

Werken met objecten en klassen in Java

Leo Rutten <leo.rutten@iwt.khlim.be>

Werken met objecten en klassen in Java

door Leo Rutten

Copyright © 2009-2010 \$Date: 2010-03-01 16:40:37 +0100 (Mon, 01 Mar 2010) \$ \$Revision: 2612 \$ Leo Rutten

Samenvatting

Deze cursus geeft de student 1ABA/Schakel een introductie van het programmeren in Java. Er wordt gestart met de concepten objecten en klassen. Vanaf de eerste les krijgt de student hiermee te maken. Anders is het niet mogelijk om object-geïntereerd programmeren te leren. Het ontwerppakket is BlueJ, dat uitstekend geschikt is voor de objecten eerst benadering. Deze tekst kan als een labotekst gebruikt worden. In elk hoofdstuk worden concepten uitgelegd die meteen in BlueJ getest kunnen worden. De studenten leren in dit vak eenvoudige klassen met eigen methodes ontwerpen. Voor de interactie tussen objecten/klassen wordt er gebruik gemaakt van generieke collecties.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. See www.gnu.org/copyleft/fdl.html [<http://www.gnu.org/copyleft/fdl.html>].

Inhoudsopgave

Inleiding	vii
1. Waarom leren programmeren in het vak informatica?	vii
2. Waarom object-georiënteerd?	vii
3. De programmeertaal Java	vii
4. Werken met BlueJ	viii
5. Visie op informatica in opleiding industriële ingenieur	viii
6. Praktische informatie	viii
1. Objecten en klassen	1
1.1. Werken met objecten	1
1.2. De broncode bekijken	4
1.3. Een methode bijvoegen	5
1.4. De toestand van een object bekijken	5
1.5. Een extra object aanmaken	6
1.6. Objecten versus klassen	6
2. Ingrediënten van klassen	7
2.1. Een zeer eenvoudige Breuk klasse	7
2.2. Een methode om het quotiënt uit te rekenen	7
2.3. Een verbeterde constructor	7
2.4. Een extra constructor	8
2.5. setter methodes	8
2.6. Een verbeterde quotientberekening	8
2.7. Een quotiëntberekening die niet door 0 deelt	9
2.8. Een breuk omkeren	9
2.9. De volledig afgewerkte klasse Breuk	10
3. Keuzen maken in methodes	12
3.1. Een eenvoudige klasse voor een bankrekening	12
3.2. Verbeteringen aanbrengen	13
3.2.1. Controle afhalingen bijvoegen	14
3.2.2. Alleen positieve limieten instellen	14
3.2.3. Alleen positieve bedragen storten	14
3.2.4. De limiet instellen volgens de gemiddelde stortingen	14
3.3. De volledig afgewerkte klasse VeiligeRekening	16
4. Herhalingsstatement	18
4.1. Een eenvoudige herhaling met het for-statement	18
4.2. Een eenvoudige herhaling met het while-statement	18
4.3. Een while met onbekend aantal iteraties	19
4.4. Bereken de som van getallen	19
4.5. Meerdere herhalingsstatements combineren	19
4.6. Toon alle delers van een geheel getal	20
4.7. Test priemgetallen	20
4.8. Toon alle priemgetallen	21
4.9. Alle herhalingen in één voorbeeld	21
5. Werken met lijsten	24
5.1. Een array maken en gebruiken	24
5.2. Een array als objectvariabele	24
6. Objecten laten samenwerken	27
A. Java syntax	28
A.1. Klassen en methodes	28
A.1.1. Klassedefinitie	28
A.1.2. Veld	28
A.1.3. Type	28
A.1.4. Constructor	29
A.1.5. Methodes	29
A.1.6. Parameters	29
A.1.7. Return statement	30

A.2. Controlestructuren en bewerkingen	30
A.2.1. Toekenningsstatement	30
A.2.2. Uitdrukkingen	30
A.2.3. Uitvoer	30
A.2.4. Conditioneel statement	31
A.2.5. Voorwaarde	31
A.2.6. Samengestelde voorwaarde	31
A.2.7. Blokstatement	31
Verklarende woordenlijst	32

Lijst van figuren

1.1. Eerste project in BlueJ	1
1.2. Het lokale menu van een klasse	2
1.3. Een object maken	2
1.4. Objecten bijhouden	3
1.5. Het lokale menu van een object	3
1.6. Het resultaat van een methode	3
1.7. De nieuwe methode starten	5
1.8. De toestand van een object	6

Lijst van voorbeelden

1.1. De temperatuur bijhouden	4
-------------------------------------	---

Inleiding

1. Waarom leren programmeren in het vak informatica?

Informatica als vak is zonder twijfel niet meer weg te denken in een opleiding in het hoger onderwijs. Computers zijn werktuigen voor dagelijks gebruik geworden en worden beschouwd als een onmisbaar stuk meubilair in het kantoor. We veronderstellen dat iedereen ermee overweg kan en in staat is om elementaire taken zoals een tekst schrijven op computer kan uitvoeren. In een hogere technische opleiding willen we een stapje verder gaan: de student moet ook inzicht hebben hoe software op een computer werkt. Om dit te begrijpen, is er maar één methode: leren programmeren.

De belangrijkste reden waarom in het vak informatica een programmeertaal aangeleerd wordt, is het verwerven van inzichten over de werking (en ook de complexiteit) van software. Een student zal snel inzien dat het niet evident is om snel een correct werkend programma te ontwerpen. Dit inzicht moet ertoe leiden dat de student bij het ontwerpen van een programma uiterst rigoureuus en gedisciplineerd te werk gaat. De minste fout in een programma kan de werking ervan ruïneren.

2. Waarom object-georiënteerd?

Het eerste concept dat meteen in deze cursustekst naar boven komt wordt beschreven met de term object-georiënteerd. Dit is zo belangrijk dat het voortdurend aanwezig blijft gedurende de hele cursus en ook in vakken in de hogere jaren. Met de term object-georiënteerd wordt bedoeld dat er gewerkt wordt met objecten die zowel informatie bevatten als acties kunnen uitvoeren. Dit concept komt overeen met hetgeen iedereen die met computers werkt automatisch aanvoelt: computers slaan informatie op en kunnen taken uitvoeren. Deze twee aspecten zijn verenigd in het concept object. Een object dat zich in het geheugen van de computer bevindt, houdt informatie bij en is in staat om acties of taken met die informatie uit te voeren. Als we een object op deze manier bekijken gedraagt het zich als een kleine elementaire computer: informatie en taken zijn er gebundeld.

Het nieuwe aan object-georiënteerd¹ is dat je vanaf de start bij het ontwerp van een programma zowel moet kijken naar de informatie en welke acties die moeten ondergaan. Dit is meteen slecht nieuws: vroeger moest je alleen maar de taken in kaart brengen, nu ook de informatie die erbij hoort en erger, je moet nu bepalen welke acties bij welke informatie horen. Dit betekent dat het ontwerpen van een object-georiënteerd programma moeizamer verloopt dan het ontwerpen van een niet-object-georiënteerd programma. Van mensen die de omschakeling niet naar wel object-georiënteerd programmeren hebben meegemaakt zul je horen dat dit een hoge drempel is die niet gemakkelijk overwonnen kan worden. De overgang van niet naar wel object-georiënteerd kan je daarom beter vermijden. Vandaar objecten en klassen eerst in dit vak.

3. De programmeertaal Java

Een aantal jaren geleden kwam Java als beste uit de bus voor de programmeertaal in het eerste jaar ABA. Hiervoor zijn een aantal redenen. Ze worden even opgesomd:

- Java ondersteunt een zuiver vorm van object-georiënteerd programmeren. Ervan afwijken is lastig. De programmeur moet de lijn van object-georiënteerd programmeren blijven volgen.
- Java is één van de populairste programmeertalen. In de industrie is er veel vraag naar Java kennis. Dit argument is meegenomen.
- Java is didactisch goed. De syntax is niet zo uitgebreid dan bijvoorbeeld C++ en C.
- Java is geliefd als taal voor onderzoeksprojecten in de academische wereld.
- Met Java kan je alle aspecten van de informatica onderwijzen. Met andere woorden, het is een echte all-round taal.

¹Zo nieuw is de object-georiënteerde aanpak niet: ze bestaat al meer dan 20 jaar.

- Er bestaat veel literatuur over Java. Ook het Internet is een goede bron voor Java gerelateerde informatie.

Met al deze argumenten zijn we ooit met Java gestart; we zijn in de opleiding industriële ingenieur nog steeds tevreden met deze keuze.

4. Werken met BlueJ

De tool die het labo gebruikt wordt is BlueJ. Met deze tool kan je meteen starten met de concepten objecten en klassen. Voor de student is BlueJ voordelig: onnodige details komen in BlueJ niet ter sprake. Deze tool werd speciaal voor het aanleren van Java aan beginners ontworpen. BlueJ is zelf in Java geschreven, draait op alle platformen en is gratis.

5. Visie op informatica in opleiding industriële ingenieur

Leren programmeren in deze opleiding is onvermijdelijk. In een academische technische opleiding mag je van elke student verwachten dat hij/zij weet hoe een programma ontworpen wordt en hoe dit op een computer loopt. Iedereen in deze opleiding leert de basis van de programmeertaal Java. In het gemeenschappelijke deel van de opleiding komen we tot een afgerond geheel. Het einddoel is het ontwerpen van GUI programma's in Java.

6. Praktische informatie

Voor dit vak zijn er alleen maar labo's. Er worden telkens zoveel concepten uitgelegd als nodig en dan starten de oefeningen. Dikwijls starten de oefeningen met het analyseren van voorbeelden uit de cursus en daarna het aanpassen van de voorbeelden. Er wordt gebruik gemaakt van de BlueJ ontwerpomgeving. Hierin worden de voorbeeldprojecten geopend en kan de student ermee werken.

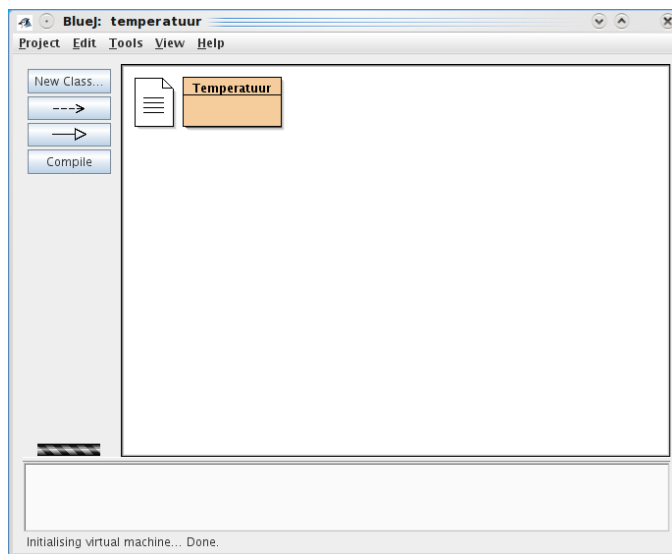
Hoofdstuk 1. Objecten en klassen

In dit eerste hoofdstuk starten we met het eerste project in BlueJ. Het is de bedoeling om na te gaan wat objecten en klassen precies inhouden. Deze eerste kennismaking vraagt helemaal geen programmeerervaring. Je kan in het eerste project meteen objecten aanmaken en ermee experimenteren.

1.1. Werken met objecten

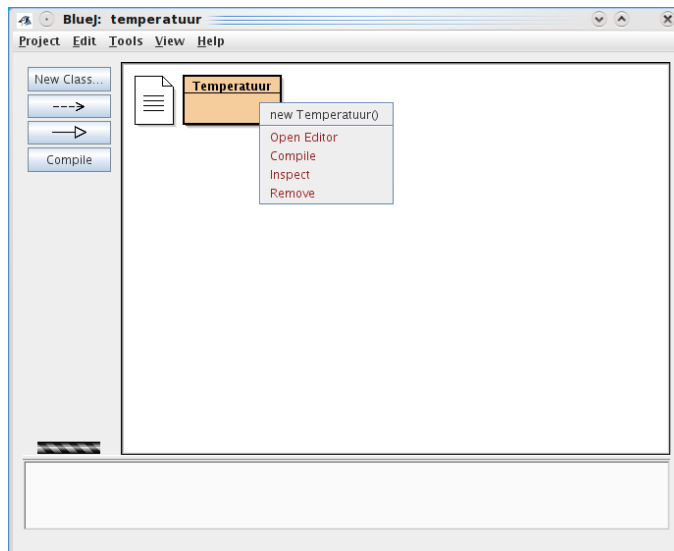
Start BlueJ en open het project `temperatuur`. Je ziet een grafische weergave van al de klassen in dit project. Zoals je ziet, is er maar één klasse, namelijk de klasse `Temperatuur`.

Figuur 1.1. Eerste project in BlueJ

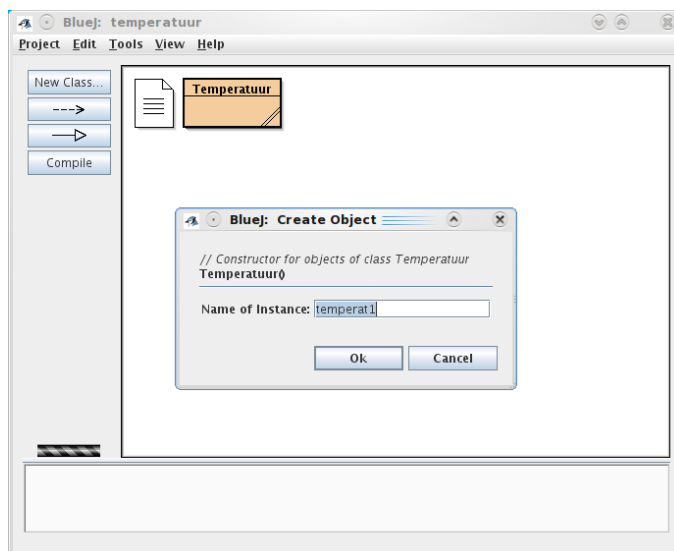


De klasse `Temperatuur` wordt voorgesteld door een rechthoek met daarin de naam van de klasse. Bij elk project in BlueJ worden alle klassen met een rechthoek in het hoofdvenster voorgesteld. Hierdoor heeft de student een goed overzicht over al de klassen die in het project bestaan. Grote programma's bestaan meestal uit een hele reeks klassen die allemaal met een eigen rechthoek zichtbaar zijn. Zo heb je altijd een goed overzicht. Voorlopig leggen we nog niet uit wat een klasse precies inhoudt. Dit wordt later duidelijk.

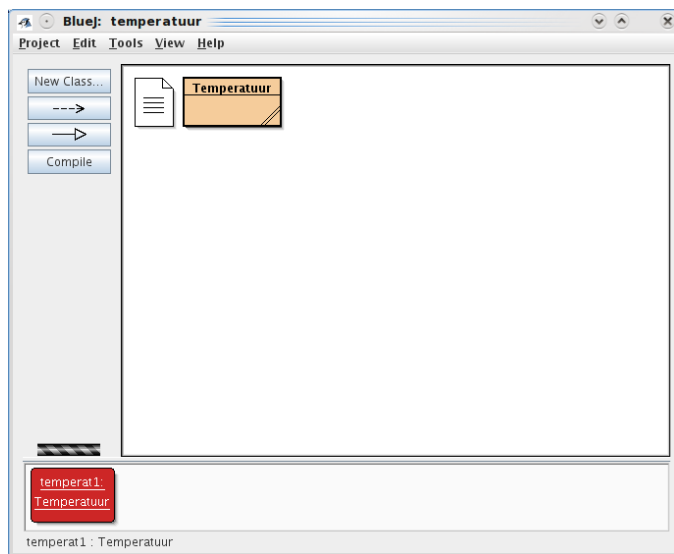
Wanneer je met de rechtermuisknop op de klasse `Temperatuur` klikt, verschijnt er een lokaal menu (Figuur 1.2, "Het lokale menu van een klasse"). Klik nu op `new Temperatuur()`. Hiermee maak je een nieuw `Temperatuur` object.

Figuur 1.2. Het lokale menu van een klasse

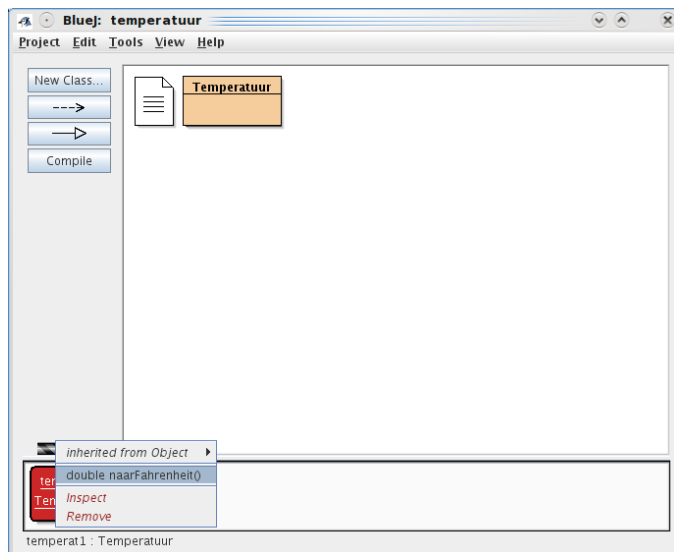
Elk nieuw object krijgt een naam. Met het dialoogvenster in Figuur 1.3, “Een object maken” wordt een nieuwe naam voorgesteld. Klik hier op ok. De voorgestelde naam `temperat1` is goed.

Figuur 1.3. Een object maken

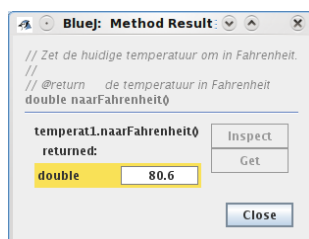
Dit object, dat we pas gemaakt hebben, verschijnt onderaan in het horizontale deelvenster zoals je kan zien in Figuur 1.4, “Objecten bijhouden”. Hier wordt weergegeven dat het object met naam `temperat1` van de klasse `Temperatuur` is. Voor de dubbele punt staat de naam van het object, erachter staat de naam van de klasse waartoe het object behoort. Je kan nu al aanvoelen wat het concept klasse betekent: elk object behoort tot een klasse. Het gebruik van klassen is een middel dat ons toelaat om objecten in categorieën onder te brengen. Het concept *object* is nog steeds niet duidelijk zijn. Naarmate we verder werken in BlueJ zal dit wel duidelijk worden.

Figuur 1.4. Objecten bijhouden

Het eerste object is gemaakt. Ook een object heeft een lokaal menu. Hierin zie je de methode `naarFahrenheit()` (Figuur 1.5, “Het lokale menu van een object”). Met deze methode kan je de in het object opgeslagen temperatuur omzetten van Celsius naar Fahrenheit.

Figuur 1.5. Het lokale menu van een object

Start de methode `naarFahrenheit()` door op de naam van de methode in het lokale menu te klikken. Het resultaat van deze actie wordt dan weergegeven in een dialoogvenster.

Figuur 1.6. Het resultaat van een methode

De in het object opgeslagen temperatuur wordt omgezet naar Fahrenheit. Dit levert de waarde 80.6 op (Figuur 1.6, “Het resultaat van een methode”). Dit is de waarde van de temperatuur in Fahrenheit uitgedrukt. Momenteel werkt dit object alleen maar met een vaste temperatuur. Later maken we dit aanpasbaar.

1.2. De broncode bekijken

We bekijken nu de Java broncode in Voorbeeld 1.1, “De temperatuur bijhouden”. Dit is de volledige broncode van de klasse `Temperatuur`.

Voorbeeld 1.1. De temperatuur bijhouden

```
/**
 * De klasse Temperatuur houdt een temperatuurwaarde bij
 *
 * @author Leo Rutten
 */
public class Temperatuur
{
    /**
     * temperatuur in Celsius
     */
    private double temp;

    /**
     * Constructor voor objecten van de klasse Temperatuur
     */
    public Temperatuur()
    {
        temp = 27.0;
    }

    /**
     * Zet de huidige temperatuur om in Fahrenheit.
     *
     * @return de temperatuur in Fahrenheit
     */
    public double naarFahrenheit()
    {
        return 9.0/5.0 * temp + 32.0;
    }
}
```

Java broncode bestanden zijn niet meer dan tekstbestanden. Het bovenstaande voorbeeld beschrijft de werking van de klasse `Temperatuur`. De opbouw van een klasse ziet er zo uit:

```
public class Temperatuur
{
    // inwendige van de klasse
}
```

Dit is het omhulsel van de klasse. Hierbinnen wordt vastgelegd wat de klasse `Temperatuur` allemaal inhoudt. Je start het omhulsel met het woord `public`, hiermee geef je aan dat iedereen deze klasse kan gebruiken. Dan volgt het woord `class` gevolgd door de naam van de klasse. Hier is dat `Temperatuur`. Je mag de naam van een klasse zelf kiezen. Het is gebruikelijk om de naam van een klasse te laten starten met een hoofdletter. Het accoladenpaar `{ }` beschrijft het blok vast waarbinnen allerlei definities kunnen geplaatst worden.

Het is ook gebruikelijk om commentaar te schrijven in de broncode. Hiermee kan je de klasse en de constructies binnen de klasse met een eigen tekst verduidelijken. Je kan deze commentaar in het Nederlands schrijven; de computer kijkt er immers niet naar. Er zijn twee manieren om commentaar te schrijven: voor één commentaarregel gebruik je `//` en voor meerdere regels gebruik je `/* */`. Hier is een voorbeeld:

```
// één regel commentaar

/*
   Meerdere regels
   commentaar
*/
```

Binnen de klasse wordt vastgelegd dat er in elk object een temperatuur opgeslagen kan worden. Dit doe je door een veld¹ vast te leggen.

¹Een andere term hiervoor is data member.

```
private double temp;
```

Met dit veld wordt vastgelegd dat in `temp` de waarde van de temperatuur als een komma getal kan opslaan. Met het woord `private` wordt aangegeven dat er beperkingen zijn voor de toegang tot het veld `temp`. Later wordt het gebruik van dit woord verder verduidelijkt.

De klasse `Temperatuur` bevat twee methoden. De eerste is een speciale vorm van een methode, het is namelijk een constructor. Elke methode heeft een naam. Bij de constructor is die naam dezelfde als de klassenaam.

```
public Temperatuur()
{
    temp = 27.0;
}
```

De constructor zorgt voor de startwaarde van het veld `temp`. Dit wordt met een toekenning gedaan. Dit herken je aan het `=` teken. In deze constructor krijgt het veld `temp` de waarde `27.0` als startwaarde.

De tweede methode is een gewone methode en geen constructor. Voor deze soort moet je een naam kiezen, hier is het `naarFahrenheit`. Zoals de naam zegt, zet deze methode de temperatuur in het veld `temp` om van graden Celsius naar graden Fahrenheit. Met het `return` wordt deze waarde teruggegeven aan de oproeper. Hoe dit in mekaar zit wordt in een later hoofdstuk uitgelegd.

Voor de namen van methoden is het gebruikelijk om een naam te kiezen die start met een kleine letter. In de naam van de methode komt bijna altijd een werkwoord voor. Hiermee wordt duidelijk gemaakt dat een methode een bepaalde actie uitvoert.

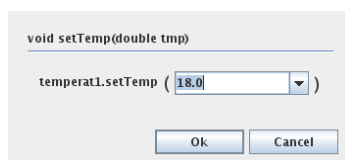
1.3. Een methode bijvoegen

Het zou handig zijn om een extra methode bij te voegen waarmee je een andere waarde in het object kan opslaan. Voeg daarom deze methode toe aan de klasse `Temperatuur`:

```
public void setTemp(double tmp)
{
    temp = tmp;
}
```

Met een dubbele klik om de klasse `Temperatuur` open je het editvenster. Hier kan je de broncode van de nieuwe methode bijtypen. Met de knop `compile` kan je de klasse opnieuw compileren. Maak dan een nieuw object aan (de objecten die je voorheen had aangemaakt, zijn nu verdwenen) en start de methode `setTemp`. Met een invoerveld krijg je de kans om een nieuwe waarde te kiezen.

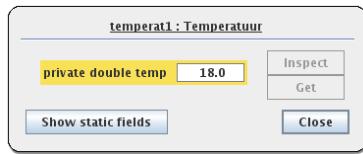
Figuur 1.7. De nieuwe methode starten



Als je daarna nog eens de methode `naarFahrenheit` start, dan krijg je ook hier een andere waarde.

1.4. De toestand van een object bekijken

Wanneer je de waarde van de velden van een klasse wil nakijken, klik dan met de rechtermuisknop op het object. Je ziet dan het menupunt `Inspect`. Kies dit en je krijgt het volgende te zien:

Figuur 1.8. De toestand van een object

1.5. Een extra object aanmaken

Je kan meerdere `Temperatuur` objecten tegelijkertijd in gebruik hebben. Maak meerder `Temperatuur` objecten aan en geef ze elk een eigen Celsius waarde. Ga dan de toestanden na.

Het moet nu wel duidelijk zijn dat tijdens de levensduur van een programma meerdere objecten van een zelfde klasse kunnen bestaan. Het gedrag van de objecten is beschreven in de broncode van de klasse waartoe de objecten behoren. We hoeven dus niet voor elk object van dezelfde klasse de broncode aan te passen.

1.6. Objecten versus klassen

Klassen vormen de kern van het object-georiënteerd programmeren. We hebben bij de eerste kennismaking met Bluej gezien dat je eerst een klasse moet maken. Deze klasse wordt dan gecompileerd en daarna kan je er één of meerdere objecten van maken. In Java kan van deze werkwijze niet afgeweken worden. Je moet altijd eerst een klasse ontwerpen voordat je met de objecten ervan kan werken. Bij het ontwerp van een klasse worden de velden en de methode voor de klasse vastgelegd. De velden bevatten de gegevens en de methoden bepalen welke acties op deze velden ingrijpen. De toegang tot de velden verloopt altijd via de methoden.

Oefeningen

1. Voeg in de klasse `Temperatuur` een methode bij die Fahrenheit omzet naar Celsius.
2. Maak een `Temperatuur` object en open daarna de broncode. Wijzig iets in de commentaar en klik op compile. Ga na wat er gebeurt met het object dat je eerder had gemaakt.

Hoofdstuk 2. Ingrediënten van klassen

2.1. Een zeer eenvoudige Breuk klasse

We starten met een eenvoudige implementatie van een klasse voor de opslag van een breuk. Logischerwijze wordt de naam van de klasse ook `Breuk`.

```
public class Breuk
{
    private int teller;
    private int noemer;

    // constructor
    public Breuk()
    {
        teller = 0;
        noemer = 0;
    }
}
```

Voor de opslag van de gegevens van een breuk heb je een teller en een noemer nodig. Daarom worden twee velden `teller` en `noemer` gedeclareerd. Beide velden kunnen alleen maar gehele getallen opslaan, vandaar het `int` type dat in de declaratie voorkomt. Beide velden zijn voor exclusief gebruik binnen de klasse. Daarom staat het woord `private` bij de declaratie.

De klasse `Breuk` heeft ook een constructor. Dit is een speciale methode die je nodig hebt om objecten te kunnen aanmaken. De constructor heeft dezelfde naam als de klasse en de beide velden `teller` en `noemer` krijgen een nul als startwaarde. Of dit wel een goede startwaarde is, zal later wel blijken.

2.2. Een methode om het quotiënt uit te rekenen

Bij het ontwerp van de klasse `Breuk` hebben we ervoor geopteerd om zowel de teller als de noemer als gehele getallen op te slaan. Als je wil weten wat de waarde van de breuk na deling van teller door noemer is, kan je best een methode maken die dit uitrekent. Dit is de methode die hiervoor uitgeschreven is:

```
public double berekenQuotient()
{
    return teller / noemer;
}
```

Deze methode krijgt de naam `berekenQuotient`, is publiek gemaakt met het woord `public`. Dit betekent dat iedereen van buiten de klasse ¹ deze methode kan starten.

Als je deze methode test, dan zal je vaststellen dat er meteen een foutmelding komt. Er wordt namelijk door nul gedeeld. Dit wordt niet toegestaan door de computer. De uitvoering van het programma zal vroegtijdig afgebroken worden. Je kan beter deze fout vermijden door in de constructor een andere startwaarde voor het veld `noemer` te kiezen. De waarde 1 is een betere kandidaat. Dit zal dan een quotiënt van 0.0 geven.

2.3. Een verbeterde constructor

Deze versie van de constructor plaatst een 0 in de teller en een 1 in de noemer. De toekenning naar `noemer` is gewijzigd: er staat nu `noemer = 1;` in plaats van `noemer = 0;`.

```
public Breuk()
{
    teller = 0;
    noemer = 1; // nieuwe startwaarde
}
```

¹Wat 'van buiten de klasse' betekent, kan pas uitgelegd worden wanneer je meerdere samenwerkende klassen in hetzelfde project hebt gemaakt. Dit is nu nog te vroeg.

2.4. Een extra constructor

Om gemakkelijk nieuwe waarden voor teller en noemer in een bestaand object in te geven, hebben we een nieuwe constructor nodig. Deze constructor ziet er zo uit:

```
public Breuk(int te, int no)
{
    teller = te;
    noemer = no;
}
```

Deze constructor kan je gebruiken om objecten aan te maken die meteen een andere waarde voor teller en noemer krijgen dan 0/1.

2.5. setter methodes

Je kan ook methoden bijvoegen in de klasse waarmee het mogelijk is een nieuwe waarde voor de teller of de noemer in een breuk in te stellen. Deze werkwijze heeft als voordeel dat je geen nieuw object hoeft te komen als alleen maar de interne waarden van teller en noemer gewijzigd moeten worden. We maken twee nieuwe methoden: `setTeller()` en `setNoemer()`.

```
public void setTeller(int te)
{
    teller = te;
}

public void setNoemer(int no)
{
    noemer = no;
}
```

Elk van deze twee methoden ontvangt een parameter; het is telkens de nieuwe waarde voor respectievelijk de teller of de noemer. Parameters zijn speciale variabelen die fungeren als doorgeefluik van gegevens buiten de klasse naar een velden. Bij het vastleggen van de methoden zie achtereenvolgens:

- Met `public` wordt aangegeven dat de methode door iedereen gestart kan worden.
- Het woord `void`² geeft aan dat de methode geen resultaat teruggeeft.
- Dan volgt de naam van de methode, bij de eerste is dat `setTeller` en bij de tweede is dat `setNoemer`.
- Na de methodenaam volgt de parameterlijst, die tussen ronde haken geschreven wordt. Bij de eerste methode is dat `int te`, bij de tweede `int no`. Bij de oproep van de methode moet telkens één waarde doorgegeven worden die terecht komt in deze variabele.
- Na het vastleggen van de regel met de naam van methode volgt een blok met de acties. Dit blok wordt aangegeven met de accoladen `{ }`.
- De enige actie in elk van de methode is een toekenningstatement dat de doorgegeven waarde naar het veld `teller` of `noemer` kopieert.

2.6. Een verbeterde quotientberekening

Als je een beetje experimenteert met nieuwe waarden voor teller en noemer, zal je zien dat het resultaat van `berekenQuotient()` meestal niet klopt. Dit komt omdat de uitdrukking `teller/noemer` een gehele deling uitvoert. Bij dit type deling wordt voor het quotiënt alleen het gehele deel overgehouden. Zo geeft de deling $7/2$ het quotiënt 3 en niet 3.5. Je kan dit probleem alleen maar oplossen door `teller` om te zetten van geheel getal naar getal met cijfers na de komma. Dit doe je met geforceerde omzetting:

```
(double) teller
```

Door een type tussen haken te schrijven wordt de waarde die erop volgt omgezet naar dit type. We passen dit toe in de methode.

```
public double berekenQuotient()
{
    return ((double) teller) / noemer;
}
```

Je ziet hier dat er nog extra haken zijn bijgevoegd om er voor te zorgen dat eerste de geforceerde omzetting naar `double` en dan pas de deling. De deling zal als `double` uitgevoerd omdat één van de waarden voor en na de deelstreep een `double`. Dit betekent dat het volstaat om ofwel `teller` of `noemer` naar `double` om te zetten.

2.7. Een quotiëntberekening die niet door 0 deelt

Omdat je met de methode `setNoemer()` nog altijd een 0 in de noemer van een `Breuk` object, bestaat de mogelijkheid om op een deling door 0 te botsen. Daarom moet je een beveiliging inbouwen die er voor zorgt dat er geen deling uitgevoerd wordt wanneer de noemer 0 is. Hiervoor gebruiken een conditioneel statement. In Java wordt dat de `if` constructie.

```
public double berekenQuotient()
{
    if (noemer != 0)
    {
        return ((double) teller) / noemer;
    }
    else
    {
        return 0.0;
    }
}
```

Wanneer de methode `berekenQuotient()` gestart wordt, kom je eerst bij de `if`. Dit statement voert dan de test `noemer != 0` uit, die je moet lezen als 'is noemer verschillend van 0?'. Het resultaat is waar of niet waar. Er horen twee blokken bij deze `if`. Het eerste blok wordt uitgevoerd wanneer de test waar is, het blok na `else` wordt uitgevoerd als de voorwaarde niet waar is. Dus één van beide blokken wordt uitgevoerd; in het eerste blok wordt de deling uitgerekend en teruggegeven met `return`; in het tweede blok wordt enkel een 0 teruggegeven. Voorlopig is dit een redelijke oplossing. Het programma zal niet meer afgebroken worden door een deling door 0.

2.8. Een breuk omkeren

Tot slot maken we nog een methode die de breuk omkeert. Hiervoor moet je `teller` en `noemer` verwisselen. Dit is de eerste versie van deze methode.

```
public void keerom()
{
    teller = noemer;
    noemer = teller;
}
```

Je zal snel vaststellen dat deze methode niet werkt: van zodra de waarde van de `noemer` in de `teller` wordt geschreven, ben je de originele waarde van de `teller` kwijt. Je hebt deze waarde nodig omdat die naar de `noemer` moet gekopieerd worden. Er zit niets anders op dan een extra tijdelijke variabele te gebruiken.

```
public void keerom()
{
    int hulp;
    hulp = teller;
    teller = noemer;
    noemer = hulp;
}
```

De variabele `hulp` is een tijdelijke variabele waarin je een geheel getal kan opslaan. Deze variabele bestaat alleen maar tijdens het uitvoeren van de methode. Na het beëindigen van de methode wordt de variabele geschrapt. Dit is precies wat we nodig hebben: een variabele om tijdelijk de originele waarde

van de teller bij te houden. Als laatste stap wordt de waarde van hulp naar noemer gekopieerd en is de breuk omgekeerd. Dit type tijdelijke variabele wordt meestal benoemd met de term 'lokale variabele'.

2.9. De volledig afgewerkte klasse Breuk

Dit is tot slot de afgewerkte klasse Breuk.

```

/**
 * Een eenvoudige klasse voor de opslag van een breuk.
 *
 * @author L. Rutten
 * @version 22/ 9/2009
 */
public class Breuk
{
    /**
     * De teller van de breuk
     */
    private int teller;

    /**
     * De noemer van de breuk
     */
    private int noemer;

    /**
     * Constructor voor Breuk met waarde 0/1
     */
    public Breuk()
    {
        // initialise instance variables
        teller = 0;
        noemer = 1;
    }

    /**
     * Constructor voor Breuk
     * met externe startwaarden
     */
    public Breuk(int te, int no)
    {
        // initialise instance variables
        teller = te;
        noemer = no;
    }

    /**
     * plaats een nieuwe waarde voor de teller
     */
    public void setTeller(int te)
    {
        teller = te;
    }

    /**
     * plaats een nieuwe waarde voor de noemer
     */
    public void setNoemer(int no)
    {
        noemer = no;
    }

    /**
     * Bereken het quotient van de breuk
     *
     * @return    quotient
     */
    public double berekenQuotient()
    {
        if (noemer != 0)
        {
            return ((double) teller) / noemer;
        }
        else
        {
            return 0.0;
        }
    }

    /**
     * Keer een breuk om
     */
    public void keerom()
    {
        int hulp;

        hulp = teller;
        teller = noemer;
        noemer = hulp;
    }
}

```

Oefeningen

3.

Hoofdstuk 3. Keuzen maken in methodes

3.1. Een eenvoudige klasse voor een bankrekening

Dit is de klasse `Rekening`. Hiermee kan je de stand van een bankrekening bijhouden. De uitwerking van deze klasse is nog beperkt; er ontbreken nog heel wat beveiligingen en de limiet is voorlopig nog vast. We bekijken eerste deze versie van de klasse en daarna gaan we deze klasse verder aanvullen met extra's.

De eerste stap bij het ontwerp van een klasse is altijd de inventaris van de velden opstellen. Met klassen kan je objecten maken en wanneer deze objecten tot leven komen, hebben ze interne ruimte voor gegevens. Alle velden van een object zorgen samen voor deze gegevensopslag. In deze klasse willen we de informatie over een bankrekening bijhouden. Het eerste veld is ook het meest evidente: je moet de stand van de rekening bijhouden. Dit levert ons het veld `stand` op.

```
private double stand;
```

We maken een bankrekening die toelaat dat de stand onder nul gaat. Om een te grote negatieve stand te vermijden heb je een limiet nodig. Dit wordt het veld `limiet`.

```
private double limiet;
```

We willen de limiet in de loop van de tijd variabel maken. We hebben dan ook een criterium nodig waarmee de limiet bepaald kan worden. Voor deze rekening wordt dit het gemiddelde van alle stortingen naar de rekening. Om dit gemiddelde te kunnen berekenen heb je het totaal van alle stortingen en het aantal stortingen nodig. Deze twee waarden worden in de velden `totaalstortingen` en `aantalstortingen` bijgehouden.

```
private int    aantalstortingen;  
private double totaalstortingen;
```

Nadat de velden zijn vastgelegd komen de methoden aan de beurt. Je moet nagaan welke bewerkingen nodig zijn voor een bankrekening. Deze analyse levert een lijst van bewerkingen op en elke bewerking wordt een methode in de klasse. De bewerkingen zijn het maken van een nieuwe bankrekening, een storting uitvoeren, geld afhalen en het instellen van de limiet. We overlopen even de methoden.

Om nieuwe objecten te kunnen maken heb je een constructor nodig. Deze constructor initialiseert de velden met een startwaarde. Voor alle velden is dit de waarde 0. De constructor heeft geen parameters nodig.

```
public Rekening()  
{  
    stand = 0;  
    aantalstortingen = 0;  
    totaalstortingen = 0;  
    limiet = 0;  
}
```

Wanneer je een bedrag op de rekening stort, moet je deze methode gebruiken. De methode `stort` verwacht het te storten bedrag als parameter en verhoogt de stand van de rekening met deze waarde. Om later het gemiddelde correct te kunnen verhogen, wordt het veld `aantalstortingen` met één verhoogd en wordt het gestorte bedrag ook bij het totaal in veld `totaalstortingen` geteld.

```
public void stort(double bedrag)  
{  
    stand = stand + bedrag;  
    aantalstortingen = aantalstortingen + 1;  
    totaalstortingen = totaalstortingen + bedrag;  
}
```

Bij het afhalen van een geldbedrag wordt de stand verlaagd. Deze methode geeft het afgehaalde bedrag terug. Later wordt duidelijk waarom dit nodig is.

```
public double haalaf(double bedrag)
{
    stand = stand - bedrag;
    return bedrag;
}
```

De klasse Rekening is nu afgewerkt in een eerste en eerder beperkte versie en ziet er nu zo uit:

```
/**
 * Een klasse voor het bijhouden van een bankrekening.
 * De stand van de rekening wordt bijgehouden samen
 * met het aantal stortingen en het totale gestorte bedrag.
 * Er is ook een limiet. Deze is momenteel vast.
 *
 * @author Leo Rutten
 * @version $Date: 2010-03-01 16:40:37 +0100 (Mon, 01 Mar 2010) $
 */
public class Rekening
{
    /**
     * stand van de rekening
     */
    private double stand;

    /**
     * het totale aantal stortingen
     */
    private int    aantalstortingen;

    /**
     * de som van alle stortingen
     */
    private double totaalstortingen;

    /**
     * hoeveel je onder 0 mag gaan
     */
    private double limiet;

    /**
     * Constructor voor nieuwe rekeningen
     */
    public Rekening()
    {
        stand = 0;
        aantalstortingen = 0;
        totaalstortingen = 0;
        limiet = 0;
    }

    /**
     * plaats geld op je rekening
     *
     * @param bedrag te storten bedrag
     */
    public void stort(double bedrag)
    {
        stand = stand + bedrag;
        aantalstortingen = aantalstortingen + 1;
        totaalstortingen = totaalstortingen + bedrag;
    }

    /**
     * haal geld af
     *
     * @param bedrag af te halen bedrag
     */
    public double haalaf(double bedrag)
    {
        stand = stand - bedrag;
        return bedrag;
    }

    /**
     * stel limiet in
     */
    public void berekenLimiet()
    {
        limiet = 1000;
    }
}
```

3.2. Verbeteringen aanbrengen

Het is nodig om de klasse op een aantal vlakken te verbeteren: enerzijds zijn er beveiligingen zoals de controle van de limiet die ontbreekt en anderzijds zijn er bewerkingen die ontbreken zoals het instellen van de limiet volgens het gemiddelde van de stortingen. De eerste verbetering is het controleren van de afhalingen.

3.2.1. Controle afhalingen bijvoegen

Je mag niet meer geld afhalen dan dat de limiet toelaat. Daarom moet de methode `haalaf` verbeterd worden. Je moet nagaan of er nog genoeg geld op de rekening staat. Dit kan je uitdrukken met de voorwaarde `stand + limiet >= bedrag`. Met een `if` kan je deze voorwaarde testen. Hierdoor ontstaan er twee mogelijke situaties:

- Er staat genoeg geld op de rekening, in dat geval mag je `stand` verlagen en het bedrag als resultaat teruggeven.
- Er staat niet genoeg geld op de rekening, in dat geval blijft de `stand` ongewijzigd en geef je 0 als resultaat terug.

Het `if` statement wordt in de methode geïntegreerd.

```
public double haalaf(double bedrag)
{
    if (stand + limiet >= bedrag)
    {
        stand = stand - bedrag;
        return bedrag;
    }
    else
    {
        return 0;
    }
}
```

3.2.2. Alleen positieve limieten instellen

Hiervoor is opnieuw een controle nodig. Indien er een negatief getal als parameter doorgegeven wordt, dan wordt het omgekeerd. Bij deze `if` heb je geen `else` nodig. Merk op dat de naam van de methode ook gewijzigd is: deze heet nu `setLimiet`. De methode heeft alleen maar het instellen van een nieuwe limiet als taak, vandaar de nieuwe naam. Voor het berekenen van een limiet gaan we later een andere methode maken.

```
public void setLimiet(double lmt)
{
    if (lmt < 0.0)
    {
        lmt = - lmt;
    }
    limiet = lmt;
}
```

De doorgegeven parameter `lmt` wordt getest, indien negatief, volgt een omkering.

3.2.3. Alleen positieve bedragen storten

Ook hier volstaat een `if` die de parameter test. In andere woorden, je moet alleen maar een `if` bijvoegen. De rest van de methode blijft ongewijzigd.

```
public void stort(double bedrag)
{
    if ( bedrag < 0)
    {
        bedrag = -bedrag;
    }

    stand = stand + bedrag;
    aantalstortingen = aantalstortingen + 1;
    totaalstortingen = totaalstortingen + bedrag;
}
```

3.2.4. De limiet instellen volgens de gemiddelde stortingen

Dit doen we op twee manieren, eerst een berekening volgens een percentage van de gemiddelde storting. Daarna gaan we de rekening volgens de gemiddelde storting in één van meerdere categorieën onderbrengen. Voor elke categorie geldt een vaste limiet.

Je kan het veld `limiet` rechtstreeks wijzigen door een toekenningsstatement te gebruiken.

3.2.4.1. Limiet volgens percentage

Deze methode is de eenvoudigste. Je berekent eerste het gemiddelde van de stortingen met de deling `totaalstortingen/aantalstortingen`, daarna neem je daarvan een percent en deze waarde sla je op in het veld `limiet`. Je kan het veld `limiet` rechtstreeks wijzigen door een toekenningstatement te gebruiken, maar je kan ook zelf de methode `setLimiet()` starten. Dit is het resultaat: de methode `berekenLimietPercent()`.

```
public void berekenLimietPercent()
{
    if (aantalstortingen > 0)
    {
        // de limiet is 10 % van de gemiddelde storting
        setLimiet(totaalstortingen/aantalstortingen * 10.0/100.0);
    }
}
```

Er is in de bovenstaande methode nog een extra `if`-statement bijgevoegd om te vermijden dat het gemiddelde uitgerekend wordt als er nog geen stortingen zijn gebeurd. Het veld `aantalstortingen` is dan nog nul en met deze waarde riskeer je een deling door nul. Dit zou het programma voortijdig afbreken, iets wat niet wenselijk is.

Het gebruik van het `if`-statement in deze situatie is een voorbeeld van wat men *defensief programmeren* noemt. Deze programmeerstijl levert programma's op die in alle situaties correct reageren.

3.2.4.2. Limiet volgens categorie

Met de tweede methode voor de berekening van de limiet wordt ook het gemiddelde van alle stortingen berekend. Deze keer wordt het gemiddelde gebruikt om te bepalen in welke categorie de klant thuishoort. Omdat je met één `if`-statement en de bijbehorende test slechts kan opsplitsen in twee categorieën, moet je nog een extra `if`-statement gebruiken.

Met de eerste `if` wordt getest of het gemiddelde lager dan 500 is. Indien ja, valt de rekening in de categorie 'slechte klant'. Dit wordt gemeld op het scherm en de limiet wordt op 100 ingesteld. Deze `if` heeft ook een `else` kant. Hier kom je terecht als het gemiddelde groter of gelijk is aan 500. In dat geval moet je een nieuwe test laten uitvoeren om na te gaan of het gemiddelde onder of boven 1000 valt. Als de test waar is, kom je in categorie tussen 500 en 1000. Als de test niet waar is, is de categorie boven 1000.

De constructie die hier is opgebouwd, levert ons een `if` binnen een andere `if` op. Denk er ook aan dat je deze methode minstens driemaal moet testen, anders ben je niet zeker of alle categorieën wel werken.

Aan het begin van de methode staat nog een `if`. Deze zorgt ervoor dat we niet door nul delen. Deze `if` heeft alleen een blok met als enige statement een `return` zonder waarde erachter. We gebruiken dit om ogenblikkelijk de uitvoering van de methode te verlaten.

```
public void berekenLimietCategorie()
{
    if (aantalstortingen == 0)
    {
        return;
    }
    // bereken gemiddelde
    double gemiddelde = totaalstortingen/aantalstortingen;

    if (gemiddelde < 500.00)
    {
        // slechte klant
        System.out.println("slechte klant");
        setLimiet(100);
    }
    else
    {
        if (gemiddelde < 1000)
        {
            // goede klant
            System.out.println("goede klant");
            setLimiet(500);
        }
        else
        {

```

```

        // zeer goede klant
        System.out.println("zeer goede klant");
        setLimiet(1000);
    }
}

```

3.3. De volledig afgewerkte klasse `veiligeRekening`

Alle besproken verbeteringen zijn nu in de klasse opgenomen. Ondertussen hebben we ook de naam van de klasse gewijzigd in `veiligeRekening`, dit geeft aan dat deze klasse een veilige versie is van het eerste ontwerp.

```

/**
 * Een klasse voor het bijhouden van een bankrekening.
 * De stand van de rekening wordt bijgehouden samen
 * met het aantal stortingen en het totale gestorte bedrag.
 *
 * Er is ook een limiet. Deze is momenteel vast.
 *
 * @author Leo Rutten
 * @version 2009-10-04 13:01:16 +0200 (Sun, 04 Oct 2009)
 */
public class VeiligeRekening
{
    /**
     * stand van de rekening
     */
    private double stand;

    /**
     * het totale aantal stortingen
     */
    private int    aantalstortingen;

    /**
     * de som van alle stortingen
     */
    private double totaalstortingen;

    /**
     * hoeveel je onder 0 mag gaan
     * uitgedrukt met een positieve waarde
     */
    private double limiet;

    /**
     * Constructor voor nieuwe rekeingen
     */
    public VeiligeRekening()
    {
        stand = 0;
        aantalstortingen = 0;
        totaalstortingen = 0;
        limiet = 0;
    }

    /**
     * plaats geld op je rekening
     *
     * @param bedrag te storten bedrag
     */
    public void stort(double bedrag)
    {
        if ( bedrag < 0 )
        {
            bedrag = -bedrag;
        }

        stand = stand + bedrag;
        aantalstortingen = aantalstortingen + 1;
        totaalstortingen = totaalstortingen + bedrag;
    }

    /**
     * haal geld af
     *
     * @param bedrag af te halen bedrag
     */
    public double haalaf(double bedrag)
    {
        if (stand + limiet >= bedrag)
        {
            stand = stand - bedrag;
            return bedrag;
        }
        else
        {
            return 0;
        }
    }
}

```

```

/**
 * stel limiet opnieuw in
 */
public void setLimiet(double lmt)
{
    if (lmt < 0.0)
    {
        lmt = - lmt;
    }
    limiet = lmt;
}

/**
 * bereken de limiet in volgens percent
 * en stel opnieuw in
 */
public void berekenLimietPercent()
{
    if (aantalstortingen > 0)
    {
        // de limiet is 10 % van de gemiddelde storting
        setLimiet(totaalstortingen/aantalstortingen * 10.0/100.0);
    }
}

/**
 * bereken de limiet in volgens categorie
 * en stel opnieuw in
 */
public void berekenLimietCategorie()
{
    if (aantalstortingen == 0)
    {
        return;
    }
    // bereken gemiddelde
    double gemiddelde = totaalstortingen/aantalstortingen;
    if (gemiddelde < 500.00)
    {
        // slechte klant
        System.out.println("slechte klant");
        setLimiet(100);
    }
    else
    {
        if (gemiddelde < 1000)
        {
            // goede klant
            System.out.println("goede klant");
            setLimiet(500);
        }
        else
        {
            // zeer goede klant
            System.out.println("zeer goede klant");
            setLimiet(1000);
        }
    }
}
}
}

```

Oefeningen

4. Maak een klasse voor de opslag van de punten van een student. Je moet enkel maar het totaal aantal behaalde punten bijhouden. Deze klasse krijgt ook een methode `toonGraad()` die de behaalde graad weergeeft. Voor de graad hebben we:

- < 50: niet geslaagd
- tussen 50 en 70: geslaagd
- tussen 70 en 80: onderscheiding
- tussen 80 en 85: grote onderscheiding
- > 85: grootste onderscheiding

5. Maak een uitbreiding op de voorgaande oefening: hou met een `true` of `false` waarde bij of de student aan de examens heeft meegedaan. Als hij meegedaan heeft, dan wordt de graad bepaald zoals in de vorige oefening. Als hij niet meegedaan heeft, dan komt er bij het uitvoeren van `toonGraad()` de melding afwezig.

Hoofdstuk 4. Herhalingsstatement

Met het herhalingsstatement kan je een actie meerdere keren uitvoeren. Voor veel problemen heb je deze constructie nodig. Als je bijvoorbeeld de som van een reeks getallen moet je telkens het totaal verhogen met elk van de getallen uit deze reeks. Dit kan je alleen maar laten uitvoeren door een herhalingsstatement.

Er zijn verschillende soorten herhalingsstatements; de eerste die we behandelen is het `for`-statement.

4.1. Een eenvoudige herhaling met het `for`-statement

Met een `for` kan je een variabele laten variëren vanaf een start- tot een eindwaarde. In het volgende voorbeeld de variabele `i` degene die varieert. Voor elke waarde in het aangegeven bereik wordt de bijbehorende actie éénmaal uitgevoerd.

```
for (int i=0; i<10; i++)
{
    System.out.println("hallo wereld " + i);
}
```

Het bovenstaande voorbeeld laat de tekst `hallo wereld` tienmaal zien. Dit is de uitvoer:

```
hallo wereld 0
hallo wereld 1
hallo wereld 2
hallo wereld 3
hallo wereld 4
hallo wereld 5
hallo wereld 6
hallo wereld 7
hallo wereld 8
hallo wereld 9
```

De syntax van het `for`-statement is als volgt:

- Na het `for` sleutelwoord schrijf je een ronde hakenpaar.

```
for ( )
```

- Tussen de haken noteer je de declaratie van een variabele met initialisatie. Meestal is de variabele van het type `int`. Sluit af met een puntkomma. Met de initialisatie bepaal je de startwaarde van de variabele¹.

```
for (int i=0; )
```

- Hierna volgt een voorwaarde gevolgd door een puntkomma. De voorwaarde bepaalt de eindwaarde. In dit geval is dit `i<10`, dit betekent dat de hoogste waarde voor `i` 9 zal zijn.

```
for (int i=0; i<10; )
```

- Tot slot schrijf je de verhoging. Deze actie zorgt ervoor dat de waarde van de luster omhoog gaat.

```
for (int i=0; i<10; i++)
```

- De actie die voor elke waarde van de luster uitgevoerd wordt, schrijf je tussen accoladen onder de `for`.

```
for (int i=0; i<10; i++)
{
    System.out.println("hallo wereld " + i);
}
```

4.2. Een eenvoudige herhaling met het `while`-statement

Je kan het voorgaande voorbeeld ook met een `while` uitschrijven in plaats van met een `for` statement. Dat ziet er dan zo uit:

```
int j=0;
while (j<10)
{
    System.out.println("hallo wereld " + j);
    j++;
}
```

Je herkent gemakkelijk de elementen die we bij het `for`-statement geleerd hebben. Dit zijn:

- de declaratie van de variabele met inktialisatie: `int j=0;`
- de voorwaarde: `j<10`
- de verhoging van de lusteller: `j++;`

De werking van de twee voorbeelden is identiek; je leert hiermee dat je een probleem op twee manieren kan oplossen, ofwel met het `for`-statement ofwel met het `for`-statement. In feite heb je bij dit probleem de keuze tussen beide: zowel `for` als `while` zijn geschikt om het probleem van het tonen van de getallen van 0 tot en met 9 op te lossen.

In de volgende sectie zullen we zien dat er problemen zijn waar je op voorhand niet kan inschatten hoeveel iteraties² er zullen zijn. In dat geval komt alleen het `while`-statement in aanmerking.

4.3. Een `while` met onbekend aantal iteraties

In het volgende voorbeeld wordt een bedrag telkens verminderd tot de waarde negatief wordt. Bij zulke bewerking weet je niet op voorhand hoeveel iteraties er zullen zijn. Het bedrag start bij 1000 en wordt telkens met 153 verminderd.

```
int bedrag = 1000;
int aantal = 0;
while (bedrag > 0)
{
    bedrag = bedrag - 153;
    System.out.println("bedrag " + bedrag);
    aantal++;
}
System.out.println("restbedrag " + bedrag);
System.out.println("aantal " + aantal);
```

In feite hou je bij het oplossen van dit probleem geen rekening met het exact aantal iteraties. Om toch te weten hoeveel iteraties er zullen zijn, moet je een extra variabele, in dit geval `aantal`, bijvoegen om dit te berekenen.

Als besluit bij de laatste twee voorbeelden kunnen we stellen dat een `for` alleen maar geschikt is als je het exacte aantal iteraties op voorhand kent. In de andere gevallen moet je een `while` gebruiken.

4.4. Bereken de som van getallen

In dit voorbeeld zie je dat de lusteller niet altijd bij 1 moet starten. Bij de voorwaarde zie je dat er kleiner of gelijk staat. Dit betekent dat de bovengrens ook in het bereik zit.

```
// bereken de som van 2 tot 1000000
int som = 0;
for (int g=2; g<=1000000;g++)
{
    som = som + g;
}
System.out.println("totale som is " + som);
```

4.5. Meerdere herhalingsstatements combineren

Je kan ook meerdere herhalingsstatements combineren zodat de ene binnen de andere wordt gebruikt. Je kan intuïtief aanvoelen dat het aantal iteraties hiermee sterk opgedreven wordt. In het volgende probleem willen we rijen en kolommen weergeven. Dit betekent dat we meerdere rijen weergeven die elk bestaan uit meerdere kolommen. Je merkt aan de vorige zin dat het woord meerdere er tweemaal in voorkomt. Hierdoor kom je tot vaststelling dat er twee `for`-statements nodig zijn.

²Een iteratie is één stap tijdens de uitvoering van een herhalingsstatement

Omdat de ene `for` binnen de andere staat, heb je twee verschillende lustellers nodig: `i` en `j`. Als je toch dezelfde lusteller voor de beide `for`'s zou gebruiken, dan loopt het programma fout. Probeer hiervoor zelf een verklaring te achterhalen.

```
public void toonMatrix(int nr, int nk)
{
    // doorloop alle rijen
    for (int r=0; r<nr; r++)
    {
        // doorloop alle kolommen
        for (int k=0; k<nk; k++)
        {
            System.out.print(r + "," + k + " ");
        }
        System.out.println("");
    }
}
```

4.6. Toon alle delers van een geheel getal

Bij alle voorgaande problemen was de actie binnen de herhaling eenvoudig. Dit is het eerste voorbeeld waar je een `if` binnen een `for` ziet. Het de methode in het volgende programmafragment toont alle delers van de parameter `getal`. De lusteller `deler` doorloopt alle waarden in het bereik van 2 tot en met `deler - 1`.

```
public void toonAlleDelers(int getal)
{
    for (int deler=2; deler<getal; deler++)
    {
        // is de rest na deling gelijk aan 0?
        if (getal % deler == 0)
        {
            System.out.println("deler " + deler);
        }
    }
}
```

Omdat je niet elke waarde van de lusteller als een deler kan beschouwen, moet je met een `if` testen of we wel met een echte deler te maken hebben. Een deler is pas een echte deler als de rest na een gehele deling nul is. Hiervoor heeft Java een speciale operator, het teken `%`. Hier zijn enkele voorbeelden met deze operator:

- `5 % 3` geeft 2
- `6 % 3` geeft 0
- `7 % 3` geeft 1
- `2 % 3` geeft 2

Alle getallen is de bovenstaande voorbeelden zijn gehele getallen.

4.7. Test priemgetallen

De methode in het vorige voorbeeld kan gemakkelijk aangepast worden in een methode die nagaat of een getal een priemgetal is. Een priemgetal is een getal dat alleen maar 1 en zichzelf als deler heeft. We lossen dit op door na te gaan hoeveel delers er zijn in het bereik van 2 tot `deler - 1`.

```
public boolean isPriemGetal(int getal)
{
    int aantalDelers = 0;
    for (int deler=2; deler<getal; deler++)
    {
        if (getal % deler == 0)
        {
            //System.out.println("deler " + deler);
            aantalDelers++;
        }
    }
    if (aantalDelers == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

```
}
}
```

De methode die hier gepresenteerd wordt, telt eerst het aantal delers en test daarna of dit aantal nul is. Indien ja, wordt een `true` als waarde teruggegeven, anders een `false`. De methode `isPriemGetal` geeft het type `boolean` voor de terugkeerwaarde.

4.8. Toon alle priemgetallen

Met de methode `isPriemGetal` als basis kan je een stap verder zetten: we maken nu een methode die alle priemgetallen in een bereik toont. Om dit probleem op te lossen maken we gebruik van de methode `isPriemGetal`. Het ontwerpen van de nieuwe methode is niet moeilijk: je doorloopt het bereik en test voor elke waarde van de lusteller of het een priemgetal is. De methode ziet er zo uit:

```
public void toonAllePriemGetallen(int bovengrens)
{
    // voor alle getallen van 2 tot bovengrens
    for (int g=2; g<=bovengrens; g++)
    {
        // is dit getal een priemgetal?
        if (isPriemGetal(g))
        {
            System.out.println("priemgetal " + g);
        }
    }
}
```

We zien dat bij deze oplossing gebruik gemaakt wordt van een reeds bestaande methode. Deze werkwijze om al het werk op te splitsen in verschillende methoden noemt men functionele decompositie. Het is een veelgebruikte techniek om complexe problemen in deelproblemen op te splitsen.

4.9. Alle herhalingen in één voorbeeld

Dit zijn alle herhalingen gebundeld in één klasse.

```
/**
 * De klasse Herhalingen
 *
 * @author L. Rutten
 */
public class Herhalingen
{
    public Herhalingen()
    {
    }

    public void doe()
    {
        for (int i=0; i<10; i++)
        {
            System.out.println("hallo wereld " + i);
        }

        int j=0;
        while (j<10)
        {
            System.out.println("hallo wereld " + j);
            j++;
        }

        int bedrag = 1000;
        int aantal = 0;
        while (bedrag > 0)
        {
            bedrag = bedrag - 153;
            System.out.println("bedrag " + bedrag);
            aantal++;
        }
        System.out.println("restbedrag " + bedrag);
        System.out.println("aantal " + aantal);

        // bereken de som van 2 tot 1000000
        int som = 0;
        for (int g=2; g<=1000000;g++)
        {
            som = som + g;
        }
        System.out.println("totale som is " + som);
    }
}
```

```
/**
 * Methode om matrix te tonen
 * @param nr aantal rijen
 * @param nk aantal kolommen
 */
public void toonMatrix(int nr, int nk)
{
    // doorloop alle rijen
    for (int r=0; r<nr; r++)
    {
        // doorloop alle kolommen
        for (int k=0; k<nk; k++)
        {
            System.out.print(r + "," + k + " ");
        }
        System.out.println("");
    }
}

/**
 * Toon alle delers van een geheel getal
 */
public void toonAlleDelers(int getal)
{
    for (int deler=2; deler<getal; deler++)
    {
        if (getal % deler == 0)
        {
            System.out.println("deler " + deler);
        }
    }
}

/**
 * Is een getal een priemgetal
 */
public boolean isPriemGetal(int getal)
{
    int aantaldelers = 0;
    for (int deler=2; deler<getal; deler++)
    {
        if (getal % deler == 0)
        {
            //System.out.println("deler " + deler);
            aantaldelers++;
        }
    }
    if (aantaldelers == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

/**
 * Toon alle priemgetallen van 2 tot bovengrens
 * @param bovengrens grootste getal in het bereik
 */
public void toonAllePriemGetallen(int bovengrens)
{
    // voor alle getallen van 2 tot bovengrens
    for (int g=2; g<=bovengrens; g++)
    {
        // is dit getal een priemgetal?
        if (isPriemGetal(g))
        {
            System.out.println("priemgetal " + g);
        }
    }
}
}
```

Oefeningen

6. Ontwerp een methode die de volgende driehoek op het scherm brengt.

```

1
2 3
4 5 6
7 8 9 10
11 12 13 14 15

```

De methode krijgt één parameter: het aantal cijfers. In het bovenstaande voorbeeld is dat 15.

7. Maak een methode die een positief reëel getal als parameter krijgt en dan de vierkantswortel berekent met de methode van Newton. Indien e de schatting van de vierkantswortel van x is, dan wordt een betere schatting verkregen door de formule $(e + x/e)/2$. Geef voor e de beginwaarde 1 en vervang dan e steeds weer door een betere schatting door gebruik te maken van de aangegeven formule. Stop wanneer 2 opeenvolgende schattingen tot een bepaalde nauwkeurigheid aan elkaar gelijk zijn.

8. Maak met een methode een tabel voor de omzetting van graden Celsius naar graden Fahrenheit. Geef de startwaarde, de eindwaarde en de stapgrootte van het interval door via parameter. Voorzie de tabel van een titel.

$$c = 5.0/9.0*(f - 32.0)$$

9. Toon met een methode alle getallen van 3 cijfers tussen 100 en 999 waarvoor geldt dat de som van de derde machten van de afzonderlijke cijfers gelijk is aan het getal zelf.

$$\text{Voor het getal 153 geldt: } 153 = 1^3 + 5^3 + 3^3.$$

10. Maak een methode die de getallen van de reeks van Fibonacci toont. Geef via parameter door hoeveel getallen de methode toont.

```
1 1 2 3 5 8 13 21 34 55 89
```

Elke getal is de som van de twee vorige getallen, behalve de eerste twee getallen; die zijn per definitie 1.

11. Maak een methode de waarde van de volgende reeks uitrekent.

$$1 + 1/2 + 1/3 + 1/4 + 1/5 + \dots 1/n$$

12. Maak een methode `ggd` die de grootste gemene deler van 2 positieve getallen berekent.

$$\begin{aligned} \text{ggd}(a, b) &= \text{ggd}(b, \text{rest van deling } a/b); \\ \text{ggd}(a, 0) &= a \end{aligned}$$

13. Maak een methode `kgv` die het kleinste gemeen veeldvoud van 2 positieve getallen berekent.

$$\text{kgv}(a, b) = a / \text{ggd}(a, b) * b$$

Hoofdstuk 5. Werken met lijsten

Voor veel toepassingen volstaat het niet om met variabelen te werken die enkel in staat zijn om één geheel getal (type `int`) of één floating point getal (type `double` of `float`) op te slaan. Daarom onderzoeken we in dit hoofdstuk de mogelijkheden van het `arraytype`. Dit is een type waarmee je variabelen kan declareren die in staat zijn om reeksen van gegevens van het hetzelfde type op te slaan. Als je bijvoorbeeld voor een toepassing een aantal meetwaarden hebt verzameld, kan je die opslaan in zulke arrayvariabele. Na opslag kan je er allerlei werkingen op uitvoeren zoals maximum- en minimumberekening, gemiddeldeberekening en sorteren.

De volgende sectie toont hoe je met arrayvariabelen kan starten. In de sectie daarna wordt uitgelegd hoe je arrays als objectvariabelen kan gebruiken.

5.1. Een array maken en gebruiken

Een arrayvariabele wordt als volgt gedeclareerd en moet altijd vóór gebruik geïnitieerd worden.

```
int[] reeks = new int[20];
```

De bovenstaande regel declareert eerst de variabele `reeks` met het type `int[]`. In dit type zie de rechte haken `[]`, ze geven aan dat het hier om een array gaat. Merk op dat er bij deze declaratie niet vastgelegd wordt uit hoeveel vakjes de array bestaat. Dit wordt pas bij de initialisatie gedaan. Met `new int[20]` wordt er nieuw geheugen voor 20 `int` vakjes gereserveerd. De verwijzing naar dit geheugen wordt in `reeks` opgeslagen. Deze initialisatie van `reeks` is absoluut noodzakelijk. Als je dit weglaat wordt het programma tijdens de uitvoering afgebroken.

Je mag de initialisatie ook later met een toekenning uitvoeren.

```
int[] reeks;  
reeks = new int[20];
```

De volgende klasse toont hoe je een array kan vullen met getallen en hoe de getallen getoond wordt. Dit voorbeeld in een compleet voorbeeld: je kan het compileren, er een object mee maken en de methode `vb1()` starten.

```
public class ArrayVoorbeeld1  
{  
    public ArrayVoorbeeld1()  
    {  
    }  
  
    public void vb1()  
    {  
        int n = 50;  
        int[] reeks = new int[n];  
  
        for (int i=0; i<reeks.length; i++)  
        {  
            reeks[i] = i*i;  
        }  
  
        System.out.println("Array met lengte " + reeks.length);  
        System.out.println("-----");  
        for (int i=0; i<reeks.length; i++)  
        {  
            System.out.println(" " + i + " : " + reeks[i]);  
        }  
    }  
}
```

5.2. Een array als objectvariabele

In het vorige voorbeeld werd de array gebruikt als een lokale variabele. Dit betekent dat de array verdwijnt zodra de methode terugkeert. In het volgende voorbeeld verplaatsen we de declaratie zodat de variabele een objectvariabele wordt. Dit betekent dat de array blijft bestaan zolang het object bestaat. Beide hebben een gelijke levensduur.

De klassenaam is nu `Reeks` geworden.

```
public class Reeks
{
    int[] reeks;
    public Reeks(int n)
    {
        reeks = new int[n];
    }
    public void vul()
    {
        for (int i=0; i<reeks.length; i++)
        {
            reeks[i] = 10 - i*i;
        }
    }
    public double gemiddelde()
    {
        double som = 0.0;
        for (int i=0; i<reeks.length; i++)
        {
            som = som + reeks[i];
        }
        return som/reeks.length;
    }
    public void toon()
    {
        System.out.println("Array met lengte " + reeks.length);
        System.out.println("-----");
        for (int i=0; i<reeks.length; i++)
        {
            System.out.println(" " + i + ": " + reeks[i]);
        }
    }
}
```

Er zijn heel wat meer verschillen met het vorige voorbeeld. We overlopen even:

- De declaratie van de array staat nu buiten de methoden. Dit betekent dat `reeks` een objectvariabele is geworden.
- De initialisatie van `reeks` staat nu niet meer bij de declaratie maar is verhuisd naar de constructor.

```
public class Reeks
{
    int[] reeks;
    public Reeks(int n)
    {
        reeks = new int[n];
    }
}
```

Vermits de constructor loopt op het moment dat het object gecreëerd is, heeft elk nieuw object ook een eigen ruimte voor de array.

- Het vullen van de array gebeurt met een aparte methode `vul()`.

De methode `verwissel()` loopt éénmaal door de array en verwisselt twee naburige getallen indien nodig.

```
public void verwissel()
{
    for (int i=0; i<reeks.length - 1; i++)
    {
        if (reeks[i] > reeks[i + 1])
        {
            int h = reeks[i];
            reeks[i] = reeks[i+1];
            reeks[i + 1] = h;
        }
    }
}
```

Oefeningen

14. Maak een methode die 10 getallen in een array plaatst en nagaat hoe dikwijls een groter getal direct wordt gevolgd door een kleiner getal.

15. Maak een methode die de getallen van de reeks van Fibonacci in een array plaatst.

16. Bereken de priemgetallen volgens de zeef van Eratosthenes¹. De methode gaat als volgt:

- Kies een bovengrens. Dit is het grootste getal waarvan je wil bepalen of het een priemgetal is.
- Maak een array van boolean en geef elk vakje de waarde `true`. Een `true` in een vakje betekent dat de index van het vakje een priemgetal is. Om te starten gaan we ervan uit dat alle getallen priemgetallen zijn. Later maken we bepaalde vakjes `false` omdat de meeste getallen geen priemgetal zijn.
- Begin bij het getal 2. Maak alle vakjes van de veelvouden `false`.
- Overloop alle volgende getallen. Als een getal een priemgetal is, dan moet je ook zijn veelvouden schrappen.

Hoofdstuk 6. Objecten laten samenwerken

Bijlage A. Java syntax

In deze bijlage wordt de Java syntax op een compacte wijze opgesomd. Je kan hier terecht als je niet weet hoe een bepaalde constructie moet geschreven worden.

A.1. Klassen en methodes

A.1.1. Klassedefinitie

Een klasse legt vast welke informatie en welke methoden voor een bepaald soort entiteit mogelijk zijn. Elke klasse wordt in een apart bestand beschreven. Als je meerdere klassen maakt, heb je dus meerdere bestanden. Meestal maak je de klasse `public`. Dit betekent dat iedereen in het programma deze klasse kan gebruiken.

```
public class Temperatuur
{
}
```

A.1.2. Veld

Velden of objectvariabelen leggen vast welke informatie in de objecten van deze klassen opgeslagen zullen worden. Voor elke objectvariabele schrijf je een declaratie. Schrijf eerst het type en daarna de naam van de objectvariabele. Objectvariabelen kan je afschermen tegen ongeoorloofde toegang van buiten de klassen. Daarom staat er `private` voor de declaratie. Objectvariabelen privaat maken is de gangbare praktijk. Je kan ook de woorden `protected` of `public` gebruiken. Dit laatste betekent dan iedereen van buiten de klasse toegang heeft tot de objectvariabele.

```
public class Temperatuur
{
    private double temp;
}
```

A.1.3. Type

Elke variabele moet gedeclareerd worden. Het doel van deze stap is het vastleggen van het type en de naam van de variabele. Als type kan je een enkelvoudig type of een klassenaam gebruiken. Dit zijn de enkelvoudige types:

- `byte`: 8 bit geheel getal met een bereik van -128 tot 127.
- `short`: 16 bit geheel getal met een bereik van -32768 tot 32767.
- `int`: 32 bit geheel getal met een bereik van -2147483648 tot 2147483647.
- `long`: 64 bit geheel getal met een bereik van -9223372036854775808 tot 9223372036854775807.
- `float`: 32 bit floating point met een bereik van 1.4e-45 tot 3.4028235e38.
- `double`: 64 bit floating point met een bereik van 4.9e-324 tot 1.7976931348623157e308.
- `boolean`: kent alleen de waarden `true` en `false`.
- `char`: 16 bit Unicode teken. In tegenstelling tot het `char` type in de taal C heeft `char` geen 8 bit ruimte maar 16 bit. Alle tekens uit de Unicode standaard kunnen opgeslagen worden. Unicode kan alle tekens van alle soorten talen weergeven.
- `String`: is eigenlijk geen eenvoudig type maar wel een klasse. Het wordt wel als een enkelvoudig type gebruikt.

Bij een declaratie schrijf je altijd eerst het type en dan de naam van de variabele.

```
int aantal;  
double t;  
float g;  
boolean aanwezig;
```

A.1.4. Constructor

Voor je met een object werkt (methodes ervan oproepen), moeten alle objectvariabele geïnitialiseerd zijn. Hiervoor is de constructor verantwoordelijk. Een constructor wordt geschreven als een methode en moet de naam van de klasse dragen. Een constructor mag geen terugkeertype hebben. Er mogen meerdere constructors binnen een klasse bestaan. Ze moeten wel verschillende parameterlijsten hebben. Dit is nodig om het onderscheid te maken van welke constructor in een bepaalde situatie moet gebruikt worden.

De volgende klasse heeft twee constructors:

```
public class Temperatuur  
{  
    private double temp;  
    public Temperatuur()  
    {  
        temp = 0.0;  
    }  
    public Temperatuur(double t)  
    {  
        temp = t;  
    }  
}
```

De eerste constructor geeft de temperatuur een vast waarde 0.0, bij de tweede constructor kan je een temperatuur als initialisatiewaarde meegeven.

A.1.5. Methodes

Een methode is een uitvoerbaar deel van een klasse. Elke methode heeft een naam en een terugkeertype. Parameters zijn optioneel. Ook moet je aanduiden of de methode voor iedereen toegankelijk is; je gebruikt dan `public`. De acties van een methode worden binnen een accoladenblok (`{ }`) beschreven.

```
public class Temperatuur  
{  
    private double temp;  
    public void toon()  
    {  
        System.out.println("Temperratuur " + temp);  
    }  
}
```

A.1.6. Parameters

Parameters laten toe om informatie door te geven bij de oproep van methode. Parameters zijn niet verplicht maar wel aan te raden. Ze verhogen immers de leesbaarheid van programma's. Je moet zoveel parameters voorzien als nodig. De parameterlijst staat tussen ronde haken na de naam van de methode. In het volgende voorbeeld zie je een methode die twee parameters krijgt. In dit geval is dat nodig omdat de methode de som van twee getallen gaat berekenen.

```
void som(int a, int b)  
{  
}
```

De parameterslijst, die de formele parameters vastlegt, is een lijst van declaraties telkens gescheiden door een komma. Bij de oproep van een methode moet je zoveel actuele parameters meegeven als er formele parameters zijn in de parameterlijst. Bij de `som` methode wordt dat:

```
void som(5, 6);
```

A.1.7. Return statement

Een `return` statement is nodig om een waarde als resultaat terug te geven. De `som` methode is nu met een `return` uitgerust.

```
public int som(int a, int b)
{
    return a + b;
}
```

De waarde die met `return` teruggeeft moet van hetzelfde type zijn als het type dat vlak voor de naam van de methode vermeld staat. Bij de `som` methode is dat `int`.

A.2. Controlestructuren en bewerkingen

A.2.1. Toekeningsstatement

De toekenning wordt geschreven met een `=` teken. Vóór het `=` teken staat een uitdrukking die gewijzigd kan worden. Meestal is dit de naam van een variabele maar het kan ook een element van een array zijn. Na het `=` teken staat een uitdrukking waarvan de waarde uitgerekend wordt (indien nodig) en dan in de variabele wordt geschreven.

```
a = 6;
tab[5] = 17;
tab[6] = tab[5] + 15;
```

A.2.2. Uitdrukkingen

Bij uitdrukkingen kan je de volgende rekenkundige bewerkingen en leestekens gebruiken:

- +

Dit is de optelling.

- -

Dit is de aftrekking.

- *

Dit is de vermenigvuldiging.

- /

Dit is de deling. Let op voor de deling door 0. Dit is namelijk verboden; het programma wordt dan meteen afgebroken.

- %

Met dit bewerkingsteken kan je ook een deling van twee gehele getallen uitvoeren. Als resultaat krijg je de rest van de deling.

- ()

De ronde haken dienen om delen van uitdrukkingen te groeperen zodat ze een hogere prioriteit krijgen.

A.2.3. Uitvoer

Voor uitvoer kan je gebruik van de twee methoden `println` en `print`:

```
System.out.println("hallo wereld");  
System.out.print("hallo ");  
System.out.println("wereld");
```

Er is een verschil tussen beide methoden: met `println` gaat de cursor naar het begin van de volgende regel, bij `print` is dat niet het geval.

Je mag letterlijke tekst en variabelen of uitdrukkingen laten afwisselen.

```
System.out.println("x is " + x + " en y is " + y);
```

A.2.4. Conditioneel statement

Met een `if`-statement kan je een acties voorwaardelijk uitvoeren. Alleen als de voorwaarde waar is, worden de acties uitgevoerd. Bij een `if` kan een `else` geschreven worden. Dit is optioneel.

```
if (a > 0)  
{  
    System.out.println("groter dan 0");  
}  
else  
{  
    System.out.println("niet groter dan 0");  
}
```

A.2.5. Voorwaarde

Bij een `if`-statement hoort een voorwaarde. Hierin worden altijd twee waarden met elkaar vergeleken. Dit geeft een `true` of een `false` die bepaalt of het eerste (de ja-kant) of het tweede (de nee-kant) blok uitgevoerd wordt.

```
a > 0    // groter dan  
a < 0    // kleiner dan  
a >= 0   // groter of gelijk aan  
a <= 0   // kleiner of gelijk aan  
a == 0   // gelijk aan  
a != 0   // verschillend van
```

A.2.6. Samengestelde voorwaarde

Met de tekens `&&`, `||` en `!` heb je respectievelijk de logische en, of en niet bewerking. De volgende voorwaarde gaat na of het getal `a` tussen 0 en 100 ligt, grenzen inbegrepen. De tweede voorwaarde heeft identiek dezelfde betekenis.

```
a >= 0 && a <= 100  
!(a < 0 || a > 100)
```

A.2.7. Blokstatement

Met een blok statement (geschreven met `{ }`) kan je acties groeperen.

```
if (a < 0)  
{  
    int h = a;  
    a = b;  
    b = h;  
}
```

Verklarende woordenlijst

anoniem object	Dit is een object waarvan je de verwijzing niet in een aparte variabele bijhoudt.
array	Een array is een variabele voor de opslag van meerdere gegevens van dezelfde soort. Een array wordt opgeslagen in een aaneengesloten gebied in het geheugen. Hierdoor kan je snel een gegevens uit de array halen met behulp van een index. De index heeft een vaste relatie met de geheugenplaats waar het gegeven voor deze index zich bevindt. In de meeste programmeertalen hebben arrays een vaste lengte die voor het gebruik moet vastgelegd worden.
byte-code	Java programma's worden in tekstvorm (bestanden met extensie <code>.java</code>) geschreven. De compiler zet deze tekst om naar een binaire voorstelling die opgeslagen wordt in class-bestanden (extensie <code>.class</code>). Deze class-bestanden vatten instructies voor de Java virtuele machine die
blokstatement	Met de accolades (<code>{ }</code>) kan je een reeks statements groeperen. Ze worden sequentiëel (na elkaar) uitgevoerd.
collectie	Een collectie is een speciale klasse die in staat is om meerdere objecten bij te houden. In tegenstelling tot een array kent een collectie geen beperking voor wat betreft het aantal objecten in de collectie. Collecties bestaan in meerdere vormen; sommige ervan kunnen sorteren of laten toe dat objecten in een willekeurige volgorde uit de collectie kunnen verwijderd worden.
compilatie	De computer kan de broncode van een klasse niet rechtstreeks uitvoeren. Daarom moet een klasse gecompileerd worden tot machine uitvoerbare instructies.
commentaar	Commentaar wordt niet door de computer gelezen, maar is toch essentieel voor een goede programmaarstijl. Van zodra jouw programma door andere wordt nagelezen, is de voorzien commentaar belangrijk om het programma te begrijpen.
conditioneel statement	Dit is hetzelfde als het Java if-statement.
constructor	Een constructor is een speciale methode automatisch uitgevoerd wordt vlak na het creëren van een object. Je moet de constructor gebruiken om alle velden van een object een startwaarde te geven. Een constructor moet de naam van de klasse hebben. Dit is verplicht. Je kan enkel meerdere constructors maken door telkens een andere parameterlijst te kiezen.
if-statement	
initialisatie	Elke variabele moet bij de declaratie geïnitieerd worden met een startwaarde. Je kan ervoor opteren om de initialisatie weg te laten. In dat geval moet je wel verderop in het programma met een toekenning de variabele een startwaarde geven. Dit moet gebeuren voor je de variabele voor de eerste maal leest.
iterator	Een iterator is een object dat je toelaat om doorheen een reeks object te doorlopen die in een collectie opgeslagen zijn.
klasse	Een klasse beschrijft de algemene eigenschappen van een bepaalde categorie van objecten. Dit omvat zowel de informatie (velden) die in de objecten opgeslagen wordt als de methoden die uitgevoerd kunnen worden. Een klasse fungeert ook als type.
klassediagramma	Een klassediagramma beschrijft alle klassen die in een programma voorkomen. Elke klasse wordt met een rechthoek weergegeven.

lokale variabele	Een lokale variabele wordt gedeclareerd binnen een blokstatement van een methode. Deze variabelen bestaan maar zolang de methode loopt. Wanneer de methode beëindigd wordt, worden de lokale variabelen uit het geheugen verwijderd. Dit mechanisme wordt mogelijk gemaakt door de stack gebaseerde architectuur van de Java virtuele machine.
logische operatoren	Je kan meerdere voorwaarden van een if combineren met een én of een óf. Hiermee zeg je dat respectievelijk de beide of één van beide voorwaarde moet waar zijn.
methodeoproep	Bij de start van een methode worden de statements van die methode uitgevoerd. Bij het beëindigen van de methode wordt er teruggekeerd naar de plaats van oproep. Elke methode heeft een naam en kan meerdere parameters hebben. Een methode kan ten hoogste één resultaat teruggeven. Methodes uitschrijven en ze daarna oproepen is een goede manier om het werk dat een programma moet verrichten in verschillende deeltaken te verdelen.
methodes	Methodes bevatten de functionaliteit van een programma. Een methode kan alleen maar voorkomen binnen een klasse. Dit betekent dat je eerst een object moet maken voor je een methode kan starten. Er mogen meerdere methodes binnen een klasse bestaan. Constructors zijn een speciale vorm van methodes, ze staan in voor het correct initialiseren van objectvariabelen.
object	Een object is een unieke instantie van een klasse. Een object heeft een levensduur: na creatie komt het tot leven en kunnen er methoden gestart worden.
objectendiagramma	Een objectendiagramma beschrijft alle of een aantal objecten die in een werkend programma voorkomen. Elk object wordt met een rechthoek met afgeronde hoeken weergegeven. Elk object wordt met zijn naam en klasse in de rechthoek geïdentificeerd.
objectconstructie	Wanneer je met <code>new</code> een object aanmaakt, dan wordt er eerst een blok geheugen gereserveerd, zoveel als het nieuwe object in beslag zal nemen. Daarna wordt de constructor gestart.
parameter	Met parameters kan je informatie doorgeven aan een methode. Het voordeel is dat hiermee duidelijk aangeeft wat de gebruiksaanwijziging van een methode is. Je kan meerdere parameters voorzien indien nodig.
rekenkundige operatoren	Met rekenkundige operatoren kan je berekeningen zoals optellingen, aftrekkingen, vermenigvuldigingen en delingen uitvoeren.
returnwaarde	Een methode kan ten hoogste één returnwaarde teruggeven. De oproep van zulke methode kan je dan in een uitdrukking schrijven.
return statement	Met de het returnstatement kan je de returnwaarde van een methode teruggeven.
samengestelde toekenningsstatement	
this	Het <code>this</code> sleutelwoord kan je gebruiken als verwijzing naar het huidige object.
toekenningsstatement	Met een toekenning kan je de waarde van een variabele wijzigen.
toestand	Het hele toestand van een object wordt bepaald door het geheel van al zijn objectvariabele. Als geen enkele van deze variabele gedurende een zekere tijd niet gewijzigd wordt, dan zeggen we dat de toestand van het betrokken object ongewijzigd is.

type	Een type legt de aard van een variabele vast. Een type is altijd een primitief type of een klasse.
UML	UML betekent Unified Modeling Language en is een grafische techniek om o.a. klassediagramma's voor te stellen.
variabele	Een klasse kan meerdere velden hebben. Elk veld bevat een stukje informatie. Deze informatie kan wijzigen in de loop van de tijd.
veld	Een klasse kan meerdere velden hebben. Elk veld bevat een stukje informatie. Deze informatie kan wijzigen in de loop van de tijd.
vergelijkingsoperatoren	Met vergelijkingsoperatoren kan je twee waarden met elkaar vergelijken. Twee waarden kunnen gelijk of niet gelijk, groter of kleiner zijn.
void	Methoden of functies kunnen met <code>return</code> één waarde teruggeven. Als je dit niet wilt doen, dan moet je voor de naam van de methode het woord <code>void</code> plaatsen. Op deze plaats staat normaal gezien het type van de waarde die de methode teruggeeft.
zijeffect	Een methode wijzigt tijdens zijn uitvoering een variabele die buiten de methode is gedeclareerd. In Java kan dit alleen maar een objectvariabele zijn. Een zijeffect betekent dus dat na het uitvoeren van een methode er een objectvariabele gewijzigd is. Het probleem van zijeffecten is soms dat ze niet zo goed zichtbaar zijn voor lezers die het programma niet zelf geschreven zijn. Daarom moet je zijeffecten goed documenteren.