekeerd is een subklasse de superklasse-met-iets-extra.



Een subklasse **erft** alles van zijn **super**klasse, met uitzondering van de constructoren. Een subklasse **IS** dus een superklasse met extra attribu(u)t(en) of methode(n). De subklasse heeft dus een **is-relation** (of 'is-a' relation) met de superklasse. Als de subklasse geen extra's heeft, dan heeft deze ook geen bestaansrecht.

In UML modelleren we dat met een pijl met dorgetrokken lijn en gesloten pijlpunt.

Dit UML-diagram geeft te weinig informatie om te kunnen bepalen hoe klassen Eend en Koe verschillen van hun superklasse. We breiden het diagram daarom uit met attributen, constructoren en methoden.



Klasse Dier breidt geen klassen uit, het is de **superklasse**. Deze lijkt op eerder behandelde klassen, maar de eerste **constructor** en het **protected** attribuut vallen wel op:

```
public class Dier {
  protected int aantalPoten;

  public Dier() {
    this(0);
  }

  public Dier(int aP) {
    aantalPoten = aP;
  }

  public String speak() {
    return "*";
  }

  public String toString() {
    return "aantal poten: " + aantalPoten;
  }
}
```

Listing 44: Superklasse Dier

CONSTRUCTOR-CHAINING – BINNEN DE KLASSE

De **constructor**-zonder-parameters valt op door het statement:

```
this(0);
```

Wanneer je het keyword **this** gevolgd door haakjes tegenkomt, dan wordt daarmee een **andere constructor in dezelfde klasse aangeroepen** (en uitgevoerd). Welke dat is kan

gemakkelijk uitgemaakt worden, want er is maar één andere constructor die slechts één **int**-parameter heeft. In zo'n geval zeg je dat er sprake is van **constructor-chaining** (ketting van constructors). Als je dat doet moet het **altijd** het eerste statement in een constructor zijn! Je kunt ook een constructor aanroepen in een superklasse. We komen op deze laatste mogelijkheid binnen enkele ogenblikken terug. Maar waarom zou je constructor-chaining eigenlijk toepassen? Dit statement zou namelijk exact hetzelfde resultaat hebben gehad:

```
aantalPoten = 0;
```

Constructor-chaining is vooral handig als je controles of acties niet dubbel wilt programmeren. De huidige klasse Dier is te klein om dat te verduidelijken maar stel dat er ook nog deze constructors zouden zijn:

```
public Dier(int aantalPoten)
public Dier(int aantalPoten, boolean isZoogdier)
public Dier(int aantalPoten, boolean isZoogdier, int wekenDrachtig)
```

Je wilt niet dat het aantal poten een negatief getal is. Wanneer je geen constructor-chaining toepast moet je die controle in **elke** constructor programmeren. Met constructor-chaining hoef je maar in één constructor de parameters checken. De overige constructors laat je hun parameters doorsturen, gecombineerd met standaardwaarden ('defaults'):

```
public class Dier {
  protected int aantalPoten;
  protected boolean isZoogdier;
  protected int wekenDrachtig;

  public Dier(int aP) {
    this(aP, false, 0);
  }

  public Dier(int aP, boolean iZ) {
    this(aP, iZ, 0);
  }

  public Dier(int aP, boolean iZ, int wD) {
    if (aP < 0) {      // check aP parameter
      System.out.println("Een dier heeft 0 of meer poten.");
      aP = 0;
    }
    aantalPoten = aP;

    // check op deze plek evt. iZ parameter
    isZoogdier = iZ;

    // check op deze plek evt. wD parameter
    wekenDrachtig = wD;
  }
}
```

Listing 45: Constructor chaining

Zo voorkom je dat je dubbele code schrijft en daardoor is je programma beter onderhoudbaar!

ACCESS MODIFIERS

Voor de **access modifier** van attribuut aantalPoten in klasse Dier staat in het UML-diagram een hekje/hashtag (#), wat wil zeggen dat het attribuut niet private maar **protected** is. Dit zie je in de code terug. In Java zijn protected variabelen zichtbaar in de eigen package of in subklassen maar niet daarbuiten. Een overzicht van de **zichtbaarheid** van access modifiers:

Modifier	Class	Package	Subclass	World
public	Ja	Ja	Ja	Ja
protected	Ja	Ja	Ja	Nee
no modifier	Ja	Ja	Nee	Nee
private	Ja	Nee	Nee	Nee

SUBKLASSE EEND MAKEN

Klasse Dier heeft attribuut aantalPoten, omdat dit een eigenschap is die bijna alle dieren **gemeenschappelijk** hebben. Ook is er methode speak() die een tekstuele uitvoer geeft die je in het scherm van de app kunt lezen. Niet elk dier kan 'spreken' ofwel geluid produceren, dus de standaard-uitvoer maakt een * zichtbaar in de app.

De dierentuin heeft diverse eenden met geknipte vleugels. Sommigen kunnen dus wel vliegen, maar anderen niet. Dit kun je niet in klasse Dier vastleggen omdat dit niet voor alle dieren relevant is. Het is geen **gemeenschappelijke** eigenschap. Daarom breidt klasse Eend de klasse Dier uit en krijgt het een extra attribuut: **kanVliegen**. Een Eend-object **IS** dus ook een Dier-object. Niet omgekeerd! Een Eend-object is een Dier-object met een **extra** attribuut en extra methode **setKanVliegen(boolean kV)**:

```
public class Eend extends Dier {
  private boolean kanVliegen;

  public Eend() {
    super();
    aantalPoten = 2;
    kanVliegen = false;
  }

  public void setKanVliegen(boolean kV) {
    kanVliegen = kV;
  }

  public String speak() {
    return "kwaak kwaak kwaak";
  }

  public String toString() {
    String alDanNiet = (kanVliegen ? "wel" : "niet");
    return "Eend met " + super.toString() + " kan " +alDanNiet+ " vliegen";
  }
}
```

Listing 46: Klasse Eend is een uitbreiding ('extends') op Dier

Er vallen nogal wat dingen op in deze code. We behandelen ze per onderwerp:

INHERITANCE / OVERERVING

Met de eerste regel:

```
public class Eend extends Dier {
```

is met het keyword **extends** aangegeven dat de klasse een uitbreiding is op Dier. Uitbreiden van klassen is vergelijkbaar met implementeren van interfaces. In de klasse die een interface implementeert moesten we de abstracte methoden, die in de interface waren gedefinieerd, implementeren (van echte uitvoerbare code voorzien).
Bij overerving waarbij een subklasse (Eend) erft van een superklasse (Dier) hoeft dat niet per sé. De superklasse heeft alleen maar concrete methoden, die al uitvoerbaar zijn. De subklasse erft deze methoden en hoeft ze dus niet zelf te implementeren (het kan wel, maar daarover straks meer).

Er is nog een belangrijk verschil tussen interfaces en klasse overerving. **Een klasse** kan namelijk meerdere interfaces implementeren, maar **mag slechts één andere klasse extenden**! Dat heeft een logische reden, want als je bijvoorbeeld twee klassen tegelijk zou extenden die beiden dezelfde methode hebben (met een verschillende body), welke methode wil je dan **overerven**? Om deze problemen tegen te gaan staat Java slechts één uitbreiding (één extend van een superklasse) per klasse toe.

CONSTRUCTOR-CHAINING – NAAR DE SUPERKLASSE

Ook van klasse Eend valt de constructor op:

```
public Eend() {
  super();   // deze regel voegt de compiler automatisch toe als je het niet zelf doet!
  aantalPoten = 2;
  kanVliegen = false;
}
```

Je ziet hier het keyword **super** staan. Dit keyword wijst altijd naar de **superklasse**. Met super() roep je de constructor van de superklasse (Dier) aan en initieer/construeer je een Dier-object dat je vervolgens gaat uitbreiden tot een Eend-object! **Dat moet in een constructor altijd als eerste gebeuren, daarna kun je andere bewerkingen uitvoeren!** Vergelijk het met het feit dat een mens er pas kan zijn als er voorouders zijn. Wanneer je de opdracht **super()** niet opneemt zal de **compiler** deze **automatisch** als eerste opdracht in de constructor toevoegen!

ⓘ *Je kunt natuurlijk de vraag stellen wie dan de voorouder van het Dier-object is. Dat is simpel: als een klasse niet **expliciet** aangeeft welke klasse er **extended** moet worden, dan is dat **impliciet klasse Object!** Dit is de meest basale klasse waarop alle andere klassen zijn gebaseerd.*

Als de aanroep naar een constructor er **altijd** staat, dan moet er dus ook **altijd** een constructor in die superklasse zijn. Dat is een constructor zonder parameters. Als je niet zelf een constructor maakt in een (super)klasse, dan voegt de **compiler automatisch** een **default-constructor** toe! Deze constructor doet niets anders dan een nieuw object van de gewenste klasse aanmaken. Er zit geen code in deze default-constructor, dus verder doet hij niets. Het zorgt ervoor dat het programma niet crasht bij een de automatische super() aanroep. Let op: als je wel zelf een constructor maakt (met of zonder parameters) dan wordt de default-constructor **niet** toegevoegd door de compiler!

Het **tweede** opvallende is dat een attribuut van een **andere** klasse (**aantalPoten**) de waarde 2 krijgt! Dit mag, omdat dit attribuut **protected** is. Echter, we weten nu dat bij een object-creatie ten allen tijde de super-constructor aangeroepen zal worden. De constructor-**zonder**-parameters van Dier zet het aantalPoten dus eerst standaard op 0 (zie Listing 44). We kunnen de Eend-constructor nog verder wijzigen zodat de constructor-**met**-parameter wordt gebruikt:

```
public Eend() {
  super(2);
  kanVliegen = false;
}
```

Met het statement **super(2)** geef je aan dat je een constructor in de superklasse wilt aanroepen, en wel een die een int-parameter accepteert. Java zoekt automatisch voor je welke dat is. In klasse Dier is zo'n constructor aanwezig (zie weer Listing 44). Daarmee wordt het aantalPoten nu in die constructor op 2 gezet en hoeven we dat niet zelf meer te doen.

Dit mechanisme heet ook **constructor-chaining**, maar nu van subklasse naar superklasse.

OVERRIDING / OVERLOADING

Het volgende dat opvalt is dat we twee **speak()** methoden hebben. Eentje in de superklasse Dier, maar ook één in de subklasse Eend. De methode speak() in de Eend klasse geeft geen * als uitvoer, maar driemaal "kwaak". Hier wordt methode speak() uit superklasse Dier geherdefinieerd ofwel overschreven. Dat noemen we **overriding**. De subklasse Eend zorgt er met overriding voor dat de methode speak() specifiek geschikt is voor Eend-objecten. Welke van de 2 methoden speak() moet worden uitgevoerd wordt bepaald door het **object-type:**

```
Koe k = new Koe();
k.speak();          // object-type is Koe; methode speak() van Koe wordt uitgevoerd!

Dier d = new Dier();
d.speak();          // object-type is Dier; methode speak() van Dier wordt uitgevoerd!
```

Dit mechanisme heet **dynamische binding** en komt in de volgende les nog uitgebreider aan de orde.

Van **overriding** is sprake wanneer **methoden** in een **subklasse** dezelfde naam, parameterlijst en returntype hebben als methoden in de **superklasse**. De **access modifier** mag wel wijzigen, maar niet **nauwer** worden. Dus public mag niet protected worden, maar protected mag wel public worden. Er mag dus wel **meer** toegang ontstaan maar niet **minder** toegang!

Wanneer je een methode overschrijft wil dat niet zeggen dat de methode verdwijnt! De methode toString() van Dier is bijvoorbeeld ook geherdefinieerd (override) in klasse Eend. Maar klasse Eend maakt handig gebruik van de overschreven methode in de superklasse:

```
public String toString() {
  String alDanNiet = (kanVliegen ? "wel" : "niet");
  return "Eend met " +super.toString()+ " kan "+alDanNiet+" vliegen";
}
```

Het resultaat van de toString() methode van klasse Dier wordt nu verwerkt in het resultaat van methode toString() van klasse Eend! Met het gebruiken van het keyword **super** kun je dus de overschreven methode alsnog aanroepen!

Deze methode bevat verder op regel 1 een **ternary-operator**. Deze kan soms een if-statement vervangen. De operator begint met een booleaanse conditie; `true` of `false` (hier `kanVliegen`). Na het vraagteken staat de waarde ("wel") die bij `true` toegekend wordt aan variabele `alDanNiet`. Achter de dubbele punt volgt de waarde ("niet") die bij `false` wordt toegekend. Een alternatief voor deze operator zou bijvoorbeeld het if-statement kunnen zijn zoals hiernaast weergegeven:

```
String alDanNiet;
if (kanVliegen) {
  alDanNiet = "wel";
} else {
  alDanNiet = "niet";
}
```

Als methoden **dezelfde naam, maar een andere parameterlijst** hebben dan methoden in de eigen- of superklasse, dan is er sprake van **overloading**. Van verschil in parameterlijsten is sprake als de parametertypen verschillen ofwel het aantal parameters niet gelijk is:

```
public String speak() {              // eerste versie
  return "kwaak kwaak kwaak";
}

public String speak(int aantal) {    // overloading van eerste versie
  String geluid = "kwaak";

  for (int i = 1; i < aantal; i++) {
    geluid += " kwaak";
  }
  return geluid;
}
```

We hebben nu alleen nog klasse Koe niet besproken. Deze is vergelijkbaar met klasse Eend:

```
public class Koe extends Dier {
  private double litersMelk;

  public Koe() {
    super(4);
    litersMelk = 0.0;
  }

  public void setLitersMelk(double lM) { litersMelk = lM; }
  public String speak()                { return "boeh!";  }

  public String toString() {
    return "Koe met "+super.toString()+ " geeft "+litersMelk+" liter melk";
  }
}
```

Listing 47: Klasse Koe

Laten we eens kijken wat de code nu precies oplevert als we de verschillende methoden aanroepen in een Main-klasse:

```
public class Main {
  public static void main(String[] arg) {
    Dier d = new Dier();
    Koe k = new Koe();
    Eend e = new Eend();

    System.out.println("Een dier zegt: " + d.speak());
    System.out.println("Een koe zegt: " + k.speak());
    System.out.println("Een eend zegt: " + e.speak());
    System.out.println("\n" + d + "\n" + k + "\n" + e);
  }
}
```

Listing 48: Simpele main klasse

Merk op dat de uitvoer voor elk dier anders is:

```
Een dier zegt: *
Een koe zegt: boeh!
Een eend zegt: kwaak kwaak kwaak

aantal poten: 0
Koe met aantal poten: 4 geeft 0.0 liter melk
Eend met aantal poten: 2 kan niet vliegen
```

De superklasse Dier heeft de gemeenschappelijke methode speak() geïmplementeerd. Alle subklassen van Dier kunnen deze methode gebruiken. Echter, als een subklasse (Eend) een specifiekere invulling van die methode wil implementeren kan dat (“kwaak, kwaak, kwaak” ipv “*”). De subklasse ‘override’ nu de methode uit de superklasse. Objecten van het type Eend zullen nu, bij het aanroepen van de methode speak(), de override versie van speak() uitvoeren (“kwaak, kwaak, kwaak”).

➔ MAAK NU OPDRACHT 7_1 UIT HET WERKBOEK!

LES 8 – OBJECT ORIENTATION: POLYMORFISME, ABSTRACTE KLASSEN

1. INLEIDING

In het laatste voorbeeld van de vorige les werd, zonder dat we het wisten, eigenlijk al een bijzonder OO-concept toegepast: **polymorfisme**! Polymorfisme betekent 'veelvormigheid'. In de informatica wijst dat op het fenomeen dat objecten of methoden uiterlijk dezelfde kenmerken hebben maar toch anders zijn geïmplementeerd. Deze les passeert dit concept de revue, naast enkele OO-onderwerpen die we nog hebben laten liggen.

2. POLYMORFISME & DYNAMIC BINDING

Een voorbeeld van polymorfisme is de methode speak() in klasse Dier, Eend en Koe. De methode heeft overal dezelfde naam, parameterlijst en return-type. Toch gedraagt de methode zich in al deze klassen anders. Een tweede voorbeeld is het voorkomen van twee AutoService implementaties terwijl ze allebei voldoen aan de interface AutoService. Dezelfde 'service' komt meerdere keren voor in verschillende gedaanten.
In Java kan je hier gebruik van maken. Stel dat je voor de kinderboerderij-casus van de vorige paragraaf een overzicht zou moeten maken van de geluiden die de aanwezige dieren maken. Je kunt dan een niet-generieke ArrayList gebruiken (dus niet van een specifiek type), maar daar kunnen we objecten van ieder willekeurig type in stoppen. Behalve koeien en eenden etc. zouden we er ook objecten van type Gebouw, Voertuig of Cursus in kunnen stoppen.
Maar als we daar dan de methode speak() van aanroepen gaat het mis, want die methode heeft een Gebouw waarschijnlijk niet. Om onze code veilig te maken voor dit soort fouten (als de code complexer wordt kan dat zomaar per ongeluk gebeuren) moeten we dan de generieke ArrayList gebruiken, dus b.v. een ArrayList<Eend>. Je krijgt dan verschillende lijsten:

```
ArrayList<Eend> eenden = new ArrayList<Eend>();
ArrayList<Koe> koeien = new ArrayList<Koe>();

eenden.add(new Eend());
koeien.add(new Koe());
koeien.add(new Dier());       // niet toegestaan!
koeien.add(new Eend());       // niet toegestaan!
koeien.add(new Gebouw());     // niet toegestaan!

for (Eend eend : eenden) {
  System.out.println(eend.speak());
}

for (Koe koe : koeien) {
  System.out.println(koe.speak());
}
```

Voor elke diersoorten moet dan dus een aparte lijst gemaakt worden om veilig de methode speak() aan te kunnen roepen. Da's nogal onhandig als we 40 diersoorten hebben in de kinderboerderij… Maar gelukkig kun je in Java een lijst maken waarbij je opgeeft wat het **supertype** van elk object **moet** zijn:

```
ArrayList<Dier> alleBeesten = new ArrayList<Dier>();

alleBeesten.add(new Koe());
alleBeesten.add(new Eend());
alleBeesten.add(new Dier());

for (Dier dier : alleBeesten) {
  System.out.println(dier.speak());
}
```

Zoals je kunt zien kun je dus nu één lijst aanmaken waar je alle Dier-typen aan kunt toevoegen. Dus objecten van klasse Dier, maar ook objecten van alle (toekomstige) subklassen van Dier zoals Koe en Eend en wellicht later nog Hond, Konijn etc… Daar komt de **is-relation** weer om de hoek kijken. Een Koe **is** namelijk een Dier, en een Eend **is** ook een Dier.

De **for-each** lus die is gebruikt om alle dieren uit te printen, lijkt gebruik te maken van de methode speak() in klasse Dier. Maar dat hoeft niet per se het geval te zijn. De **compiler** kijkt weliswaar tijdens het compileren naar het **reference-type**. Dat is Dier:

```
for (Dier dier : alleBeesten) {
```

De compiler controleert daarna bij de volgende regel of klasse Dier de methode **speak()** heeft:

```
  System.out.println(dier.speak());
```

Dat is inderdaad het geval (zie klasse Dier in Listing 44). De compiler keurt de code dus goed! Bij het uitvoeren (tijdens **runtime**) kijkt de Java Virtuele Machine (JVM) echter naar welk object er eigenlijk aan de reference-variabele ('dier') is gekoppeld. En dat zijn achtereenvolgens de objecten die in de lijst zijn toegevoegd (Koe, Eend & Dier).

Dit principe heet **dynamic binding**: het object-type bepaalt welke versie van methode speak() uitgevoerd zal worden. Als het betreffende object in de lijst een Koe-object is, wordt de methode van klasse Koe uitgevoerd. Maar als het een Eend-object of Dier-object is moet methode speak() van respectievelijk klasse Eend of Dier uitgevoerd worden. Dit kunnen we duidelijker maken aan de hand van het volgende voorbeeld:

```
public class Main {
  public static void main(String[] args) {
    Dier d1 = new Koe();              // reference-type = Dier & object-type = Koe
    Dier d2 = new Eend();             // reference-type = Dier & object-type = Eend
    Dier d3 = new Dier();             // reference-type = Dier & object-type = Dier

    System.out.println(d1.speak());   // object-type = Koe, dus speak() uit klasse Koe
    System.out.println(d2.speak());   // object-type = Eend, dus speak() uit klasse Eend
    System.out.println(d3.speak());   // object-type = Dier, dus speak() uit klasse Dier
  }
}
```

Listing 49: Dynamic binding: object-type bepaalt welke methode uitgevoerd moet worden

We zien hier nog duidelijker dat we een object van het type Koe of Eend kunnen toekennen aan een variabele (d1) die we declareren als type Dier. Dat deden we hierboven in de ArrayList<Dier> eigenlijk ook al. Dat mag dus omdat een Koe een Dier **is** en een Eend ook een Dier **is.**
Een reference-variabele (d1 in ons geval) wijst naar een object van **hetzelfde type of een subtype ervan**! Tijdens runtime wordt gekeken welk object erbij hoort en welke object-type die heeft. Dat levert drie verschillende object-typen en dus drie keer een andere uitvoer op:

```
boeh!
kwaak kwaak kwaak
*
```

We hebben het al eerder over het **casten** van het ene type naar het andere gehad. Wat we hierboven gedaan hebben is eigenlijk een **impliciete cast:**

```
Koe k1 = new Koe();
Dier d1 = k1;              // hier casten we van type Koe naar type Dier
```

De compiler doet deze impliciete cast al voor ons. Hij ziet aan de overerving dat een Koe een Dier **is** en kan daarom zelf de conversie veilig doen.

Andersom (Dier naar Koe) kunnen we ook **casten**, maar nu moet het een **expliciete cast** zijn**:**

```
Dier d1 = new Koe();
Koe k1 = (Koe) d1;         // expliciete cast van type Dier naar type Koe
```

Op de eerste regel is de variabele d1 gedeclareerd als het type Dier. En die willen we op regel 2 toekennen aan de variabele k1 van het type Koe. Deze cast (van Dier naar Koe) kan de compiler niet zelf want een Dier is niet per sé een Koe, dus de compiler weet niet of deze cast wel correct is. Daarom moeten we dat **expliciet** maken in de code. Dit hebben we bij het implementeren van de 'equals' methode al toegepast.

Let op! Omdat de compiler niet kan checken of deze cast eigenlijk wel correct is kan het ook tot een fout leiden:

```
Dier d1 = new Dier();
Koe k1 = (Koe) d1;         // weer expliciete cast van type Dier naar type Koe
```

Bovenstaand voorbeeld kun je wel compileren, maar op het moment dat je de code runt gaat het mis. De JVM checkt dan van welk type het object d1 nu eigenlijk is. Dat blijkt een Dier te zijn, en een Dier **is** geen Koe, dus dit kan helemaal niet. De JVM zal een fout gooien.
Om dit soort fouten te voorkomen moet je, voor je zo'n expliciete cast doet, eerst checken of dat wel kan. Dat kan met de **instanceof** operator, zoals we ook bij het implementeren van 'equals' gedaan hebben (6.4: Methode Equals).

```
Dier d1 = new Dier();
if (d1 instanceof Koe) {
   Koe k1 = (Koe) d1;
}
```

Polymorfisme kun je ook toepassen bij interfaces. We hebben gezien dat het daar ook om **is-relations** gaat. Je kunt dat b.v. vergelijken met een student. Die heeft allerlei eigenschappen en 'studiemethoden' (b.v. een methode schrijfInVoorCursus()), maar de student is ook een gamer en in die rol heeft hij/zij andere mogelijkheden (methoden). In de rol van gamer hebben de studiemethoden geen belang. Je zou 'Gamer' als een interface kunnen zien die de student heeft geïmplementeerd en waardoor je de 'Gamer' methoden kunt aanroepen als er een game gespeeld gaat worden (b.v. koopNieuweSkin()). Nu zijn docenten echter ook gamer. Ook die hebben de 'Gamer interface' geïmplementeerd. Student en Docent zijn andere klassen, maar je kunt ze nu samen in een ArrayList<Gamer> stoppen want ze implementeren allebei de interface 'Gamer'. En daarmee kun je voor alle personen in deze ArrayList b.v. de methode koopNieuweSkin() aanroepen.

Wat niet meer kan, is op alle personen in deze ArrayList b.v. de methode schrijfInVoorCursus() aanroepen. De Java compiler kent deze personen nu als Gamer en kan alleen de methoden van de interface Gamer aanroepen (we kunnen wel proberen te casten, net als hierboven).

➔ MAAK NU EERST OPDRACHT 8_1 UIT HET WERKBOEK!

3. ABSTRACTE KLASSEN

Als het goed is heb je inmiddels gezien dat we ook een aantal keer een Dier-object hebben aangemaakt. Voor de voorbeelden kon dat prima, maar voor de kinderboerderij is dat niet wenselijk. Een Dier-object kan helemaal niet want elk beest is eigenlijk een subtype van dier. Ieder beest heeft namelijk iets unieks waarmee onderscheid ontstaat met andere dieren.

Door klasse Dier **abstract** te maken kunnen we ervoor zorgen dat je **geen objecten** van deze klasse meer kunt **creëren**! Een **abstracte klasse** kun je maken met het gelijknamige keyword:

```
public abstract class Dier {
```

Vervolgens is dit **niet** meer mogelijk:

```
Dier d3 = new Dier();
```

Een tweede voordeel van abstracte klassen is dat er ook **abstracte methoden** in mogen voorkomen. Met abstracte methoden hebben we bij de interfaces al kennisgemaakt; dit zijn methoden **zonder body**. In een interface zijn echter alle methoden abstract (zoals we hebben gezien is dat sinds Java 8 niet helemaal correct, zie Listing 43), maar in een abstracte klasse hoeft dat niet zo te zijn (mag wel). Daarom moet je dat **expliciet aangeven**:

```
public abstract String speak();
```

Alle abstracte methoden **moeten in een niet-abstracte subklasse worden uitgeschreven!** Je zult je misschien afvragen wat een abstracte methode dan voor nut heeft… Het is vooral handig als je wilt **afdwingen** dat een bepaalde methode in een subklasse aanwezig is. Door

bijvoorbeeld de methode speak() abstract te maken weet je zeker dat elke niet-abstracte subklasse deze methode heeft uitgeschreven om succesvol gecompileerd te kunnen worden.

En dat kan weer handig zijn als je bijvoorbeeld een lijst met dieren hebt zoals in de vorige paragraaf is besproken: ArrayList<Dier>. In deze lijst kunnen nu alleen maar objecten van Dier-subklassen voorkomen. Door methode speak() in Dier abstract te maken weet je dus zeker dat alle objecten in die lijst de methode speak() hebben.
Voordeel is nu ook dat we geen implementatie van die speak() methode in de Dier klasse hoeven op te nemen. Eerder moesten we daar als implementatie iets inzetten zoals *return "*"*, wat natuurlijk een beetje raar was (welk dier zegt er nou "*").

Een klasse kan **alleen** abstracte methoden hebben als de klasse zelf ook abstract is. Anderzijds kan een klasse dus wel abstract zijn zonder abstracte methoden te hebben! Een abstracte klasse mag verder gewoon attributen, constructors en methoden hebben. Als een **subklasse** van een abstracte klasse **niet alle abstracte methoden uitschrijft** moet de **subklasse ook abstract** zijn!

Een abstracte klasse met alleen abstracte methoden lijkt heel veel op een interface. Als je alleen abstracte methoden hebt kun je in sommige gevallen beter voor een interface kiezen omdat een klasse slechts één andere klasse mag extenden maar meerdere interfaces kan implementeren!

Let op: In een interface kun je geen attributen plaatsen, alleen algemene variabelen met een definitieve waarde (**static** & **final**). Op het Java keyword **static** gaan we later nog dieper in. **Final** betekent dat aan de variabele maar 1 keer een waarde kan worden toegekend. Dit kun je gebruiken voor variabelen die een constante voorstellen (b.v. final double pi = 3.14, ook al is dat natuurlijk al in de Math library opgenomen). Een abstracte klasse kan **wel** gebruikt worden om gezamenlijke attributen in te definiëren! Zie het voorbeeld van klasse Dier:

```
public abstract class Dier {
  protected int aantalPoten;

  public Dier() {
    aantalPoten = 0;
  }

  public Dier(int aP) {
    aantalPoten = aP;
  }

  public abstract String speak();

  public String toString() {
    return "Dier met " + aantalPoten + " poten";
  }
}
```

Listing 50: Klasse Dier, nu abstract zodat je er geen objecten van kunt maken

4. KLASSE OBJECT

Nu we super- en subklassen hebben besproken tenslotte nog een paar slotopmerkingen met betrekking tot klasse Object. In Java is klasse **Object** uiteindelijk de superklasse van alle klassen die er zijn, ook van de standaardklassen als String, Integer etc.

Hoe werkt het? Als je **geen** andere klasse extend, dan is klasse Object **impliciet** de superklasse van jouw klasse. Als je **wel** een andere klasse extend, dan is die andere klasse ofwel zelf of via zijn superklasse uiteindelijk een subklasse van Object! Het is in Java onmogelijk om geen subtype van Object te zijn.

Dat heeft als consequentie dat **elke klasse de methoden van klasse Object overerft**! Een aantal van die methoden ken je al: methode **equals(Object)** en **toString()**. Klasse Object heeft echter nog meer methoden zoals je in de documentatie kunt vinden. We hebben die methoden bij deze lessen verder niet nodig, maar een aantal ervan komen later in de studie aan de orde zoals notify, notifyAll en wait().

De reden dat je elk willekeurig object kunt uitprinten met System.out.println() is dus dat elk object een versie van methode toString() overerft die dan aangeroepen wordt:

```
Klant k1 = new Klant("Wim", "Nijenoord 1", "Utrecht");
System.out.println(k1);
```

Via dynamic binding kijkt de JVM eerst in Klant of daar een versie van toString() bestaat. Zo ja, dan zal die uitgevoerd worden. Zo niet, dan wordt de versie van klasse Object uitgevoerd! In deze standaardversie krijg je een niet-leesbare uitvoer (iets als 'Klant@6a5fc7f7'), maar de methode doet wel z'n werk.

Op dezelfde wijze functioneert ook methode **equals(Object)**. Ook al schrijf je die methode niet uit in je eigen klasse, dan nog kun je objecten met elkaar vergelijken. Bijvoorbeeld een string met een Klant-object:

```
Klant k1 = new Klant("Wim", "Nijenoord 1", "Utrecht");
k1.equals("dit mag natuurlijk nooit true opleveren!");
```

Via dynamic binding kijkt de JVM eerst in Klant of daar een versie van equals(Object) bestaat. Zo ja, dan zal deze versie uitgevoerd worden. Heb je zelf geen methode equals uitgeschreven? Dan voert de JVM de versie van klasse Object uit. We hebben al eerder gezien dat de standaardvergelijking de == (vergelijken van geheugenadressen) toepast.

➔ MAAK NU OPDRACHT 8_2 UIT HET WERKBOEK!

LES 9 – GRAPHICAL USER INTERFACES (JAVAFX)

1. INLEIDING

LES 10 – JAVAFX & STATICS

1. INLEIDING

LES 11 – INPUT / OUTPUT (IO)

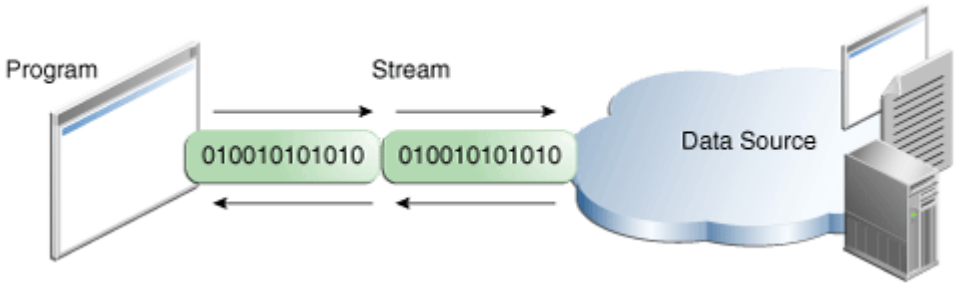
1. INLEIDING

In de programma's die je tijdens deze cursus hebt gemaakt heb je regelmatig de gebruiker van het programma op de hoogte gesteld van informatie die in je programma, en dus in het geheugen van je computer, aanwezig was. Dit deed je tot nog toe steeds met behulp van de System.out.println(..) opdracht. Dit is in feite een vorm van uitvoer, ook wel **output** genoemd.

Naast output is er natuurlijk ook sprake van **input**. Aan input en output wordt samen vaak gerefereerd als **I/O**. Input kan ook via verschillende wegen je programma binnenkomen: via het toetsenbord, van een bestand op de harde schijf of van een internetbron.

Java kent twee verschillende input/output Application Programming Interfaces (API's), de IO en NIO packages. NIO staat voor new I/O API's en is een uitbreiding op de oudere IO API. NIO leunt ook deels op de oudere klassen, dus je zult in de voorbeelden klassen uit beide API's voorbij zien komen. Welke klassen bij welke API horen, kan je herkennen aan de package: java.io of java.nio. Verder is dit onderscheid voor ons niet zo belangrijk.

Simpel gezegd is input of output een reeks van bytes die je programma **in**gaat,of **uit**gaat. De meest basale vorm van zo'n reeks bytes noemen we een stream. Het is mogelijk om vanuit een bron meerdere blokken bytes tegelijk te lezen, maar zeker voor een beperkte hoeveelheid bytes is dit niet altijd efficiënt. Onze voorbeelden zijn dus ook als volgt samen te vatten:



Figuur 9: I/O in Java via streams (afb. Oracle, bewerkt)

Output kan verschillende bestemmingen hebben. Output kan voorkomen door middel van een println-statement, maar je kunt data ook naar een bestand op de harde schijf sturen. Dat is ook output, maar hoeft niet zichtbaar te zijn in de console.

In deze les kijken we hoe Java omgaat met I/O en welke manieren er zijn om gegevens in te lezen en weg te schrijven. We zien daarbij streams voorbijkomen, maar ook readers en writers, scanners en files. Na deze les heb je als het goed is een redelijk overzicht wat je in welke situatie moet toepassen.

2. KLASSE FILES & PATH

Als je een bestand wilt lezen of schrijven, dan moet je weten waar dit bestand te vinden is, of moet komen te staan. De locatie geef je in een besturingssysteem aan middels een ***pathname***. Een voorbeeld van een pathname is **C:\Temp\info.txt**. Je mag in Java zowel slash als backslash gebruiken in een pathname, maar een backslash moet je wel dubbel opgeven, omdat de backslash '\' het escape-teken in strings is. Een pathname leg je vast in een **Path**-object:

```
Path pad = Path.of("C:\\Temp\\getallen.txt");
```

Hiervoor wordt de statische methode `'of'` van klasse Path gebruikt. Een Path-object kan ook gebruikt worden om directories aan te wijzen! Het maakt niet uit of dat een directory op Linux, Windows of Mac is. Een Path-object bevat echter alleen **informatie over** een bestand of directory. Het object kan dus niet gebruikt worden om een bestand te **schrijven** of **lezen**!

Om een bestand te lezen of te schrijven, gebruik je de klasse **java.nio.file.Files**. Met die klasse kun je bijvoorbeeld controleren of een bestand bestaat. Je kunt er ook (lege) **bestanden** en **directories** mee maken. Hieronder zijn enkele methodes van deze klasse gedemonstreerd:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class FilesDemo {
  public static void main(String[] arg) throws IOException {
    Path p = Path.of("getallen.txt");
    if (Files.exists(p) && Files.isRegularFile(p)) {
      System.out.println("Lengte file in bytes: " + Files.size(p));
      System.out.println(p.getFileName());
      System.out.println(p.toAbsolutePath());
    }
    Path p2 = Path.of("C:\\Program Files\\Java");
    if (Files.exists(p2) && Files.isDirectory(p2)) {
      System.out.println("Pad: " + p2.toAbsolutePath());
      System.out.println("Parent: " + p2.getParent());
    }
  }
}
```

Listing 51: Klasse Files gebruik je om het bestandssysteem te communiceren

De uitvoer van het programma is als volgt:

```
Lengte file in bytes: 12
getallen.txt
C:\1920_V1OOP-19\getallen.txt
Pad: C:\Program Files\Java
Parent: C:\Program Files
```

In het programma is sprake van verschillende soorten ***pathnames***. Een pathname is **absoluut** of **relatief**. Een **absoluut path** is bijvoorbeeld **c:\Program Files\Java**. Dit path is compleet, wat wil zeggen dat geen aanvullende informatie nodig is om de locatie te kunnen vinden!

Een **relatief path** daarentegen is **getallen.txt**. Je weet dan wat de naam van het bestand is, maar niet in welke directory het staat. Er is nog extra informatie nodig om het bestand te kunnen vinden. Standaard is dat de directory waar het programma werd gestart. In IntelliJ de hoofddirectory van je programma. Dus als de directory van je project **c:\school\v1oop\les11** is, dan breidt Java het relatieve path uit naar **c:\school\v1oop\les11\getallen.txt**.

In het voorbeeld zijn methoden exists(), isRegularFile(), isDirectory(), size(), getAbsolutePath() en getParent() gebruikt. Methode **size()** geeft de lengte van een bestand in aantal bytes. Methode **getParent()** gebruik je om de directory op te vragen waar het betreffende path zich in bevindt. Klasse **Files** heeft veel meer handige methoden. Zie daarvoor de documentatie!

ⓘ *Veel I/O methoden kunnen een **IOException** veroorzaken. Bijvoorbeeld omdat een ander programma het bestand al in gebruik heeft. In de voorbeelden is steeds aan methode main de "**throws IOException**" toegevoegd. Dit zorgt voor de kortste code, maar je programma stopt wel abrupt bij een exception. Zelf kan je natuurlijk ook een try-catch blok toepassen!*

3. FILES SCHRIJVEN EN LEZEN: TEKST

Nu we weten hoe je bestanden kunt benaderen, gaan we tekst naar een bestand schrijven. Streams gebruik je om '**ruwe**' bytes van of naar een stream te lezen (input) of te schrijven (output). Voor tekst gebruik je echter een **Reader** of **Writer**! Dat is omdat Java karakters (letters, cijfers, leestekens etc.) intern bijhoudt als Unicode. Anders dan ASCII (alleen Engels) is de opzet van Unicode om alle gebruikte schriften te kunnen representeren. Handig, maar het zorgt er ook voor dat een karakter meer bytes kan beslaan dan een ASCII-karakter.

Het besturingssysteem gebruikt vaak geen Unicode, en dus moet de codering bij het schrijven van tekst naar bytes geconverteerd worden. Readers en Writers doen dit automatisch (en ze gebruiken dan 'onder de motorkap' een stream om die bytes weer weg te schrijven). Ze vormen dus een tussenlaag tussen de onderliggende streams en jouw code.

TEKST NAAR FILE **SCHRIJVEN**: BUFFEREDWRITER

Klasse Files bevat diverse methoden om een verbinding te openen naar bestanden. Een daarvan is de statische methode:

```
Files.newBufferedWriter(Path, OpenOptions)      : BufferedWriter
```

Deze methode vereist een path-object (naar het bestand dat geschreven moet worden), en eventueel kan je 'open'-opties meegeven, bijvoorbeeld 'APPEND' om de nieuwe tekst aan het einde toe te voegen. Je krijgt van deze methode een BufferedWriter gereturned, waarin diverse handige methoden beschikbaar zijn om strings naar het bestand te schrijven:

```java
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;

public class Text_WritingFile {
  public static void main(String[] args) throws IOException {
    Path path = Path.of("demo.txt");
    BufferedWriter out = Files.newBufferedWriter(path);

    out.write("Demonstratie van schrijven naar file\n");
    out.write("8-4-2020");
    out.newLine();
    out.newLine();
    int i = 7;
    out.write("" + i + " x 9 = " + i * 9+ "\n");
    out.write(String.valueOf(12345.6789));
    out.close();
  }
}
```

Listing 52: Schrijven van pure tekst (geen objecten) naar een file

Deze code heeft geen uitvoer in de console, maar bestand **demo.txt** ziet er nu zo uit:

```
Demonstratie van schrijven naar file
8-4-2020

7 x 9 = 63
12345.6789
```

In de gegeven code is geen open-optie meegegeven bij het openen van de BufferedWriter. Dan is het zo dat altijd een nieuwe, lege, file wordt gemaakt. Wil je ander gedrag, bijvoorbeeld dat een nieuwe file wordt gemaakt als deze er nog niet is, maar dat tekst altijd aan het einde wordt toegevoegd bij een bestaande file? Dan kan je dat als volgt doen:

```java
Files.newBufferedWriter(path,
                        StandardOpenOption.CREATE,
                        StandardOpenOption.APPEND);
```

Verder is af te leiden dat methode **write(..)** niet eindigt met een regeleinde! Je kunt een '\n' in je string opnemen, of methode **newline()** gebruiken. Dat laatste heeft de voorkeur, omdat deze checkt wat op jouw OS een newline precies is. Dat is op Windows en Linux bijvoorbeeld verschillend ('\r\n' versus '\n'). Sommige teksteditors lossen dit zelf op, maar het gaat niet altijd goed. Methode newline() helpt dit te voorkomen!

Methode write(..) accepteert alleen strings of char(lijsten). Wil je een double of int naar het bestand schrijven, dan moet je deze dus eerst naar een string converteren. In het voorbeeld is zowel string-concatenatie gebruikt als de statische methode **String.valueOf(..)**. De laatste oogt netter als je een enkele double of int wilt wegschrijven.

TEKST UIT FILE **LEZEN**: KLEINE BESTANDEN

Als je een bestand hebt met relatief weinig tekst, dan kan je ervoor kiezen om de klasse Files alle tekst in één keer in te lezen. Daarvoor zijn er twee methoden:

```
Files.readString(path)          : String
Files.readAllLines(path)        : List<String>
```

De eerste methode leest het hele bestand als één string in, inclusief de regeleinden. De tweede methode leest de tekst uit het bestand, en splitst op de regeleinden, waardoor je een lijst met alle afzonderlijke regels terugkrijgt.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class Text_ReadingSmallFile {
  public static void main(String[] args) throws IOException {
    Path path = Path.of("demo.txt");

    List<String> alleRegels = Files.readAllLines(path);
    String alleTekst = Files.readString(path);

    for (String regel : alleRegels) {
      System.out.println(regel);
    }

    System.out.println(alleTekst);
  }
}
```

Listing 53: Lezen van kleine tekstbestanden

Deze manier van bestanden lezen is in veel gevallen prima. Maar let op, als je bestanden hebt met duizenden regels tekst, dan worden deze ook allemaal in het geheugen geladen. Dat kan de performance van je applicatie verlagen. Er is daarom ook nog een alternatieve wijze waarop je tekstbestanden kunt verwerken.

TEKST UIT FILE **LEZEN**: GROTE BESTANDEN (BUFFEREDREADER)

Op het moment dat je (zeer) grote bestanden wilt lezen, dan kan het handig zijn om een BufferedReader te gebruiken. Hiermee lees je een bestand regel voor regel, en hoef je niet alle tekst in het geheugen te laden.

Hiervoor kan je een **BufferedReader** gebruiken. Deze reader heeft methode readLine(). Deze methode begint aan het begin van het bestand, en geeft steeds de eerstvolgende regel terug. Als er geen regels meer zijn, is het resultaat van deze methode null. Zolang (while) dat niet het geval is, kan je dus steeds de volgende regel blijven lezen:

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class Text_ReadingBigFile {
  public static void main(String[] args) throws IOException {
    Path path = Path.of("grootbestand.txt");
    BufferedReader br = Files.newBufferedReader(path);
    String regel = br.readLine();

    while (regel != null) {
      System.out.println("Regelinhoud: " + regel);
      regel = br.readLine();
    }
    br.close();
  }
}
```

Listing 54: Inlezen van tekst kan met een BufferedReader en FileReader

➔ MAAK NU EERST OPDRACHT 11_1 UIT HET WERKBOEK!

4. FILES SCHRIJVEN EN LEZEN: BYTES

Stel dat je de data van je Java-programma wilt opslaan omdat het programma wordt afgesloten. Dan kan je al je objecten gaan vertalen naar tekst, maar het zou veel handiger zijn om die objecten rechtstreeks in een bestand te plaatsen en later weer in te kunnen lezen.

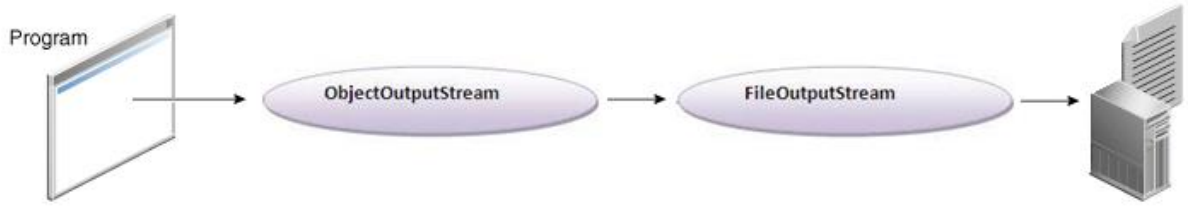
OBJECTEN NAAR FILE **SCHRIJVEN**: OBJECTOUTPUTSTREAM

Objecten bestaan in het geheugen uit enen en nullen (binaire data). Voor het opslaan van die data op de harde schijf gebruiken we daarom geen writers, maar output**streams**. Deze zijn speciaal geschikt voor 'ruwe' of binaire data. Klasse Files biedt een handige oplossing:

```
Files.newOutputStream(Path)      : OutputStream
```

Hiermee krijg je een OutputStream naar het opgegeven Path. Maar als je de documentatie van klasse OutputStreambekijkt, dan zie je dat deze geen methoden bevat om objecten weg te schrijven, alleen bytes.

Om objecten te schrijven gebruiken we een **ObjectOutputStream**, die we 'vastknopen' aan de OutputStream die we van de klasse Files krijgen. Daarmee krijg je de beschikking over de methode writeObject(..), maar ook methoden om ints, doubles, booleans etc. naar een bestand weg te schrijven. Wij zullen alleen methode writeObject(..) gebruiken. Je programma 'praat' dan tegen die ObjectOutputStream, die objecten vertaalt naar bytes, en op zijn beurt deze bytes doorgeeft aan de OutputStream:



Figuur 10: Het programma schrijft via de ObjectOutputStream, die een OutputStream gebruikt om het bestand te schrijven

Je kunt allerlei soorten objecten wegschrijven. We kunnen zelfs een lijst met objecten maken en vervolgens de hele lijst wegschrijven. Doordat de lijst referenties heeft naar alles wat erin zit (reference-koppeling) zullen alle lijst-items worden opgeslagen. En alles wat weer de lijst-items vastzit, zal ook worden opgeslagen. Het is dus een soort kettingreactie.

Maar er is wel een voorwaarde; alle objecten die naar het bestand geschreven moeten worden, moeten ook **Serializable** zijn! Dat wil zeggen dat de klassen waar de objecten van zijn, de ***interface*** **Serializable** moeten implementeren. Daarom moeten **alle** klassen waarvan je een object naar bestanden wilt schrijven -na de imports- beginnen met:

```
import java.io.Serializable;

public class Klant implements Serializable {
  /* overige code van deze klasse */
}
```

Listing 55: Alle klassen waarvan je objecten wilt opslaan MOETEN Serializable zijn!

Als je dit vergeet krijg je een **NotSerializableException**! De klasse ArrayList implementeert deze interface, en daarom kunnen we lijsten opslaan. We zullen wat objecten opslaan:

```
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;

public class Bytes_WritingObjectFile {
  public static void main(String[] arg) throws IOException {
    ArrayList<Klant> lijst = new ArrayList<>();

    lijst.add(new Klant("Kees de Bruin", "Lindelaan 12", "Abcoude"));
    lijst.add(new Klant("Lily van Iterson", "Weide 29", "Groningen"));
    lijst.get(0).setMijnRekening(new Rekening(103512547));

    OutputStream os = Files.newOutputStream(Path.of("klanten.obj"));
    ObjectOutputStream oos = new ObjectOutputStream(os);

    oos.writeObject(lijst);
    oos.close();
  }
}
```

Listing 56: Objecten wegschrijven met ObjectOutputStream en FileOutputStream

Deze code levert geen uitvoer in de console op, maar wel het bestand **klanten.obj** in de projectdirectory. Ook hier geldt (net als bij de writer) dat bestand **klanten.obj** wordt gemaakt als 'ie niet bestaat. Bestaande bestanden worden overschreven. Ook hier kan je via open-opties dit gedrag aanpassen aan jouw wensen.

Aan het einde van het programma wordt het object nog daadwerkelijk weggeschreven met methode writeObject(lijst). Dit is de laatste stap en hierna kan de stream gesloten worden. Het is voldoende om de ObjectOutputStream te sluiten, want deze sluit ook de onderliggende OutputStream. Als we het bestand **klanten.obj** bekijken zien we onleesbare binaire data.

OBJECTEN UIT FILE **LEZEN**: OBJECTINPUTSTREAM

Het zojuist aangemaakte bestand kan -op voorwaarde dat het bestand in tussentijd niet is gewijzigd- eenvoudig weer worden ingelezen. Dat doen we nu met dezelfde streamsoorten, maar nu de input-varianten: InputStream en ObjectInputStream:

```
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;

public class Bytes_ReadingObjectFile {
  public static void main(String[] arg)
        throws IOException, ClassNotFoundException {
    InputStream is = Files.newInputStream(Path.of("klanten.obj"));
    ObjectInputStream ois = new ObjectInputStream(is);

    Object obj = ois.readObject();
    ArrayList<Klant> lijst = (ArrayList<Klant>)obj;
    ois.close();

    for (Klant klant : lijst) {
      System.out.println(klant);
    }
  }
}
```

Listing 57: Bij het inlezen van objecten moet je ze nog wel casten!

Na aanmaken van de streams wordt het object ingelezen met **Object obj = ois.readObject()**. Omdat de compiler vooraf niet kan weten wat voor object er in het bestand zit, krijg je altijd een algemene Object-reference terug. Deze moet nog gecast worden naar het juiste type. Dat gebeurt met **ArrayList<Klant> lijst = (ArrayList<Klant>) obj;**.

Naast IOExceptions kan hier ook een **ClassNotFoundException** voorkomen. Dat kan gebeuren als iemand anders een object van een zelfgemaakte klasse naar een bestand heeft geschreven en jij dat object weer inleest, maar de bewuste klasse niet hebt.

Tot slot moet ook hier de stream gesloten worden. Als we dat niet doen heeft dat in dit programma niet zoveel effect omdat het eindigen van het programma er ook voor zorgt dat de streams gesloten worden. Maar als je een GUI-programma schrijft en je vergeet de stream te sluiten terwijl het programma gewoon blijft draaien, dan blokkeer je het bestand ook voor andere gebruikers en programma's!

5. TEKST NAAR SYSTEM.OUT

Streams kunnen ook voor andere doeleinden gebruikt worden dan het schrijven en lezen van bestanden. De meest bekende stream hebben bijvoorbeeld nog niet besproken: **System.out**, om tekst naar de console te schrijven.

Maar we hebben juist in de voorgaande paragrafen besproken dat als je tekst wilt schrijven, je dan een Writer moet gebruiken. Is dat dan niet in tegenspraak met het feit dat System.out een Print**Stream** is? Dat is inderdaad het geval want een string is gewoon tekst. Dit is echter een overblijfsel uit Java 1.0 waarin er geen onderscheid werd gemaakt tussen byte-streams en character-streams.

```
public class SysOutDemo {
  public static void main(String[] arg) {
    String s = "118 * 127 = ";
    System.out.print(s);
    int i = 118 * 127;
    System.out.println(i);
    System.out.printf("De wortel van %d = %.2f\n", i, Math.sqrt(i));
  }
}
```

Listing 58: System.out is de meestgebruikte PrintStream!

De standaard methoden om te printen zijn print(..) en println(..). De laatste print na iedere aanroep ook een newline en **flushed** (leegt) ook de buffer van de stream. Methode print(..) doet dat niet en het kan daarom gebeuren dat tekst niet direct op de console verschijnt.

```
118 * 127 = 14986
De wortel van 14986 = 122,42
```

Naast deze methoden is ook gebruikgemaakt van de methode printf(..). Deze is handig als je niet alle variabelen met plus-tekens aan de string wilt plakken. Ook kan je de uitvoer **formatteren** in een gewenst formaat. Alle format-aanduidingen beginnen altijd met een `%` en eindigen met conversie karakter. De hier gebruikte conversies zijn:

- `%d` verwerkt een int als decimaal getal.
- `%.2f` verwerkt een komma-getal als getal met max. 2 cijfers achter de komma.

Andere mogelijkheden zijn:

- `%x` verwerkt een int als hexadecimaal getal.
- `%s` verwerkt elke willekeurige waarde als string.

Alle formatteringsregels staan in de Java documentatie. Er zijn echter nog duidelijker overzichten te vinden zoals deze. Methode printf(..) biedt een zeer krachtige manier om op met weinig code veel te kunnen formatteren. Het kan echter ook op een alternatieve wijze:

6. SYSTEM.IN & KLASSE SCANNER

Naast System.out bestaat er ook een **System.in**. Deze InputStream wordt automatisch gekoppeld aan het toetsenbord, maar kent alleen methoden om bytes in te lezen. Om toch door de gebruiker ingevoerde tekst te kunnen inlezen in een programma, kan je een **Scanner** gebruiken. Een scanner kan uit een bron primitieve types en strings verwerken (parsen). De bron kan een gewone string zijn, maar ook een BufferedReader of InputStream zoals System.in.

We zullen dit demonstreren in het onderstaande programma. Allereerst moet de gebruiker een bestandsnaam opgeven via het toetsenbord. Een scanner leest de invoer. Daarna wordt het opgegeven bestand geopend en gelezen met behulp van, wederom, een scanner:

```
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Scanner;

public class ScannerDemo {
  public static void main(String[] args) throws IOException {
    System.out.print("Bronbestand: ");

    Scanner keyboardScanner = new Scanner(System.in);
    String bestandsnaam = keyboardScanner.nextLine();
    System.out.println("Bron: " + bestandsnaam);

    BufferedReader br = Files.newBufferedReader(Path.of(bestandsnaam));
    Scanner fileScanner =  new Scanner(br);

    while (fileScanner.hasNext()) {
      System.out.println(fileScanner.next());
    }
  }
}
```

Listing 59: Een Scanner kan gewone strings parsen

De uitvoer van het programma en de inhoud van het opgegeven bestand staan hieronder. Zolang (while) de scanner een volgend 'token' heeft – hasNext() – wordt dat token opgevraagd – next() – en geprint. Standaard ziet de scanner whitespace-karakters (spatie, enter, tab etc.) als scheidingsteken (**delimiter**), tenzij je anders hebt opgegeven.

```
Bronbestand:  cijfers.txt
Bron: cijfers.txt
Karel:
9,0
Hans:
3,1
Marietje:
7,3
```

```
Karel: 9,0
Hans: 3,1
Marietje: 7,3
```

Figuur 11: Links: uitvoer van Listing 67. Rechts: inhoud van cijfers.txt

De delimiter kan bestaan uit een eenvoudige string, bijvoorbeeld `";"`, maar ook uit een **reguliere expressie**. Stel dat we uit de file alleen de getallen willen 'filteren', dan kunnen we de volgende expressie gebruiken: `"\\s*\\w*:\\s*"`. De 's' staat daarbij voor whitespace-karakter, en de 'w' staat voor een letter of getal. De '*' geeft aan dat genoemde tekens 0 of meer keer voor kunnen komen. Als dit de delimiter is, dan valt 'Karel: ' daaronder, maar ook '\nHans: ' en '\nMarietje: '. Je houdt dan alleen de 3 cijfers over als je de beschikbare tokens op zou vragen. Aangezien we dan ook al direct weten dat dit kommagetallen zijn, kunnen we die ook direct naar doubles laten vertalen:

```
BufferedReader br = Files.newBufferedReader(Path.of("cijfers.txt"));
Scanner fileScanner =  new Scanner(br);
fileScanner.useDelimiter("\\s*\\w*:\\s*");

while (fileScanner.hasNextDouble()) {
    System.out.println(fileScanner.nextDouble());
}
```

Listing 60: Reguliere expressies zijn erg krachtig en te gebruiken als delimiter

Reguliere expressies vallen buiten de scope van deze cursus, maar er zijn oneindig veel verschillende reguliere expressies te maken. Het is een erg krachtig middel om kernachtig zoektermen samen te stellen. Je kunt er ook mee controleren of bijvoorbeeld een emailadres voldoet aan de regels. Alle instellingen zoals je ze in Java kunt gebruiken kun je vinden in de documentatie van klasse Pattern.

BIJLAGE 1: EXCEPTIONS DEFINIËREN

EXCEPTIONS DEFINIËREN

Als je in een try-blok meerdere methoden aanroept die fouten **kunnen** opleveren, dan is het niet handig als al die methoden allemaal dezelfde algemene exception gooien. Je kunt dan namelijk maar 1 catch-blok schrijven. Namelijk voor het algemene type Exception. Echter, niet alle fouten kunnen op dezelfde wijze worden afhandeld.

Door in je methoden specfiekere exceptions te gooien, kan je dit soort problemen voorkomen. Je kunt hiervoor standaard Java-exceptions gebruiken, of, als er geen geschikte is, een zelfgedefinieerde exception:

```
public void wijzigEmail(String em) throws EmailIncorrectException {
  if (!em.contains("@")) {
    throw new EmailIncorrectException("@ ontbreekt!");
  }

  email = em;
}
```

De `EmailIncorrectException` bestaat nog niet, dus die klasse moet je wel zelf toevoegen:

```
public class EmailIncorrectException extends Exception {
  public EmailIncorrectException() {}

  public EmailIncorrectException(String message) {
    super(message);
  }
}
```

Listing 61: De zelfgemaakte EmailIncorrectException

Het keyword **extends** in voorgaande listing geeft aan dat de EmailIncorrectException een subtype is van Exception. Pas in les 7 gaan we verder in op super- en subtypen. Verder zijn er twee constructors; 1 voor het maken van een EmailIncorrectException met bericht, en 1 zonder bericht. Vanzelfsprekend is het altijd het handigste om een exception **met** bericht te maken, zodat degene die de exception afhandelt, weet wat er aan de hand is.

ⓘ *Voor nu mag je deze code als gegeven beschouwen. Wil je een andere exception-naam, dan wijzig je in de gegeven klasse de naam EmailIncorrectException driemaal in de naam die je beter geschikt lijkt.*

Het catch-blok van de klasse Main kan nu ook deze specifiekere exception afhandelen:

```
public class Main {
  public static void main(String[] args) throws Exception {
    Student s1 = new Student("Kees", null);

    try {
      s1.wijzigEmail("keeshu.nl");
    } catch (EmailIncorrectException eie) {
      System.out.println("Wijzigen niet geslaagd: " + eie.getMessage());
    }
  }
}
```

Listing 62: Alleen de zelfgemaakte EmailIncorrectException wordt afgevangen in het catch-blok

De uitvoer die deze code oplevert:

```
Wijzigen niet geslaagd: @ ontbreekt!
```

ⓘ *Zou er aan het try-blok extra code worden toegevoegd die ook exceptions kan opleveren, dan kan daarvoor nu eenvoudig een extra catch-blok worden toegevoegd, zoals we al eerder zagen.*