HO
GENT

H6 Methodes - next level

Table of Contents

1. Doelstellingen

Na het bestuderen van dit hoofdstuk ben je in staat

- een **methode aan te roepen** in Java
- een **methode te implementeren** in Java
- het datatype **array** in een methode te gebruiken

2. Inleiding

In dit hoofdstuk maken we kennis met *methodes*. We gaan eigen methodes schrijven, maar ook gebruik maken van bestaande methodes. We leren het verschil tussen *static* en *non-static* methodes. Daarnaast gaan we ook informatie doorgeven tussen methodes onderling. Tenslotte bekijken we ook nog even wat een *recursieve methode* is en we eindigen dit hoofdstuk met enkele extra's.

Een goed ontworpen programma maakt als volgt gebruik van methodes:

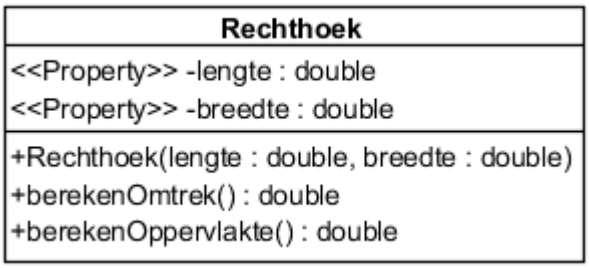
- De methodes zijn "klein", waarbij "klein" betekent dat ze weinig regels code bevatten
- Elke methode heeft één specifieke taak
- De naam van de methode benoemt die ene taak
- Methodes zijn goed herbruikbaar

3. Static en non-static methodes

3.1. non-static

We gebruiken al sinds H3 methodes om met de domeinklasse te communiceren. We gebruiken daarvoor een **non-static methode**. Dat betekent dat we in de applicatieklasse eerst een object moeten construeren om deze methodes aan te moeten roepen.

Voorbeeld:



We bekijken enkele methodes uit deze klasse:

- de constructor

```
    public Rechthoek(double lengte, double breedte)
    {
        setLengte(lengte);
        setBreedte(breedte);
    }
```

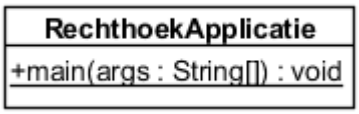
- de getters

```
    public double getLengte()    { return lengte; }
    public double getBreedte()   { return breedte;  }
```

- de methodes berekenOmtrek en berekenOppervlakte

```
    public double berekenOmtrek()       { return 2*(lengte + breedte); }
    public double berekenOppervlakte()  { return lengte * breedte; }
```

Als we deze methodes uit de klasse Rechthoek willen gebruiken, dan maken we eerst een object van de Rechthoekklasse aan en dan roepen we de methode aan met behulp van de puntnotatie.

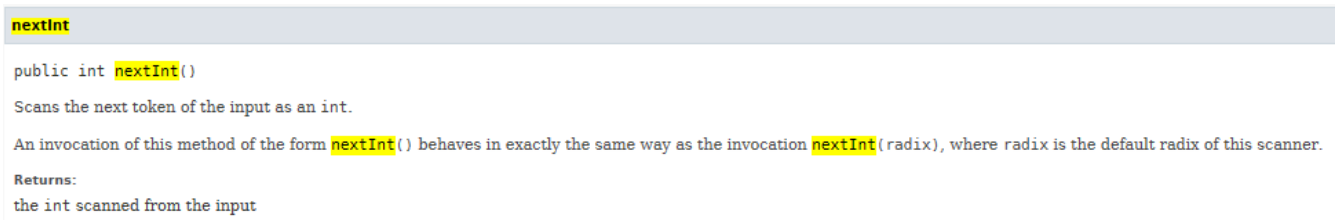


```
package cui;

import domein.Rechthoek;

public class RechthoekApplicatie
{
     public static void main(String[] args)
    {
          Rechthoek rechthoek1 = new Rechthoek(2,4.5);
          Rechthoek rechthoek2 = new Rechthoek(3.5,6.5);

          System.out.println("De eerste rechthoek:");
          System.out.printf("Met als lengte %.2f%n", rechthoek1.getLengte());
          System.out.printf("Met als breedte %.2f%n", rechthoek1.getBreedte());
          System.out.printf("Met als omtrek %.2f%n", rechthoek1.berekenOmtrek());
          System.out.printf("Met als oppervlakte %.2f%n", rechthoek1
.berekenOppervlakte());
          System.out.println();
          System.out.println("De tweede rechthoek:");
          System.out.printf("Met als lengte %.2f%n", rechthoek2.getLengte());
          System.out.printf("Met als breedte %.2f%n", rechthoek2.getBreedte());
          System.out.printf("Met als omtrek %.2f%n", rechthoek2.berekenOmtrek());
          System.out.printf("Met als oppervlakte %.2f%n", rechthoek2
.berekenOppervlakte());
    }
}
```

Je kan dit vergelijken met een baas die aan zijn werknemer een opdracht geeft. Het oproepen van een methode komt dan overeen met het aanspreken van een werknemer. Deze vervult zijn taak en geeft (eventueel) informatie terug (bv. de berekende omtrek of oppervlakte). De baas hoeft niets te weten over hoe de werknemer de taak vervuld heeft.

Een ander voorbeeld van een **non-static methode** die we reeds gebruikt hebben, zijn de methodes die we bij Scanner-objecten hebben gebruikt. Deze methodes hebben we niet zelf geschreven, maar vinden we terug in de API (https://docs.oracle.com/en/java/javase/14/docs/api/index.html).

Bijvoorbeeld voor de methode nextInt():



nextInt

public int nextInt()

Scans the next token of the input as an int.

An invocation of this method of the form nextInt() behaves in exactly the same way as the invocation nextInt(radix), where radix is the default radix of this scanner.

Returns:
the int scanned from the input

Deze methode roep je dus ook aan via een object van de klasse waartoe de methode behoort (m.a.w. een Scanner-object). Dat gaat zo:

```
Scanner input = new Scanner(System.in);
int x = input.nextInt();     // lees een geheel getal x van het toetsenbord
```

3.2. static

Daarnaast zijn er ook **static methodes**. Deze voeren een taak uit, onafhankelijk van de inhoud van om het even welk object van de klasse. Een dergelijke methode noemen we dan ook vaak een **klassemethode**.

Een voorbeeld van een **static methode** dat we al kennen, is de main-methode. Deze moet wel static zijn aangezien die bij de start van het programma, wanneer er nog geen objecten zijn, wordt aangeroepen. Door main static te declareren, kan de JVM deze methode activeren zonder dat er al een object van de klasse moet bestaan.

In de console start je je Java-programma op met `java Klassenaam [argument1 argument2]`. De JVM laadt dan de klasse, gespecifieerd door Klassenaam en gebruikt deze naam om de main-methode aan te roepen.

In de klasse Math komen static attributen en methodes voor. We bekijken nu ook even deze klasse als voorbeeld. De klasse voorziet de gebruikelijke mathematische bewerkingen.

Voorbeeld: de vierkantswortel berekenen van een (komma)getal

```
double x = Math.sqrt(900.0);
```

De methode sqrt wordt aangeroepen via de naam van de klasse, niet via een object van de klasse. Dit komt omdat de methode **static** is, zoals alle methodes in de klasse Math.

een static methode roep je aan via Klassenaam.methodenaam(argumenten)

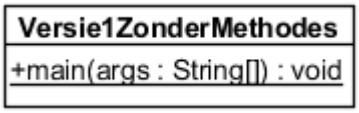
3.3. argumenten van een methode

De argumenten van een (al dan niet static) methode volgen na de methodenaam, tussen ronde haakjes en gescheiden door een komma. Deze argumenten kunnen

- constanten zijn: vb. `double x = Math.sqrt(50);`
- variabelen zijn: vb. `double y = 14, x = Math.sqrt(y);`
- expressies zijn: vb. `double z = 70.5; System.out.println(Math.sqrt(z+7));`

3.4. uitgewerkt voorbeeld

3.4.1. zonder methodes




```
package cui;

import java.util.Scanner;

public class Versie1ZonderMethodes
{
    public static void main(String args[])
    {
        Scanner scan = new Scanner(System.in);
        int getal1, getal2;

①
        do
        {
            System.out.print("Geef een getal verschillend van 0 in: ");
            getal1 = scan.nextInt();
        } while (getal1 == 0);

②
        do
        {
            System.out.print("Geef een getal verschillend van 5 in: ");
            getal2 = scan.nextInt();
        } while (getal2 == 5);

        System.out.printf("De ingegeven getallen zijn %d en %d%n", getal1, getal2);
    }
}
```

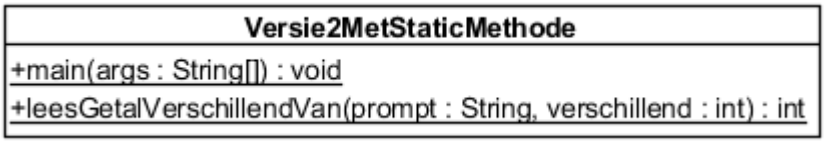
① in deze lus wordt een getal verschillend van 0 ingelezen

② in deze lus wordt een getal verschillend van 5 ingelezen

Beide do-whilelussen doen eigenlijk hetzelfde werk: je gaat een getal inlezen dat een bepaalde waarde niet mag hebben. De eerste keer is die waarde 0 en de tweede keer is die waarde 5 in het voorbeeld, maar het principe is hetzelfde. Wanneer de gebruikersinvoer niet juist is, wordt om nieuwe invoer gevraagd en dit tot de invoer aan de wensen voldoet.

Aangezien dit dubbele code is, gaan we hiervoor een methode gebruiken. Op die manier hoeven we de code maar één keer te schrijven en kunnen we ze daarna hergebruiken.

3.4.2. met een static methode




```
package cui;

import java.util.Scanner;

public class Versie2MetStaticMethode
{
    public static void main(String args[])
    {
        int getal1, getal2;

        getal1 = leesGetalVerschillendVan("Geef een getal verschillend van 0: ", 0);
        getal2 = leesGetalVerschillendVan("Geef een getal verschillend van 5: ", 5);
        System.out.printf("De ingegeven getallen zijn %d en %d%n", getal1, getal2);
    } // main

    public static int leesGetalVerschillendVan(String prompt, int verschillend)
    {
        Scanner scan = new Scanner(System.in);
        int getal2;
        do
        {
            System.out.print(prompt);
            getal2 = scan.nextInt();
        } while (getal2 == verschillend);
        return getal2;
    }

}
```

Aangezien een programma altijd vanuit de main gestart wordt, heb je geen object ter beschikking en begin je dus met een static methode. Als je een methode uit dezelfde klasse wil aanroepen zonder dat je een object hebt, moet die methode ook weer static zijn.

De methode wordt hier 2 keer aangeroepen:

```
getal1 = leesGetalVerschillendVan("Geef een getal verschillend van 0: ", 0);
```

EN

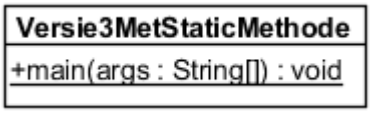
```
getal2 = leesGetalVerschillendVan("Geef een getal verschillend van 5: ", 5);
```

Telkens gebruik je hiervoor dus de naam van de methode en geef je een waarde door voor de

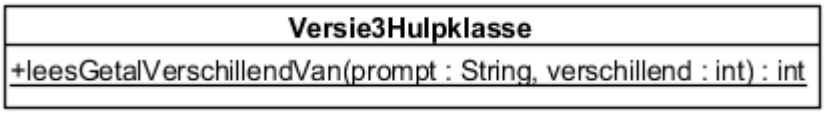
verschillende parameters. De terugkeerwaarde (een int) vang je op door het resultaat van de methode toe te kennen aan een variabele van het gewenste type. In het voorbeeld gebruik je hiervoor respectievelijk getal1 en getal2.

Dankzij de parameters is de methode vlot herbruikbaar. Zolang er maar 1 integer waarde is die het ingelezen getallen niet mag hebben, kan je deze methode gebruiken.

3.4.3. met een static methode in een hulpklasse




```
package cui;

public class Versie3MetStaticMethode
{
    public static void main(String args[])
    {
        int getal1, getal2;

        getal1 = Versie3Hulpklasse.leesGetalVerschillendVan("Geef een getal
verschillend van 0: ", 0);
        getal2 = Versie3Hulpklasse.leesGetalVerschillendVan("Geef een getal
verschillend van 5: ", 5);
        System.out.printf("De ingegeven getallen zijn %d en %d%n", getal1, getal2);
    } // main
}
```



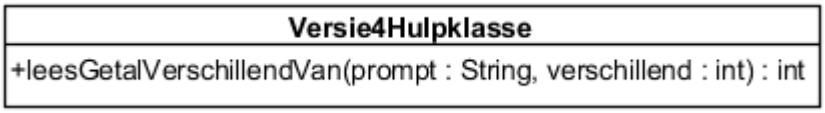
```
package cui;

import java.util.Scanner;

public class Versie3Hulpklasse
{
    public static int leesGetalVerschillendVan(String prompt, int verschillend)
    {
        Scanner scan = new Scanner(System.in);
        int getal2;
        do
        {
            System.out.print(prompt);
            getal2 = scan.nextInt();
        } while (getal2 == verschillend);
        return getal2;
    }
}
```

Je kan ook de static methode in een aparte klasse zetten. In dat geval kan je de methode nog steeds aanroepen zonder object maar moet je bij de aanroep wel de naam van de klasse voor de methodenaam zetten. Dus zo:

```
getal1 = Versie3Hulpklasse.leesGetalVerschillendVan("Geef een getal verschillend van
0: ", 0);
```

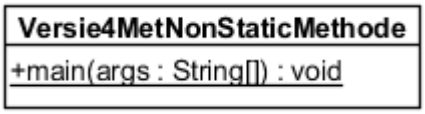
3.4.4. met een non-static methode in een hulpklasse

Aangezien de hulpklasse geen andere methodes bevat, is er eigenlijk geen enkele reden om de methode daar static te maken. Je kan dus ook het woordje `static` gewoon weglaten, zodat het een non-static methode wordt.



```
package cui;

import java.util.Scanner;

public class Versie4Hulpklasse
{
    public int leesGetalVerschillendVan(String prompt, int verschillend)
    {
        Scanner scan = new Scanner(System.in);
        int getal2;
        do
        {
            System.out.print(prompt);
            getal2 = scan.nextInt();
        } while (getal2 == verschillend);
        return getal2;
    }
}
```

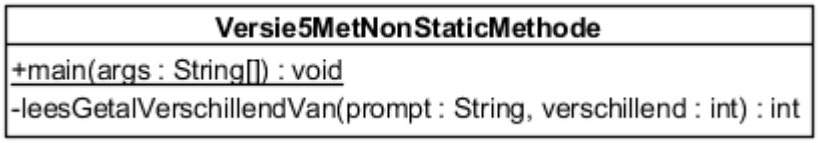
In dat geval moet je de methode in de andere klasse vanuit de main natuurlijk wel aanroepen zoals je dat gewoon bent om te doen als je vanuit de cui een methode uit een domeinklasse aanroept. Met andere woorden: je moet eerst een object aanmaken van de klasse waarin de methode staat vooraleer je de methode kan aanroepen. Hier is dat het object `hulp` van de klasse `Versie4Hulpklasse`.




```
package cui;

public class Versie4MetNonStaticMethode
{
    public static void main(String args[])
    {
        int getal1, getal2;

        Versie4Hulpklasse hulp = new Versie4Hulpklasse();

        getal1 = hulp.leesGetalVerschillendVan("Geef een getal verschillend van 0: ",
0);
        getal2 = hulp.leesGetalVerschillendVan("Geef een getal verschillend van 5: ",
5);
        System.out.printf("De ingegeven getallen zijn %d en %d%n", getal1, getal2);
    } // main
}
```

3.4.5. met een non-static methode (zonder hulpklasse)

Als je nu echter de (non-static) methode die in de hulpklasse staat, terug wil zetten in de oorspronkelijke klasse, dan kan dat natuurlijk ook! In dat geval zal je de methode ook weer moeten aanroepen met een object (hier ook `hulp` genaamd), maar dit keer gaat het om een object van de klasse zelf (klasse `Versie5MetNonStaticMethode` dus).




```
package cui;

import java.util.Scanner;

public class Versie5MetNonStaticMethode
{
    public static void main(String args[])
    {
        int getal1, getal2;

        Versie5MetNonStaticMethode hulp = new Versie5MetNonStaticMethode();

        getal1 = hulp.leesGetalVerschillendVan("Geef een getal verschillend van 0: ",
0);
        getal2 = hulp.leesGetalVerschillendVan("Geef een getal verschillend van 5: ",
5);
        System.out.printf("De ingegeven getallen zijn %d en %d%n", getal1, getal2);
    } // main

    private int leesGetalVerschillendVan(String prompt, int verschillend)
    {
        Scanner scan = new Scanner(System.in);
        int getal2;
        do
        {
            System.out.print(prompt);
            getal2 = scan.nextInt();
        } while (getal2 == verschillend);
        return getal2;
    }
}
```

3.4.6. vanuit de ene non-static methode een andere methode aanroepen

Wat hier nog niet aan bod kwam, maar natuurlijk wel ook mogelijk is, is het aanroepen van een methode vanuit een andere methode waarbij geen van beide methodes static is.

Een voorbeeld hiervan is de aanroep van een setter vanuit de constructor. In de domeinklasse

Rechthoek vind je hiervoor volgende code terug:

```
    public Rechthoek(double lengte, double breedte)
    {
        setLengte(lengte);
        setBreedte(breedte);
    }
```

```
    private void setLengte(double lengte)
    {
        if (lengte < 1) ①
            throw new IllegalArgumentException("Lengte voldoet niet"); ②
        this.lengte = lengte; ③
    }
    private void setBreedte(double breedte)
    {
        if (breedte < 1) ①
            throw new IllegalArgumentException("Breedte voldoet niet"); ②
        this.breedte = breedte; ③
    }
```

Zoals je in dit voorbeeld ziet, hoef je voor de aanroep van de non-static methode in dit geval geen object aan te maken. Dit komt omdat je al in de context van een object zit: je hebt namelijk binnen de domeinklasse het object `this` al ter beschikking. De aanroep `setLengte(lengte);` mag je dan ook vervangen door `this.setLengte(lengte);` en idem voor `setBreedte(breedte);` die mag vervangen worden door `this.setBreedte(breedte);`

4. Gebruik van methodes uit een bibliotheekklasse (klasse Math)

De klasse Math bevat 2 static attributen, nl. Math.PI (= 3.14159) en Math.E (= 2.7182818) Ze zijn **public, final en static** gedeclareerd in de klasse Math

- public: voor iedereen toegankelijk
- final: constant, de waarde kan niet gewijzigd worden
- static: voor alle objecten dezelfde waarde, te gebruiken via de klassenaam

We noemen static attributen ook **klassevariabelen** omdat ze gedeeld worden door alle objecten van de klasse. De waarde van dit attribuut is dus dezelfde voor elk object van deze klasse.

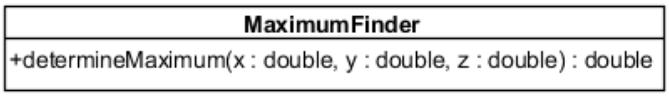
Enkele veelgebruikte methodes uit de Math-klasse:

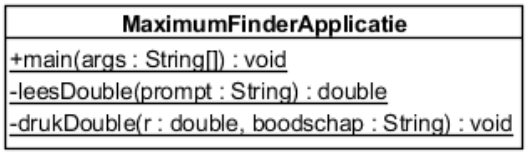
Methode	Omschrijving	Voorbeeld

`abs( x )`	Absolute waarde van *x*
	(eveneens versies voor **`float`**, **`int`** en **`long`** waarden)
		`abs( 23.7 )` is `23.7`
		`abs( 0.0 )` is `0.0`
		`abs( -23.7 )` is `23.7`

`ceil( x )`	Rondt *x* af naar de kleinste integer die niet kleiner is dan *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	Trigonometrische cosinus van *x* (*x* is in radialen)	`cos( 0.0 )` is `1.0`
`exp( x )`	Exponentiele methode *ex*	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`
`floor( x )`	Rondt *x* af naar de grootste integer niet groter dan *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`log( x )`	natuurlijk logarithme van *x* (basis *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`
`max( x, y )`	Grootste waarde van *x* en *y* (eveneens versies voor **`float`**, **`int`** en **`long`** waarden)	`max( 2.3, 12.7 )` is `12.7` `max( -2.3, -12.7 )` is `-2.3`
`min( x, y )`	Kleinste waarde van *x* en *y* (eveneens versies voor **`float`**, **`int`** en **`long`** waarden)	`min( 2.3, 12.7 )` is `2.3` `min( -2.3, -12.7 )` is `-12.7`
`pow( x, y )`	*x* verheven tot de macht *y* (x*y*)	`pow( 2.0, 7.0 )` is `128.0` `pow( 9.0, .5 )` is `3.0`
`sin( x )`	Trigonometrische cosinus van *x* (*x* is in radialen)	`sin( 0.0 )` is `0.0`
`sqrt( x )`	Vierkantswortel uit *x*	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`tan( x )`	Trigonometrische tangens van *x* (*x* is in radialen)	`tan( 0.0 )` is `0.0`

In het volgende voorbeeld gaan we een applicatie maken die het maximum van 3 kommagetallen bepaalt. In versie 1 schrijven we een methode determineMaximum die het maximum bepaalt aan de hand van 2 enkelvoudige selecties. In versie 2 maken we 2 keer gebruik van de voorgedefinieerde methode Math.max()





Versie 1:

```
package domein;

public class MaximumFinder
{
    public double determineMaximum(double x, double y, double z)
    {
        // veronderstel dat x de grootste waarde bevat om te starten
        double maximumWaarde = x;

        // ga na of y groter is dan de maximumWaarde
        if (y > maximumWaarde)
        {
            maximumWaarde = y;
        }

        // ga na of z groter is dan de maximumWaarde
        if (z > maximumWaarde)
        {
            maximumWaarde = z;
        }

        return maximumWaarde;
    }  // einde methode determineMaximum
}
```

Versie 2:

```
public double determineMaximum( double x, double y, double z )
{
    return Math.max( x, Math.max( y, z ) );
} // einde methode determineMaximum
```

En de bijhorende applicatie:

```
package cui;

import domein.MaximumFinder;
import java.util.Scanner;

public class MaximumFinderApplicatie
{
    public static void main(String[] args)
    {
        // maak een object van de klasse MaximumFinder aan de hand van de
defaultconstructor
        MaximumFinder maxF = new MaximumFinder();
        // vraag gebruikersinput aan de hand van een leesmethode
        double number1 = leesDouble("Geef eerste double in :");
        double number2 = leesDouble("Geef tweede double in :");
        double number3 = leesDouble("Geef derde double in :");

        // methode-aanroep maximum om grootste waarde te bepalen
        double max = maxF.determineMaximum(number1, number2, number3);

        // toon resultaat aan de hand van de drukmethode
        drukDouble(max, "Het maximum is ");
    } // einde methode main

    // de leesmethode om één double in te lezen
    private static double leesDouble(String prompt)
    {
        // declareer een double om de waarde in te stockeren
        double input;

        // de vraagstelling
        System.out.print(prompt);

        // object van de klasse Scanner maken
        Scanner inp = new Scanner(System.in);
        // input lezen
        input = inp.nextDouble();

        // de waarde teruggeven aan de caller
        return input;
    } // einde methode leesDouble

    // de drukmethode om één double te tonen
    private static void drukDouble(double r, String boodschap)
    {
        System.out.printf("%s % .2f", boodschap, r);
    } // einde methode drukDouble
}
```

In beide gevallen heeft de zelfgedefinieerde methode determineMaximum volgende

eigenschappen:

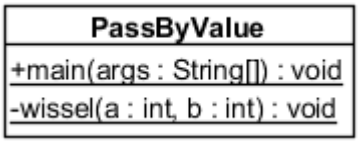
- Terugkeerwaarde: double
- Methodenaam: determineMaximum
- Parameterlijst: 3 doubles x, y en z
- Returnwaarde: maximum van de 3 doubles

5. Parameters doorgeven in een methode

5.1. Pass by value

Voor primitieve datatypes gebruikt Java **pass by value**. Dit wil zeggen dat de waarde van de variabele doorgegeven wordt aan de methode die aangeroepen wordt en deze dus niet gewijzigd kan worden.

Voorbeeld:



```java
package cui;

public class PassByValue
{
   public static void main(String[] args)
   {
      int a = 30;
      int b = 45;
      System.out.println("Voor de wissel, in main: a = " + a + " en b = " + b);
      // methode wissel aanroepen
      wissel(a, b);
      System.out.println();
      System.out.println("En toch zijn ze niet echt omgewisseld...:");
      System.out.println("Na de wissel, in main: a = " + a + " en b is " + b);
   }

   private static void wissel(int a, int b)
   {
      System.out.println("Voor de wissel, in methode: a = " + a + " en b = " + b);
      // Wissel de 2 waarden
      int c = a;
      a = b;
      b = c;
      System.out.println("Na de wissel, in methode: a = " + a + " en b = " + b);
   }
}
```

Dit geeft volgende output:

```
Voor de wissel, in main: a = 30 en b = 45
Voor de wissel, in methode: a = 30 en b = 45
Na de wissel, in methode: a = 45 en b = 30

En toch zijn ze niet echt omgewisseld...:
Na de wissel, in main: a = 30 en b is 45
```

Uit dit voorbeeld blijkt dus dat de `a` en `b` uit de `main` een "andere" `a` en `b` zijn dan die uit de methode `wissel`. Met andere woorden: er worden verschillende geheugenplaatsen gebruikt om deze variabelen bij te houden.

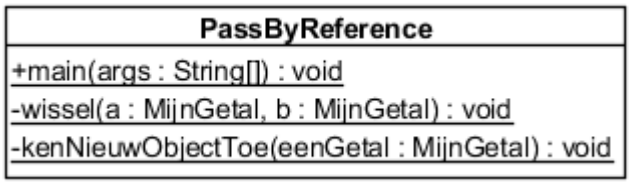
Bij de aanroep van de methode `wissel` worden de waarden van de `a` en `b` uit de `main` gekopieerd in de waarden van de parameters `a` en `b` uit de methode `wissel`. Binnen de methode `wissel` wordt enkel gebruik gemaakt van deze parameters en van een nieuwe lokale variabele `c`. In deze methode wordt de wissel dus wel degelijk uitgevoerd.

Eens we aan het einde van de methode `wissel` gekomen zijn, keren we terug naar de `main`-methode waar we opnieuw gebruik maken van (andere) lokale variabelen `a` en `b` die nog steeds de oorspronkelijke waarden (respectievelijk 30 en 45) hebben. De andere `a` en `b` verdwijnen uit het geheugen.

5.2. Pass by reference

Ook voor objecten gebruikt Java eigenlijk **pass by value**. De referentie wordt dus gekopieerd en kan door de aangeroepen methode worden gebruikt. Aangezien je met een kopie van de referentie werkt, zal het oorspronkelijk object dus wel gewijzigd kunnen worden. Het enige wat niet lukt en waarvoor je dus een echte **pass by reference** zou nodig hebben, is om een nieuw object te maken dat dan toegekend wordt aan de referentie.

Voorbeeld:




```
package domein;

public class MijnGetal
{
    private int getal;

    public MijnGetal(int getal)
    {
        this.getal = getal;
    }

    public int getGetal()
    {
        return getal;
    }

    public final void setGetal(int getal)
    {
        this.getal = getal;
    }
}
```

```
package cui;

import domein.MijnGetal;

public class PassByReference
{
   public static void main(String[] args)
   {
      MijnGetal a = new MijnGetal(30);
      MijnGetal b = new MijnGetal(45);
      System.out.println("Voor de wissel, in main: a bevat " + a.getGetal() + " en b
bevat " + b.getGetal());
      // methode wissel aanroepen
      wissel(a, b);
      System.out.println();
      System.out.println("En deze keer zijn ze echt omgewisseld...:");
      System.out.println("Na de wissel, in main: a bevat " + a.getGetal() + " en b
bevat " + b.getGetal());
      // methode kenNieuwObjectToe aanroepen
      System.out.println();
      System.out.println("Voor de toekenning, in main: het object bevat " + a.
getGetal());
      kenNieuwObjectToe(a);
      System.out.println();
      System.out.println("Maar de toekenning van een nieuw object lukt niet!");
      System.out.println("Na de toekenning, in main: het object bevat " + a.
getGetal());
   }

   private static void wissel(MijnGetal a, MijnGetal b)
   {
       System.out.println("Voor de wissel, in methode: a bevat " + a.getGetal() + " en
b bevat " + b.getGetal());
       // Wissel de 2 waarden
      MijnGetal c = new MijnGetal(a.getGetal());
      a.setGetal(b.getGetal());
      b.setGetal(c.getGetal());
      System.out.println("Na de wissel, in methode: a bevat " + a.getGetal() + " en b
bevat " + b.getGetal());
   }

   private static void kenNieuwObjectToe(MijnGetal eenGetal)
   {
       System.out.println("Voor de toekenning, in methode: het object bevat " +
eenGetal.getGetal());
       eenGetal = new MijnGetal(60);
       System.out.println("Na de toekenning, in methode: het object bevat " +
eenGetal.getGetal());
   }
}
```

Dit geeft volgende output:

```
Voor de wissel, in main: a bevat 30 en b bevat 45
Voor de wissel, in methode: a bevat 30 en b bevat 45
Na de wissel, in methode: a bevat 45 en b bevat 30

En deze keer zijn ze echt omgewisseld...:
Na de wissel, in main: a bevat 45 en b bevat 30

Voor de toekenning, in main: het object bevat 45
Voor de toekenning, in methode: het object bevat 45
Na de toekenning, in methode: het object bevat 60

Maar de toekenning van een nieuw object lukt niet!
Na de toekenning, in main: het object bevat 45
```

Ook hier zijn de `a` en `b` uit de `main` een "andere" `a` en `b` dan die uit de methode `wissel`, maar dit keer gaat het om referenties. Beide `a's verwijzen dus naar dezelfde geheugenlocatie en beide `b's ook.

Bij de aanroep van de methode `wissel` worden de referenties van de `a` en `b` uit de `main` gekopieerd in de referenties van de parameters `a` en `b` uit de methode `wissel`. Binnen de methode `wissel` wordt enkel gebruik gemaakt van deze parameters en van een nieuwe lokale variabele `c`. In deze methode wordt de wissel dus wel degelijk uitgevoerd.

Wanneer we aan het einde van de methode `wissel` komen en terugkeren naar de `main`-methode, verdwijnen er dus opnieuw 2 referenties, maar de `a` en `b` verwijzen naar dezelfde objecten die de intussen gewijzigde waarden (respectievelijk 45 en 30) bevatten voor het attribuut getal.

Bij de aanroep van de methode `kenNieuwObjectToe` wordt de referenties van de `a` uit de `main` gekopieerd in de referentie van de parameters `eenGetal` in de methode `kenNieuwObjectToe`. Dit laat ons toe om dit object te wijzigen, maar wanneer we een nieuw object aanmaken, dan krijgen we een nieuwe referentie terug die niet meer zal overeenstemmen met de referentie in de `a` uit de `main`.

Bij de terugkeer naar de `main`, is deze nieuwe referentie dan ook uit het geheugen verdwenen en zal dus niet de nieuwe waarde, waar `eenGetal` naar verweest, maar de oorspronkelijke waarde van `a` gebruikt worden om verder mee te werken.

6. Exceptions gebruiken in de setters

We kijken nog eens opnieuw naar de klasse Rechthoek. De methodes waarvan we de code nog niet geanalyseerd hebben, zijn de setters. Die zien er nu als volgt uit:

```
    private void setLengte(double lengte)
    {
        if (lengte < 1) ①
            throw new IllegalArgumentException("Lengte voldoet niet"); ②
        this.lengte = lengte; ③
    }
    private void setBreedte(double breedte)
    {
        if (breedte < 1) ①
            throw new IllegalArgumentException("Breedte voldoet niet"); ②
        this.breedte = breedte; ③
    }
```

① Hier wordt gecontroleerd of de parameterwaarde aan een voorwaarde voldoet. Dit is meestal een domeinregel.

② Indien de parameterwaarde NIET voldoet, wordt een Exception geworpen. Het gaat hier om een argument dat niet geldig is, dus de klasse **IllegalArgumentException** is hiervoor geschikt.
Je werpt een instantie van deze klasse, dus je hebt de constructor nodig om dit object te maken. Je kan gebruik maken van de defaultconstructor (zonder parameters) of je kan een parameter meegeven die een String is en de foutboodschap bevat.
In het voorbeeld werd gebruik gemaakt van de constructor met een parameter waarin aangegeven wordt welke afmeting niet voldoet.

③ Het gooien van een Exception zorgt ervoor dat de rest van de code binnen deze methode niet meer wordt uitgevoerd. Als we het laatste statement van de setter bereiken, betekent dit dus dat er geen uitzonderingssituatie is opgetreden en dat de parameterwaarde dus WEL aan de voorwaarde voldoet.
Er hoeft dus **geen else** geschreven te worden, aangezien je dit statement sowieso alleen maar kan bereiken indien er zich geen exception heeft voorgedaan.
Wat je in dit geval wel nog moet doen, is de parameterwaarde toekennen aan het bijhorend attribuut.

Merk op:

- We gaan dus geen defaultwaarde toekennen aan het attribuut als de parameterwaarde niet voldoet, aangezien we op die manier de gebruiker als het ware "misleiden" omdat het programma gewoon verder loopt, terwijl er eigenlijk een ongeldige waarde werd opgegeven, waardoor er dus eigenlijk ongemerkt met foute informatie verder zou kunnen gewerkt worden. In plaats daarvan krijgt de gebruiker een foutmelding te zien, waardoor hij weet dat er iets is misgelopen.
- In OOSD I gaan we alleen exceptions uit deze klasse werpen. Andere exceptionklasses, andere constructoren van de exceptionklasse en het opvangen van exceptions behandelen we in OOSD II. Dit betekent dat we op dit moment niks anders kunnen doen dan het programma beëindigen als er zich een exception voordoet.

7. Arrays en methodes

7.1. Inleiding

Ook array kans je doorgeven als parameter van een methode. Hiervoor gebruik je de arraynaam zonder de vierkante haakjes.

Voorbeeld:

Je declareert een array temperatuurPerUur als volgt:

```
    int temperatuurPerUur[] = new int[24];
```

Je kan nu bijvoorbeeld een methode aanroepen met deze array als parameter:

```
     wijzigArray(temperatuurPerUur);
```

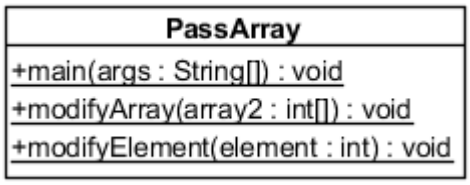
De array `temperatuurPerUur` wordt doorgegeven aan de methode `wijzigArray`.

Deze methode ziet er dan als volgt uit:

```
public void wijzigArray( int[] eenTemperatuurArray)
{
   ...
}
```

De array wordt, zoals een object, "**by reference**" doorgegeven, waardoor je dus een kopie van de referentie naar het juiste geheugenadres als parameter doorkrijgt in de methode wijzigArray. Hierdoor zullen de wijzigingen die je in deze methode in de array aanbrengt, permanent zijn. De performantie is hierdoor ook hoger dan als het om een "pass by value" zou gaan, vermits er geen data moet gekopieerd worden.

7.2. Voorbeeld 1-dimensionale array



```
package cui;

public class PassArray
{

    // main creëert array and roept de methoden modifyArray en modifyElement op
    public static void main(String[] args)
    {
        int[] array = {1, 2, 3, 4, 5}; ①

        System.out.printf("Effects of passing reference to entire array\nThe values of
the original array are:");

        // de oorspronkelijke elementen van de array weergeven ②
        for (int value : array)
        {
            System.out.printf("%5d", value);
        }
        modifyArray(array); ③
        System.out.printf("\n\nThe values of the modified array are:");

        // de gewijzigde elementen van de array worden weergegeven ④
        for (int value : array)
        {
            System.out.printf("%5d", value);
        }

        System.out.printf("\n\nEffects of passing array element value:\narray[3]
before modifyElement: %d%n", array[3]); ⑤
        modifyElement(array[3]); // een poging om array[3] te wijzigen ⑥
        System.out.printf("array[3] after modifyElement: %d%n", array[ 3]); ⑦
    } // end main

    // elk element van de array vermenigvuldigen met 2 ⑧
    public static void modifyArray(int[] array2)
    {
        for (int counter = 0; counter < array2.length; counter++)
        {
            array2[ counter] *= 2;
        }
    }

    // het argument met 2 vermenigvuldigen ⑨
    public static void modifyElement(int element)
    {
        element *= 2;
    }
}
```

① Initialisatie van de array. Deze is gedefinieerd als lokale variabele in de methode `main` en moet

dus als parameter doorgegeven worden aan andere methodes indien we wensen de array daar ook te gebruiken.

② We doorlopen de array en tonen elk element met een veldbreedte van 5. Resultaat:

```
    1    2    3    4    5
```

③ We roepen de methode modifyArray aan met als parameter de volledige array

④ We doorlopen de gewijzigde array en tonen elk element met een veldbreedte van 5. Resultaat:

```
    2    4    6    8   10
```

⑤ We tonen het element op index 3 van de array Resultaat: 8

⑥ We roepen de methode modifyElement aan met als parameter het element op index 3 van de array

⑦ We tonen het gewijzigde (?) element op index 3 van de array Resultaat: 8

⑧ Methode modifyArray. Het doorgeven van de array gebeurt via **pass by reference**. De elementen worden dus permanent gewijzigd.

⑨ Methode modifyElement. Het doorgeven van het arrayelement (een int) gebeurt via **pass by value**. Het element wordt dus slechts tijdelijk gewijzigd binnenin de methode en zal bij terugkeer naar de aanroepende methode (`main`) zijn waarde **niet** behouden.

```
Effects of passing reference to entire array
The values of the original array are:    1    2    3    4    5

The values of the modified array are:    2    4    6    8   10

Effects of passing array element value:
array[3] before modifyElement: 8
array[3] after modifyElement: 8
```

Onthou dus...

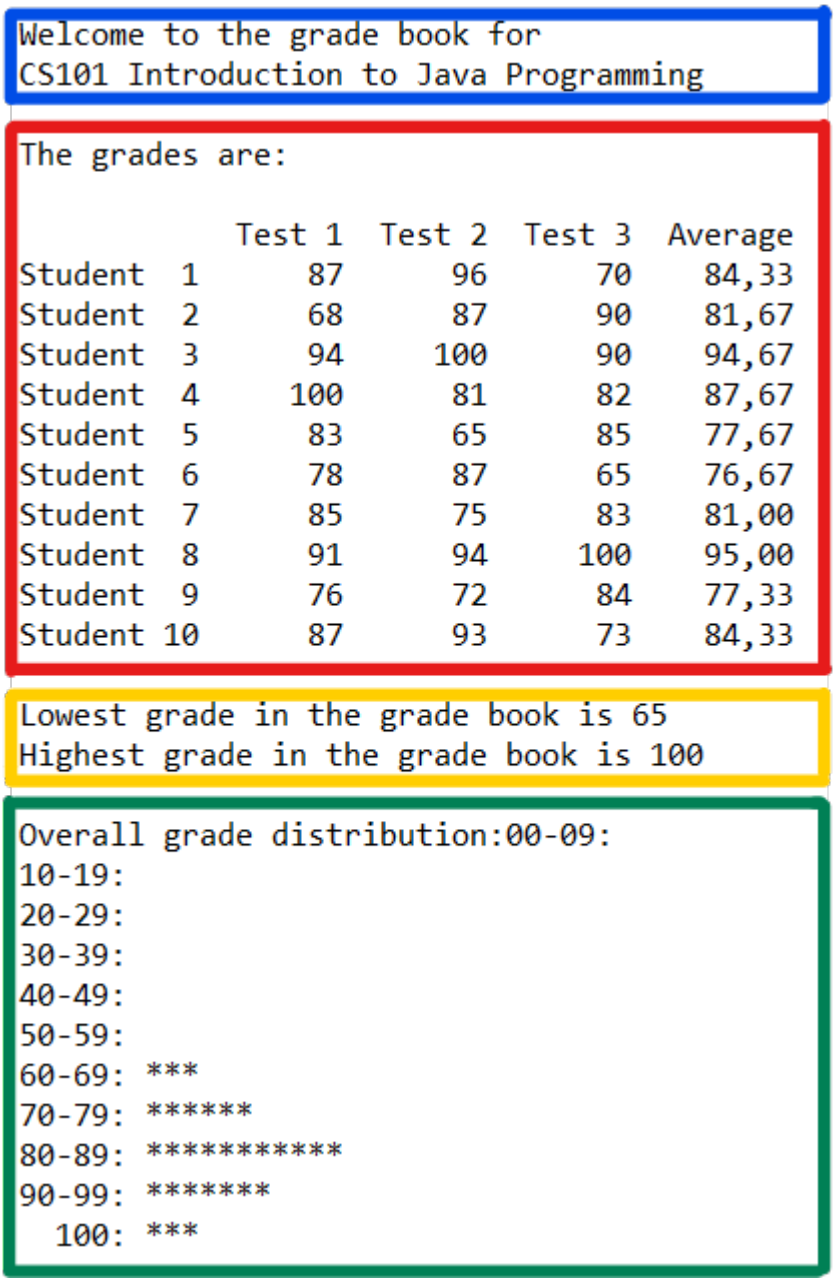
Pass by value: primitieve types

Pass by reference: objecten en arrays (zijn eigenlijk ook objecten)

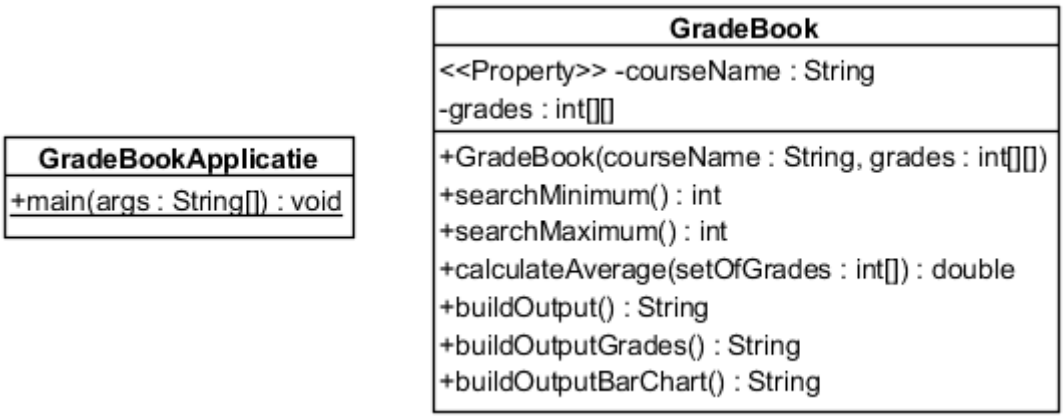
7.3. Voorbeeld 2-dimensionale array

In het puntenboek voor een bepaalde cursus worden de punten van 10 studenten voor 3 testen bijgehouden. We hebben dus een 10 x 3 array nodig.

Bedoeling is volgende output te creëren:



Op het scherm zie je achtereenvolgens:
* een verwelkoming met de naam van de cursus (blauw)
* alle cijfers per student met hun respectievelijk gemiddelde (rood)
* hoogste en laagste resultaat over alle studenten en alle testen heen (geel)
* verdeling van de punten in een staafdiagram (groen)




```
package cui;

import domein.GradeBook;

public class GradeBookApplicatie
{
    public static void main(String[] args)
    {
        // two-dimensional array of student grades ①
        int[][] gradesArray = {{87, 96, 70}, {68, 87, 90},
        {94, 100, 90}, {100, 81, 82}, {83, 65, 85}, {78, 87, 65},
        {85, 75, 83}, {91, 94, 100}, {76, 72, 84}, {87, 93, 73}};

②
        GradeBook myGradeBook = new GradeBook("CS101 Introduction to Java
Programming", gradesArray);
        System.out.printf("Welcome to the grade book for%n%s%n%n", myGradeBook
.getCourseName());

③
        System.out.printf(myGradeBook.buildOutput());
    } //einde methode main
}
```

① In de applicatie definiëren we eerst een tweedimensionale array met als rijen de punten van één van de 10 studenten. In elk van de 3 kolommen per rij vind je de punten voor één van de testen.

② Daarna maken we een nieuw puntenboek met als parameters de naam van de cursus en de array met punten. We printen een welkomstboodschap met de naam van de cursus erbij (blauwe kleur in de output).

③ Tenslotte bouwen we de rest van de uitvoer op (rode, gele en groene kleur) via een aanroep van de methode buildOutput.

```
    // output string opvullen
    public String buildOutput()
    {
        String output = buildOutputGrades(); ①

        output += String.format("%n%s %d%n%s %d%n%n",
                "Lowest grade in the grade book is", searchMinimum(),
                "Highest grade in the grade book is", searchMaximum());  ②

        output += buildOutputBarChart();  ③
        return output;
    } //einde methode buildOutput
```

De methode buildOutput bouwt alles op wat uitgeschreven wordt van het puntenboek:

① Overzicht alle punten + gemiddelde per student (rode kleur)

② Minimum en maximum (gele kleur)

③ Staafdiagram puntenverdeling van alle studenten (groene kleur)

De methode buildOutputGrades (rode kleur):

```
    // output the contents of the grades array
    public String buildOutputGrades()
    {
        String output = "The grades are:%n%n";
        output += "            "; // align column heads

①
        // create a column heading for each of the tests
        for (int test = 0; test < grades[0].length; test++)
        {
            output += String.format("Test %d  ", test + 1);
        }

        output += "Average%n"; // student average column heading

②
        // create rows/columns of text representing array grades
        for (int student = 0; student < grades.length; student++)
        {
            output += String.format("Student %2d", student + 1);

③
            for (int test : grades[student]) // output student's grades
            {
                output += String.format("%8d", test);
            }

④
            // call method getAverage to calculate student's average grade;
            // pass row of grades as the argument to getAverage
            double average = calculateAverage(grades[student]);
            output += String.format("%9.2f%n", average);
        }
        return output;
    }
```

① grades[0].length geeft het aantal kolommen van de eerste rij terug, met andere woorden het aantal testen dat de eerste student heeft afgelegd

② grades.length geeft het aantal rijen terug, dus het aantal studenten

③ alle punten van de student worden getoond op een veldbreedte van 8 karakters

④ het gemiddelde van de punten van deze student wordt berekend en getoond op een veldbreedte van 9 met 2 cijfers na de komma

De methode calculateAverage (onderdeel van de rode kleur):

```
①
    // de gemiddelde punten van een student bepalen
    public double calculateAverage(int setOfGrades[])
    {
        int total = 0;  // total initialiseren op nul

        // som van de punten van één student
        for (int grades : setOfGrades)
        {
            total += grades;
        }
②
        // geeft de gemiddelde punten terug
        return (double) total / setOfGrades.length;
    }
```

① De methode calculateAverage heeft 1 formeel argument: een één-dimensionale array, nl. de 3 punten van één student

② De som van de punten (total) wordt gedeeld door het aantal punten (setOfGrades.length). Het eindresultaat moet een double zijn. Er is dus een cast nodig!

De methode searchMinimum (deel 1 van gele kleur):

```
    // het slechtste examen zoeken
    public int searchMinimum()
    {
①
        // initialisatie: eerste element is het slechtste examen
        int lowGrade = grades[0][0];

②
        // de rijen van de array doorlopen
        for (int[] studentGrades : grades) // de kolommen per rij doorlopen
        {
            for (int grade : studentGrades)
            // Controleren of huidige punt van examen kleiner is dan "lowGrade"
            // Zoja, huidig punt toekennen aan "lowGrade"
            {
③
                if (grade < lowGrade)
                {
                    lowGrade = grade;
                }
            }
        }
        return lowGrade;  // geeft het slechtste examen terug
    }//einde methode minimum
```

met deze techniek kan je makkelijk de kleinste waarde uit een lijst halen

① Je initialiseert een variabele op het eerste element uit de lijst

② Je overloopt de volledige lijst. Hier gaat het om een 2-dimensionale array, dus je hebt een geneste (enhanced) for nodig.

③ Als je een waarde tegenkomt die kleiner is dan de huidige kleinste waarde, dan wordt dit de nieuwe kleinste waarde.

De methode searchMaximum (deel 2 van gele kleur):

```
    // het beste examen zoeken
    public int searchMaximum()
    {
        // initialisatie: eerste element is het beste examen
        int highGrade = grades[0][0];

        // de rijen van de array doorlopen
        for (int[] studentGrades : grades) // de kolommen per rij doorlopen
        {
            for (int grade : studentGrades)
            // Controleren als huidige punt van examen groter is dan highGrade.
            // Zoja, huidig punt toekennen aan highGrade.
            {
                if (grade > highGrade)
                {
                    highGrade = grade;
                }
            }
        }
        return highGrade; // geeft het beste examen terug
    }//einde methode maximum
```

je vindt dezelfde 3 stappen terug als bij de bepaling van het minimum

De methode buildOutputBarChart (groene kleur):

```
    public String buildOutputBarChart()
    {
        String output = "Overall grade distribution:";

        // stores frequency of grades in each range of 10 grades
        int[] frequency = new int[11];

        // for each grade in GradeBook, increment the appropriate frequency
        for (int[] studentGrades : grades)
        {
            for (int grade : studentGrades)
            {
                ++frequency[grade / 10];
            }
        }

        // for each grade frequency, print bar in chart
        for (int count = 0; count < frequency.length; count++)
        {
            // output bar label ("00-09: ", ..., "90-99: ", "100: ")
            if (count == 10)
            {
                output += String.format("%5d: ", 100);
            }
            else
            {
                output += String.format("%02d-%02d: ", count * 10, count * 10 + 9);
            }

            // print bar of asterisks
            for (int stars = 0; stars < frequency[count]; stars++)
            {
                output += "*";
            }
            output += "%n";
        }
        return output;
    }
```

Analoog met een van de voorbeelden uit H5.

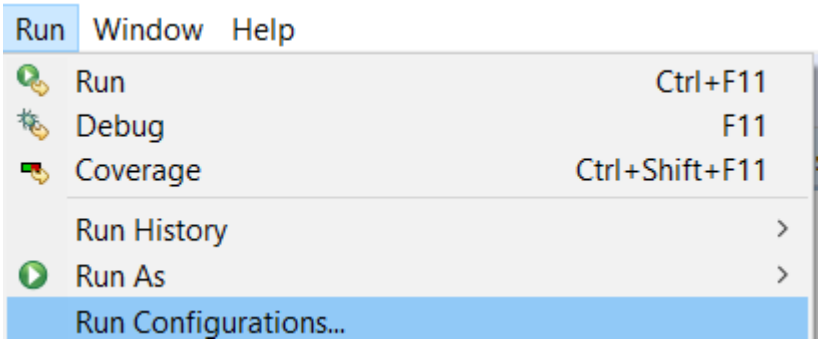
7.4. Programmaparameters

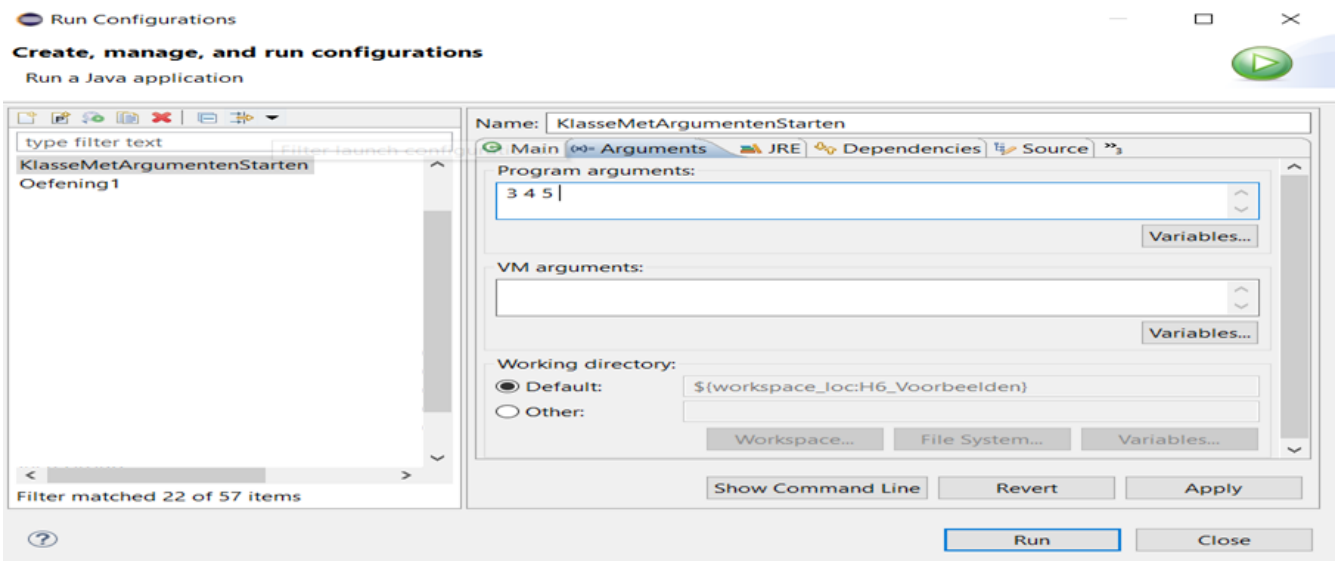
De methode main heeft 1 argument : een array van Strings.

```
public static void main (String[] args)
```

Je kan dus ook een array van String(s) aan de methode main doorgeven. In **Eclipse** doe je dit via het

menu **"Run (As) – Run Configurations..."**





```
package cui;

public class KlasseMetArgumentenStarten
{
    public static void main(String[] args)
    {
        // controleer of er 3 strings werden doorgegeven aan de methode main
        if (args.length != 3)
        {
            System.out.printf(
                    "Error: Please re-enter the entire command, including%n"
                    + "an array size, initial value and increment.%n");
        }
        else
        {
            // vraag het eerste element van de array args op
            int arrayLength = Integer.parseInt(args[ 0]);
            int array[] = new int[arrayLength]; // creatie van een array

            // Vraag het tweede en derde element van de array args op
            int initialValue = Integer.parseInt(args[ 1]);
            int increment = Integer.parseInt(args[ 2]);

            // bereken de waarde van elk element van de array array
            for (int counter = 0; counter < array.length; counter++)
            {
                array[ counter] = initialValue + increment * counter;
            }

            System.out.printf("%s%8s%n", "Index", "Value");

            // geef de index en de waarde van elk element weer
            for (int counter = 0; counter < array.length; counter++)
            {
                System.out.printf("%5d%8d%n", counter, array[ counter]);
            }
        } // end else
    }
}
```

Stel: je wil dit programma runnen en de array {"3","4","5"} doorgeven als parameter aan de methode main.

In het menu "Run (As) – Run Configurations..." vind je het tabblad **Arguments**. Hier kan je de programmaparameters doorgeven. Vervolgens klik je op "Run".

Het resultaat ziet er als volgt uit:

```
Index   Value
    0       4
    1       9
    2      14
```

Indien je geen (of te veel of te weinig) programmaparameters meegeeft of vergeet de parameters in te stellen, dan krijg je volgende foutmelding:

```
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

8. Recursie

(Zie hoofdstuk 18 in het handboek)

8.1. Wat is recursie?

Een recursieve methode

- roept zichzelf op (direct of indirect via een andere methode)
- kan enkel de **basis case(s)** oplossen, m.a.w. roepen we de methode aan met de basis case(s), dan geeft deze een resultaat terug.
- verdeelt een probleem in een basis case en een **eenvoudiger probleem** indien de methode wordt aangeroepen met een complexer probleem en gaat door met het eenvoudiger probleem verder te verdelen totdat het opgelost is.

Een recursieve methode bevat dus een recursieve oproep (**call**).

Het nieuwe, eenvoudiger probleem lijkt op het originele probleem, dus roept de methode zichzelf terug aan om het eenvoudiger probleem op te lossen. Dit is een recursieve doorloop (**step**).

De recursieve stap bevat normaal ook het keyword `return`, vermits het resultaat van deze aanroep moet gecombineerd worden met het resultaat van het gedeelte dat de methode wel kan oplossen om uiteindelijk een eindresultaat aan de originele caller te kunnen teruggeven!

8.2. Voorbeeld 1: faculteitsberekening

Je berekent de faculteit van het getal 5 als volgt:

5! = 5 * 4 * 3 * 2 * 1

5! = 5 * (4 * 3 * 2 * 1)

5! = 5 * 4!

De recursieve methode verdeelt het probleem in een **eenvoudiger probleem**:

5! = 5 * 4!

De recursieve methode verdeelt het probleem in een **eenvoudiger probleem**:

4! = 4 * 3!

De recursieve methode verdeelt het probleem in een **eenvoudiger probleem**:

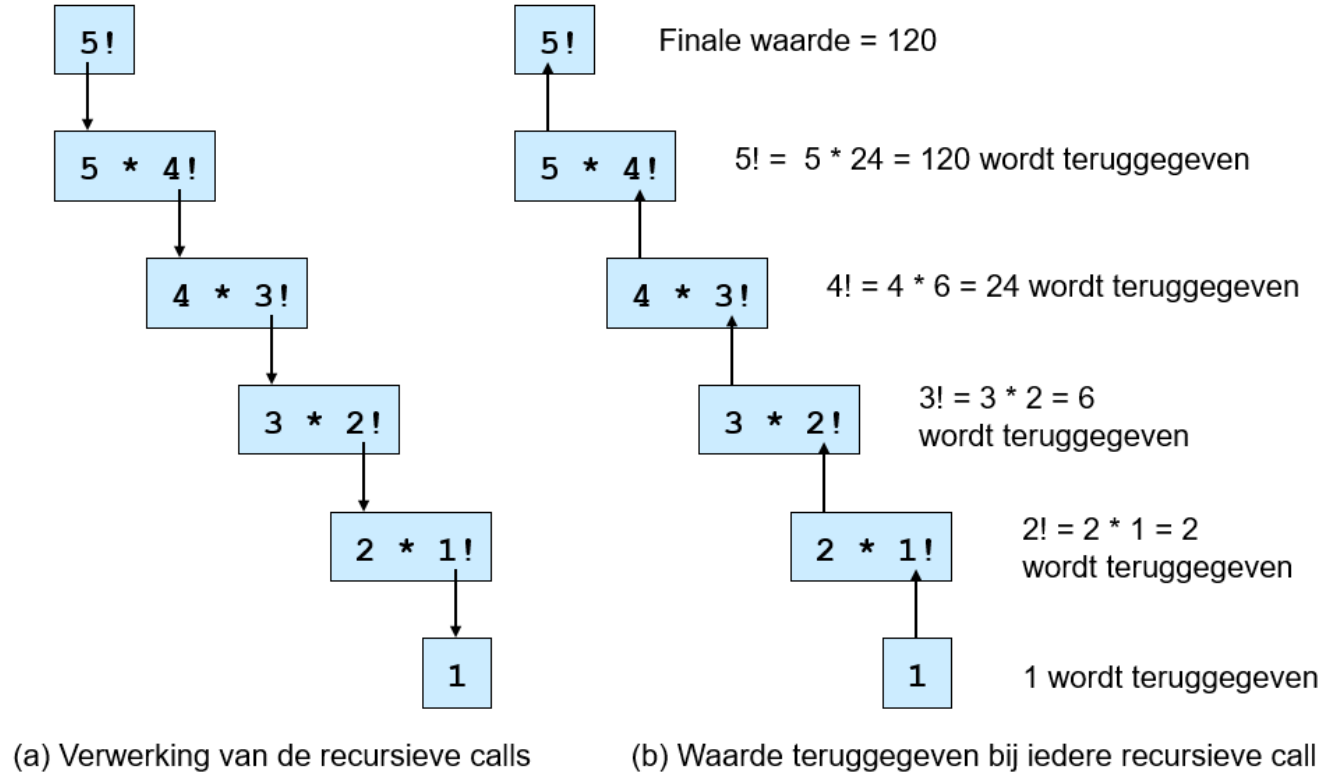
3! = 3 * 2!

De recursieve methode verdeelt het probleem in een **eenvoudiger probleem**:

2! = 2 * 1!

De recursieve methode verdeelt het probleem in een **basis case**:

1! = 1



Implementatie: berekening van de faculteit van de getallen 0 tot en met 10

```
package cui;

public class FactorialCalculator
{
    // calculate factorials of 0-10
    public static void main(String args[])
    {
        FactorialCalculator factorialCalculator = new FactorialCalculator();
        factorialCalculator.displayFactorials();
    } // end method main
    // output factorials for values 0-10

    public void displayFactorials()
    {
        // calculate the factorials of 0 through 10
        for (int counter = 0; counter <= 10; counter++)
        {
            System.out.printf("%d! = %d\n", counter, factorial(counter));
        }
    } // end method displayFactorials

    public long factorial(long number)
    {
①
        // basis case
        if (number <= 1)
        {
            return 1;
        }
②
        // recursive step
        else
        {
            return number * factorial(number - 1);
        }
    }  // einde methode factorial
}
```

De recursieve methode factorial verdeelt het probleem in: <.> het basisgeval: 0! = 1 en 1! = 1 <.> een eenvoudiger probleem: n! = n * (n-1)!

8.3. Voorbeeld 2: de Fibonacci reeks

In de Fibonnaci reeks is elk getal de som van de twee voorgaande getallen

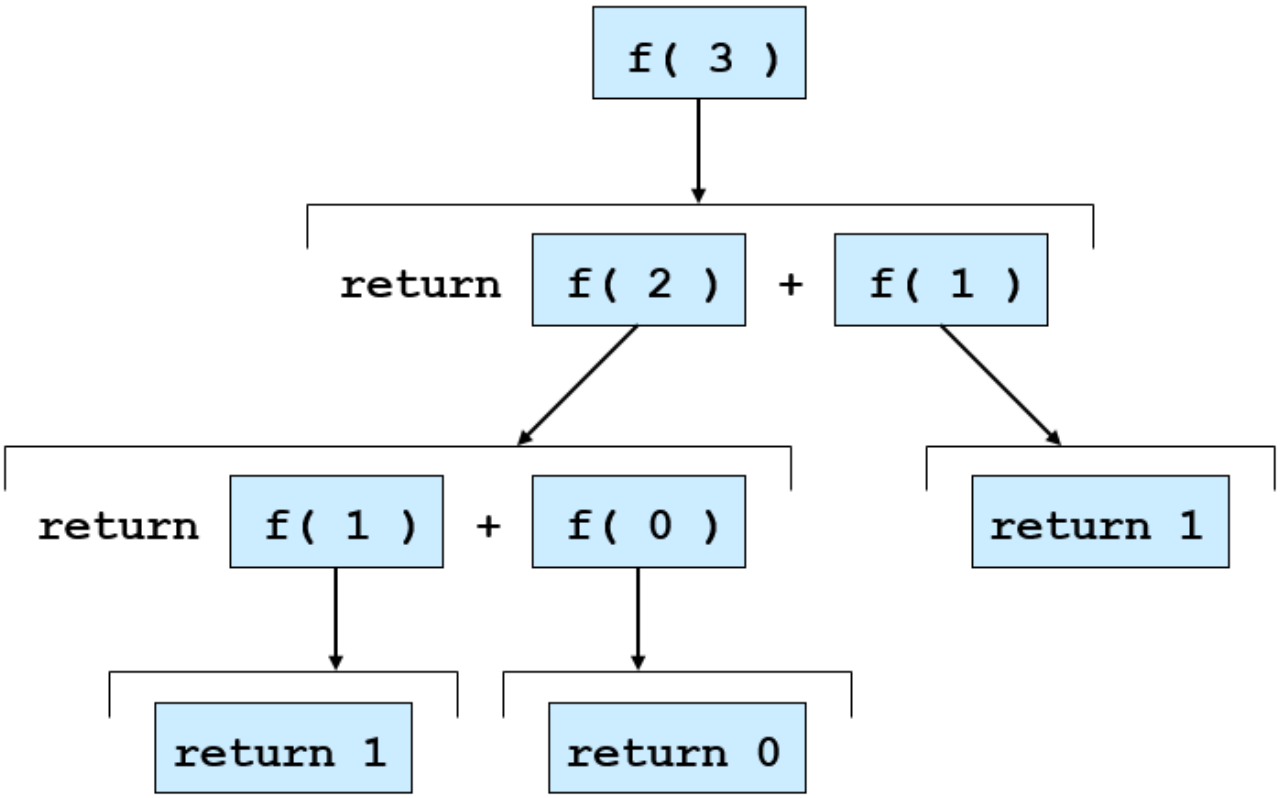
De reeks gaat dus als volgt: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

fibonacci(0) en fibonacci(1) zijn **basis cases**:

- fibonacci(0) = 0
- fibonacci(1) = 1

recursieve stap:

fibonacci(n) = fibonacci(n-1) + fibonacci(n–2)



Implementatie: berekening van de Fibonaccigetallen met nummer 0 tot en met 10

```
package cui;

public class FibonacciCalculator
{
    public static void main( String args[] )
    {
        FibonacciCalculator fibonacciCalculator = new FibonacciCalculator();
        fibonacciCalculator.displayFibonacci();
    } // end main

    // recursive declaration of method fibonacci
    public long fibonacci( long number )
    {
①
        // base cases
        if (( number == 0 ) || ( number == 1 ))
            return number;
②
        // recursion step
        else
            return fibonacci( number - 1 ) + fibonacci( number - 2 );
    } // end method fibonacci

    public void displayFibonacci()
    {
        for ( int counter = 0; counter <= 10; counter++ )
            System.out.printf("Fibonacci of %d is: %d\n", counter, fibonacci( counter
));
    } // end method displayFibonacci
}
```

De recursieve methode fibonacci verdeelt het probleem in:

① het basisgeval: fibonacci(0) = 0 en fibonacci(1) = 1

② een eenvoudiger probleem: fibonacci(n) = fibonacci(n-1) + fibonacci(n–2)

9. Een aantal extra's

9.1. Promotie en casting

Typeconversie (coercion) van argumenten betekent dat je argumenten forceert naar het gepaste type om door te geven aan de methode.

Voorbeeld:

`sqrt()` verwacht een double:

```
System.out.println(Math.sqrt(4));
```

wordt dus geïnterpreteerd als `System.out.println(Math.sqrt(4.));`

Dus uiteindelijk resultaart dat in `System.out.println(2.);`

Promotieregels

Promotieregels specifiëren hoe je bepaalde types kan converteren naar andere types zonder gegevensverlies. Het type van een argument kan steeds gepromoveerd worden naar een "hoger" type (zie onderstaande tabel).

Type	Toegelaten promoties
double	*geen*
float	double
long	float *of* double
int	long, float *of* double
char	int, long, float *of* double
short	int, long, float *of* double
byte	short, int, long, float *of* double
boolean	*geen* (**boolean** waarden zijn geen getallen in Java)

Deze regels worden eveneens toegepast bij een gemengde expressie!

Voorbeeld: `int a = 10; double b = 0; b += a;`

Om de methode `square(int y)` aan te roepen met een `double (x = 3.5)`, gebruiken we de volgende aanroep: `double res = square((int)x);`

9.2. Random getallen genereren

9.2.1. Klasse Math

Java kan willekeurige getallen (random-numbers) genereren via een methode uit de klasse `Math`: `Math.random()`

`double randomValue = Math.random();` waarbij `1.0 > randomValue >= 0.0``

Het zijn **pseudo-random** getallen, want ze worden bepaald door een complexe wiskundige berekening, die gebruik maakt van de actuele tijd.

Het interval van waarden dat geproduceerd wordt door deze methode, is meestal verschillend van het gewenste interval.

Voorbeelden:

- kop (0) of munt (1)
- gooien van een dobbelsteen (1, 2, ..., 6)

- bepalen van lottogetallen (1, 2, ..., 45)
- ...

Nemen we even het voorbeeld van de dobbelsteen:

- Om in plaats van een kommagetal uit het interval [0.0, 1.0[een waarde uit het interval [0.0,6.0[te verkrijgen, gebruiken we de volgende expressie:

```
Math.random() * 6
```

We brengen het interval van waarden op schaal en het cijfer **6** noemen we de **schaalfactor**!

Aangezien we gehele getallen nodig hebben in plaats van kommagetallen, gaan we daarna casten. Zo verkrijgen we een geheel getal uit het interval [0,6[of dus een waarde 0, 1, 2, 3, 4 of 5.

```
(int) (Math.random() * 6)
```

Door tenslotte **1** op te tellen (**shiftwaarde**) bij ons vorig resultaat, verkrijgen we het gewenste resultaat, namelijk een getal uit het interval [1,7[. De waarde van `dobbel` in onderstaande formule zal dus steeds 1, 2, 3, 4, 5 of 6 zijn.

```
int dobbel = (int) (Math.random() * 6) + 1;
```

Algemeen: `int n = a + (int) (Math.random() * b);`

Hierbij is a de shiftwaarde (= kleinste waarde in het interval) en b is de schaalfactor (= aantal mogelijke waarden).

9.2.2. Klasse Random

Er zijn in Java ook 2 klasses die toelaten om willekeurige getallen te bepalen: klasse **Random** uit de package java.util en klasse **SecureRandom** uit de package java.security.

Het grote verschil tussen de twee is dat de getallen die de klasse **SecureRandom** genereert minder voorspelbaar zijn doordat er een andere random generator gebruikt wordt.

Voor "huis, tuin en keukengebruik" kan je echter nog steeds perfect **Random** gebruiken, dat minder geheugen gebruikt. Gebruik echter zeker **SecureRandom** wanneer je niet wilt dat je random waardes makkelijk te kraken zijn, zoals bijvoorbeeld bij willekeurig gegenereerde wachtwoorden.

De methodes van de klasse Random worden overgeërfd door de klasse **SecureRandom** en zijn dus dezelfde.

9.2.3. Klasse SecureRandom

Tenslotte kan je in Java eveneens random getallen genereren via een object van de klasse

SecureRandom

```
SecureRandom randomNumbers = new SecureRandom ();
```

Dit object kan gebruikt worden om randomwaarden te genereren van het type boolean, byte, float, double, int en long. Een willekeurig getal dat op deze manier gegenereerd wordt, is niet-deterministisch (kan niet voorspeld worden).

Bijvoorbeeld:

`int randomValue = randomNumbers.nextInt();` genereert een randomwaarde uit het bereik van `int`, dus tussen –2.147.483.648 en 2.147.483.647

Nemen we opnieuw het voorbeeld van de dobbelsteen:

Om een geheel getal tussen 1 en 6 (beide inbegrepen) te verkrijgen, gebruiken we de volgende formule:

```
int randomValue = 1 + randomNumbers.nextInt(6);
```

De aanroep van de methode nextInt met de integer n als parameter, levert een intwaarde uit het interval [0,n-1] op. Door er 1 bij op te tellen, komen we dus uit op het gewenste interval [1,n].

Algemeen:

```
int n = a + randomNumbers.nextInt(b);
```

Hierbij is a opnieuw de shiftwaarde (= kleinste waarde in het interval) en b is de schaalfactor (= aantal mogelijke waarden).

Om een willekeurig getal uit de reeks 2, 5, 8, 11 en 14 te genereren, gebruiken we het volgende statement:

```
int randomValue = 2 + 3 * randomNumbers.nextInt(5);
```

Hier is de parameter van de methode nextInt (hier: 5) het aantal mogelijke waarden en de waarde die we erbij optellen is de minimumwaarde (hier: 2). De waarde (hier: 3) waarmee we vermenigvuldigen, is het verschil tussen de verschillende waarden!

De hierna volgende voorbeelden maken telkens gebruik van de klasse SecureRandom om een willekeurig getal te bepalen!

9.2.4. Voorbeeld 1: 20 dobbelsteenworpen

In de volgende applicatie worden 20 willekeurige getallen gegenereerd met een waarde tussen 1 en 6.

Deze applicatie maakt gebruik van een for-lus om 20 keer een random getal te bepalen.

```
package cui;

import java.security.SecureRandom;

public class RandomIntegers
{
    public static void main(String args[])
    {
        int value;
        SecureRandom randomNumbers = new SecureRandom();

        // lus 20 keer
        for (int counter = 1; counter <= 20; counter++) {
            // kies willekeurige integer tussen 1 en 6
            value = 1 + randomNumbers.nextInt(6);

            System.out.printf("%d ", value);  // value tonen

            // als counter deelbaar is door 5, voeg newline toe
            if (counter % 5 == 0)
            {
                System.out.println();
            }
        }
    }  // einde methode main
}
```

Mogelijke uitvoer:

```
5 5 5 2 3
5 1 1 4 1
6 3 6 1 3
5 2 2 2 3
```

OF

```
6 2 1 2 5
6 2 4 3 2
3 5 5 4 2
4 2 6 4 6
```

9.2.5. Voorbeeld 2: 6 miljoen dobbelsteenworpen

De volgende applicatie is een simulatie van 6 miljoen teerlingworpen om te bewijzen dat elk getal 1 tot en met 6 (bijna) evenveel kans heeft om voor te komen.

Ook hier wordt gebruik gemaakt van een for-lus om 6 miljoen keer te werpen. Een array met 7 elementen (index 0 wordt niet gebruikt) doet dienst als frequentietabel om bij te houden hoeveel

keer elk van de mogelijke worpen is voorgekomen. Als je dus bijvoorbeeld 4 gooit, dan wordt het element van de frequentietabel op index 4 met 1 verhoogd.

```
package cui;

import java.security.SecureRandom;

public class RollDie
{
    public static void main(String[] args)
    {
        SecureRandom randomNumbers = new SecureRandom(); //random nummer generator
        int frequency[] = new int[7];

        // werp de dobbelsteen 6 miljoen keer
        for (int roll = 1; roll <= 6000000; roll++)
        {
            ++frequency[ 1 + randomNumbers.nextInt(6)];
        }

        System.out.printf("%s%10s%n", "Face", "Frequency");
        // het resultaat van de 6 miljoen worpen weergeven
        for (int face = 1; face < frequency.length; face++)
        {
            System.out.printf("%4d%10d%n", face, frequency[ face]);
        }
    } // einde main
}
```

Mogelijke uitvoer:

```
Face Frequency
   1    999705
   2   1000014
   3   1000497
   4    999806
   5   1000929
   6    999049
```

OF

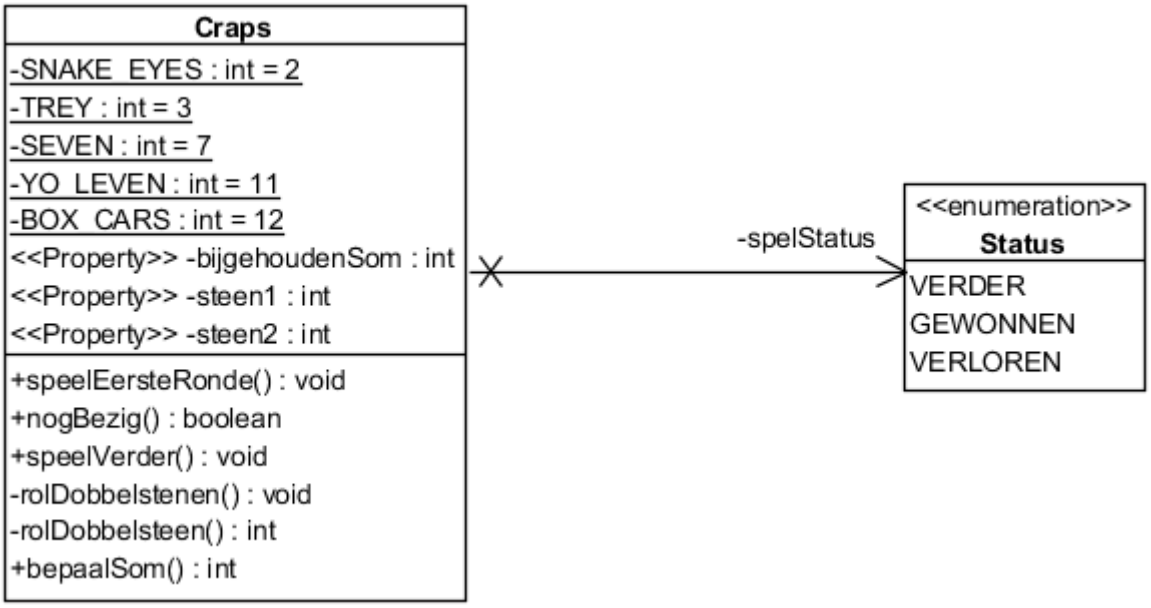
```
Face Frequency
   1    999679
   2    998799
   3   1000161
   4   1000500
   5    999582
   6   1001279
```

9.2.6. Voorbeeld 3: het Craps-spel

In dit laatste voorbeeld bekijken we een simulatie van een kansspel. Hiervoor gebruiken we het spel Craps waarvan de spelregels als volgt luiden:

- Gooi de dobbelstenen (2 stuks) een eerste keer
  - Als de som gelijk is aan 7 of 11 **wint** de speler
  - Als de som gelijk is aan 2, 3 of 12, **verliest** de speler
  - Bij elke andere som (4, 5, 6, 8, 9 of 10) wordt de **som bijgehouden**
- Blijf met de teerlingen gooien totdat…
  - de som overeenkomt met de eerder bijgehouden som → in dat geval **wint** de speler
  - de som 7 is → in dat geval **verliest** de speler

In deze applicatie wordt ook gebruik gemaakt van een eenvoudige Enumeration




```
package domein;

import java.security.SecureRandom;

public class Craps
{
    // constanten voor de belangrijkste dobbelsteenworpen
    private final static int SNAKE_EYES = 2;
    private final static int TREY = 3;
    private final static int SEVEN = 7;
    private final static int YO_LEVEN = 11;
    private final static int BOX_CARS = 12;

①
    // enumeratie voor de verschillende toestanden waarin het spel zich kan beginden
    private enum Status
```

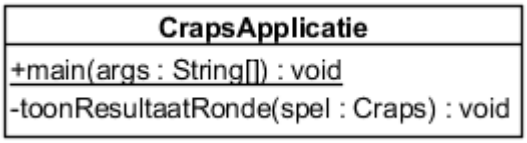
```
    {
        VERDER, GEWONNEN, VERLOREN
    };

    // attributen
    private int bijgehoudenSom; // bijhouden welk getal er opnieuw moet gegooid worden
    private Status spelStatus; // bijhouden in welke toestand het spel zich bevindt:
kan VERDER, GEWONNEN of VERLOREN bevatten
    private int steen1, steen2; // bijhouden wat er geworpen werd;

    public int getBijgehoudenSom() { return bijgehoudenSom; }
    public Status getSpelStatus() { return spelStatus; }
    public int getSteen1() { return steen1; }
    public int getSteen2() { return steen2; }

    // eerste worp
    public void speelEersteRonde()
    {
        rolDobbelstenen();
        // toestand waarin het spel zich bevindt bepalen op basis van de eerste worp
        int som = bepaalSom();
        switch (som)
        {
            // 7 of 11: geGEWONNENnen
            case SEVEN:
            case YO_LEVEN:
                spelStatus = Status.GEWONNEN;
                break;
            // 2, 3 of 12: verloren
            case SNAKE_EYES:
            case TREY:
            case BOX_CARS:
                spelStatus = Status.VERLOREN;
                break;
            // alle andere worpen: nog niet bepaald - point instellen en verder rollen
            default:
                spelStatus = Status.VERDER; // spel gaat verder
                bijgehoudenSom = som; // som van de dobbelstenen bijhouden
                break;
        }
    }

    public boolean nogBezig()
    {
        return spelStatus == Status.VERDER;
    }

    // als we na de eerste worp nog geen resultaat kennen
    public void speelVerder()
    {
        rolDobbelstenen(); // nieuwe worp
```

```
        // bepaal nieuwe speltoestand
        int nieuweSom = bepaalSom();
        if (nieuweSom == bijgehoudenSom) // geGEWONNENnen door bijgehouden worp
opnieuw te gooien
            spelStatus = Status.GEWONNEN;
        else if (nieuweSom == SEVEN) // verloren door 7 te rollen
            spelStatus = Status.VERLOREN;
    }

②
    private void rolDobbelstenen()
    {
        steen1 = rolDobbelsteen();
        steen2 = rolDobbelsteen();
    }

③
    // rol met een dobbelsteen en hou het resultaat van de worp bij
    private int rolDobbelsteen()
    {
        SecureRandom generator = new SecureRandom(); // maak een generator voor random
getallen
        return 1 + generator.nextInt(6); // bepaal een getal in het interval [1,6]
voor de worp
    }

    public int bepaalSom()
    {
        return steen1 + steen2;
    }
}
```

① Een enumeratie is een speciaal soort klasse, die gedefinieerd wordt aan de hand van het keyword enum (ipv class) en een typenaam (hier Status). In de eenvoudigste vorm bevat de body van een enumeratie een lijst van enumeratieconstanten in hoofdletters, gescheiden door een komma. Elk stelt een unieke waarde voor. Meer uitgebreide mogelijkheden van een enumeratie komen aan bod in de volgende paragraaf.

② De methode rolDobbelstenen roept tweemaal de methode rolDobbelsteen aan, zodat je 2 aparte dobbelsteenworpen krijgt. Op die manier kom je een getal uit het interval [2,12] uit als som, maar moest je in één keer een willekeurig getal uit [2,12] laten bepalen, dan zou bijvoorbeeld het getal 7 evenveel kans hebben om voor te komen als het getal 12. Nochtans kan je 7 bekomen door 1+6, 2+5, 3+4, 4+3, 5+2 en 6+1 te werpen en 12 alleen maar door 6+6 te werpen. Je hebt dus met andere woorden 6 keer meer kans om 7 te werpen dan om 12 te werpen.

③ De methode rolDobbelsteen gebruikt een SecureRandom object om een getal uit het interval [1,6] te bepalen en simuleert zo de worp van één dobbelsteen.




```
package cui;

import domein.Craps;

public class CrapsApplicatie
{
    public static void main(String args[])
    {
        CrapsApplicatie app = new CrapsApplicatie();
        Craps spel = new Craps();
        spel.speelEersteRonde();
        app.toonResultaatRonde(spel);

        while (spel.nogBezig())
        {
            spel.speelVerder();
            app.toonResultaatRonde(spel);
        }
    }

    private void toonResultaatRonde(Craps spel)
    {
        System.out.printf("Speler gooide %d en %d = %d%n", spel.getSteen1(), spel
.getSteen2(), spel.bepaalSom());
        if (spel.nogBezig())
            System.out.printf("Probeer opnieuw %d te gooien%n", spel.
getBijgehoudenSom());
        else
            System.out.printf("Resultaat: %s%n", spel.getSpelStatus());
    }
}
```

Mogelijke uitvoer:

```
Speler gooide 1 en 2 = 3
Resultaat: VERLOREN
```

```
Speler gooide 6 en 4 = 10
Probeer opnieuw 10 te gooien
Speler gooide 5 en 4 = 9
Probeer opnieuw 10 te gooien
Speler gooide 6 en 2 = 8
Probeer opnieuw 10 te gooien
Speler gooide 3 en 2 = 5
Probeer opnieuw 10 te gooien
Speler gooide 2 en 5 = 7
Resultaat: VERLOREN

Speler gooide 3 en 4 = 7
Resultaat: GEWONNEN

Speler gooide 3 en 5 = 8
Probeer opnieuw 8 te gooien
Speler gooide 4 en 1 = 5
Probeer opnieuw 8 te gooien
Speler gooide 5 en 6 = 11
Probeer opnieuw 8 te gooien
Speler gooide 2 en 1 = 3
Probeer opnieuw 8 te gooien
Speler gooide 4 en 4 = 8
Resultaat: GEWONNEN
```

9.3. Enumeration

Zoals we reeds eerder aangaven, is een Enumeration in de eerste plaats een speciaal soort klasse, die gebruikt wordt om een set van constanten, die unieke identifiers zijn, te definiëren.

Een enum-type is een referentietype (zoals een klasse) dat impliciet final is, vermits het constanten declareert, die niet wijzigbaar mogen zijn. De enum-constanten zijn ook impliciet static.

Objecten van een enum-type met new creëren, leidt tot een compilatiefout. Het gaat hier immers om een opsomming van elementen, waar geen nieuwe elementen meer aan kunnen toegevoegd worden.

Voorbeeld:

De enumeratie Dag somt de namen van alle dagen van de week op:


```
package utils;

public enum Dag
{
    MAANDAG,DINSDAG,WOENSDAG,DONDERDAG,VRIJDAG,ZATERDAG,ZONDAG;
}
```

Naast de lijst van enum-constanten, gescheiden door een komma, kunnen in de definitie van een Enumeration ook andere componenten voorkomen zoals bij traditionele klassen: constructors, attributen en methoden.

Voorbeeld:

Elke dag krijgt nu een dagnummer en als je een object uit de enumeratie wil laten zien op het scherm, zal het in kleine letters verschijnen dankzij de name().toLowerCase() in de toString.

```
package utils;

public enum Dag2
{
    MAANDAG(1),DINSDAG(2),WOENSDAG(3),DONDERDAG(4),VRIJDAG(5),ZATERDAG(6),ZONDAG(7);

    private final int dagNr; // moet final!

    Dag2(int dagNr) //constructor met packagetoegankelijkheid
    {
        this.dagNr = dagNr;
    }

    public int getDagNr()
    {
        return dagNr;
    }

    @Override
    public String toString()
    {
        return name().toLowerCase();
    }
}
```

Een enumeratie wordt vaak gebruikt in case-labels bij een switch of in een for-statement.

Dit wordt geïllustreerd in volgende applicatie:

```
package cui;

import java.security.SecureRandom;
import java.util.EnumSet;
import utils.Dag;
import utils.Dag2;

public class EnumApplicatie
{
    public static void main(String[] args) {

        System.out.println("Dagen van de week");
        for (Dag d : Dag.values())
        {
            System.out.println(d.name()); // name() is altijd beschikbaar!
          // of: System.out.println(d);
        }

        System.out.println("Lang weekend");
        for (Dag d : EnumSet.range(Dag.VRIJDAG, Dag.ZONDAG))
```

```
        {
            System.out.println(d.name());
        }

        for (Dag2 d : Dag2.values())
        {
            System.out.println(String.format("Dag %d: %s", d.getDagNr(), d.name()));
        }

        for (Dag2 d : Dag2.values())
        {
            System.out.println(String.format("Dag %d: %s", d.getDagNr(), d));
        }

        SecureRandom s = new SecureRandom();
        int getal;
        Dag d;

        for (int i = 0; i < 10; i++)
        {
            getal = s.nextInt(7);
            d = Dag.values()[getal];// 0 - 6
            System.out.println(String.format("Dagnummer %d: %s", getal + 1, d.
name()));
        }

        d = Dag.MAANDAG;

        if (d == Dag.MAANDAG)
        {
            System.out.println("Eerste dag van de werkweek.");
        } else
        {
            System.out.println("Dit is niet de eerste dag van de werkweek.");
        }

        d = Dag.values()[3];
        System.out.println(d);
    }
}
```

De uitvoer van deze applicatie is:

```
Dagen van de week
MAANDAG
DINSDAG
WOENSDAG
DONDERDAG
VRIJDAG
ZATERDAG
ZONDAG
Lang weekend
VRIJDAG
ZATERDAG
ZONDAG
Dag 1: MAANDAG
Dag 2: DINSDAG
Dag 3: WOENSDAG
Dag 4: DONDERDAG
Dag 5: VRIJDAG
Dag 6: ZATERDAG
Dag 7: ZONDAG
Dag 1: maandag
Dag 2: dinsdag
Dag 3: woensdag
Dag 4: donderdag
Dag 5: vrijdag
Dag 6: zaterdag
Dag 7: zondag
Dagnummer 3: WOENSDAG
Dagnummer 5: VRIJDAG
Dagnummer 1: MAANDAG
Dagnummer 2: DINSDAG
Dagnummer 7: ZONDAG
Dagnummer 7: ZONDAG
Dagnummer 4: DONDERDAG
Dagnummer 7: ZONDAG
Dagnummer 2: DINSDAG
Dagnummer 1: MAANDAG
Eerste dag van de werkweek.
DONDERDAG
```

9.4. Scope van variabelen

De scope of het bereik van een variabele kan je bepalen aan de hand van volgende basisregels:

- Een parameter heeft als bereik de body van de methode waarin de parameter gedeclareerd werd
- Een lokale variabele is “bereikbaar” vanaf het punt waar ze gedeclareerd werd tot aan het einde van dit blok
- Een lokale variabele die gedeclareerd wordt in het initialisatiegedeelte van de header een for-lus is “bereikbaar” in de rest van die header én in de body van de for-lus

- Een methode en een attribuut hebben als bereik de volledige body van de klasse. Hierdoor kunnen de methodes in een klasse de attributen en andere methodes gebruiken!

Een **schaduwveld** is een lokale variabele of parameter die dezelfde naam heeft als een objectvariabele, waardoor de objectvariabele tijdelijk “overschaduwd” (en dus onzichtbaar) wordt.

In het volgende voorbeeld maken we gebruik van een objectvariabele x en twee lokale variabelen, ook met als naam x. Een objectvariabele heeft **klasse scope**, m.a.w. deze objectvariabele x kan gebruikt worden in alle methoden en in alle inwendige klassen van de klasse Scope.

```
package cui;

public class Scope
{
    // field that is accessible to all methods of this class
    private int x = 1;

    // application starting point
    public static void main(String args[])
    {
        Scope testScope = new Scope();
        testScope.begin();
    } // end main

    // method begin creates and initializes local variable x
    // and calls methods useLocalVariable and useField
    public void begin()
    {
        int x = 5; // method's local variable x shadows field x

        System.out.printf("local x in method begin is %d\n", x);

        useLocalVariable(); // useLocalVariable has local x
        useField(); // useField uses class Scope's field x
        useLocalVariable(); // useLocalVariable reinitializes local x
        useField(); // class Scope's field x retains its value

        System.out.printf("\nlocal x in method begin is %d\n", x);
    } // end method begin

    // create and initialize local variable x during each call
    public void useLocalVariable()
    {
        int x = 25;// initialized each time useLocalVariable is called

        System.out.printf("\nlocal x on entering method useLocalVariable is %d\n",x );
      ++x; // modifies this method's local variable x
        System.out.printf("local x before exiting method useLocalVariable is %d\n",x
);
   } // end method useLocalVariable

   // modify class Scope's field x during each call
   public void useField()
    {
        System.out.printf("\nfield x on entering method useField is %d\n",x );
      x *= 10; // modifies class Scope's field x
        System.out.printf("field x before exiting method useField is %d\n",x );
   } // end method useField
}
```

In de methode **useLocalVariable** bevindt zich een lokale variabele x die **block scope** heeft en dus bereikbaar is vanaf de declaratie tot en met het einde van de methode. Het is dus deze x die bij elke aanroep geïnitialiseerd, geprint, met 1 verhoogd en nog eens geprint wordt.

In de methode **useField** is er geen lokale x en gebruiken we dus de objectvariabele x, die **klasse scope** heeft. Deze x wordt geprint, met 10 vermenigvuldigd en opnieuw geprint. Aangezien ze binnen de methode niet geïnitialiseerd wordt, werken we dus steeds verder met de vorige waarde.

```
local x in method begin is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```

9.5. final attribuut

```
private final int aantal;
```

Betekenis van het keyword **final**: eens het attribuut aantal een waarde heeft gekregen, kan deze waarde niet meer wijzigen! Het is dus een constante.

Final attributen **MOETEN** een initiële waarde krijgen bij de declaratie OF in elke constructor.

Als aantal dus in een klasse Voorbeeld voorkomt, kan je in de constructor van die klasse een waarde geven aan deze constante.

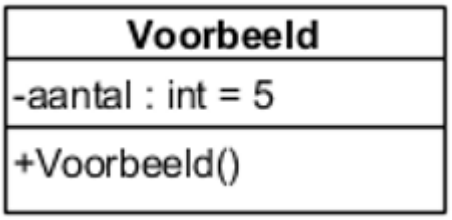
```
public Voorbeeld()
{
    aantal = 5;
}
```

Aangezien een final attribuut ten laatste een initiële waarde krijgt in de constructor, heb je **NOOIT** een setter voor zo'n attribuut.

Als een attribuut dus niet mag veranderen van waarde, dan maak je het final.

Als je ervoor kiest om de waarde van de constante reeds in te stellen bij de declaratie, is dit ook

zichtbaar in de UML.



9.6. BigDecimal voor nauwkeurige berekeningen

Bij het werken met een binaire voorstelling van kommagetallen kunnen er altijd afrondingsfouten voorkomen. Om berekeningen te doen met decimale getallen die echt nauwkeurig moeten zijn, kan je dus geen float of double gebruiken. Maak in dat geval altijd gebruik van een object van BigDecimal.

In financiële toepassingen is BigDecimal dus altijd een goede keuze. Laat ons daarom het voorbeeld uit hoofdstuk 2 voor de berekening van de interest even herschrijven zodat er gebruik gemaakt wordt van BigDecimal.

Deze nieuwe applicatie ziet er dan als volgt uit:

```
package cui;

import java.math.BigDecimal;
import java.text.NumberFormat;

public class Interest
{
    public static void main(String[] args)
    {
①
        BigDecimal principal = BigDecimal.valueOf(1000.0);
        BigDecimal rate = BigDecimal.valueOf(0.05);

        System.out.printf("%s%20s%n", "Year", "Amount on deposit");

        for (int year = 1; year <= 10; year++)
        {
            BigDecimal amount = principal.multiply(rate.add(BigDecimal.ONE).pow(
year)); ②
            System.out.printf("%4d%20s%n", year, NumberFormat.getCurrencyInstance
().format(amount)); ③
        }
    }
}
```

① De static methode valueOf van de klasse BigDecimal kent de exacte waarde (via de Stringrepresentatie die je bekomt met de methode `Double.toString(double)`) toe aan een BigDecimal object.

② De wiskundige operators +, -, * en / worden vervangen door methode-aanroepen (van de respectievelijke methodes add, subtract, multiply en divide) en geven opnieuw een precies resultaat, zonder afrondingsfouten.

③ De klasse NumberFormat laat toe om getallen te formatteren. In dit geval formatteren we het getal als een bedrag volgens de landinstellingen van het toestel waarop de software uitgevoerd wordt.

Met Belgische landinstellingen:

```
Year   Amount on deposit
   1          1.050,00 €
   2          1.102,50 €
   3          1.157,62 €
   4          1.215,51 €
   5          1.276,28 €
   6          1.340,10 €
   7          1.407,10 €
   8          1.477,46 €
   9          1.551,33 €
  10          1.628,89 €
```

Met Amerikaanse landinstellingen:

```
Year   Amount on deposit
   1           $1,050.00
   2           $1,102.50
   3           $1,157.62
   4           $1,215.51
   5           $1,276.28
   6           $1,340.10
   7           $1,407.10
   8           $1,477.46
   9           $1,551.33
  10           $1,628.89
```

9.7. static klassevariabelen

Een static klassevariabele bestaat maar één keer voor alle objecten van die klasse. De objecten delen de static variabele.

Zo'n attribuut wordt gebruikt om klasse-informatie in op te slaan. Aangezien elke klassevariabele klasse-scope heeft, kan je deze variabele dus gebruiken in alle methoden van de klasse.

Mogelijk gebruik:

- het aantal objecten bijhouden dat reeds van de klasse aangemaakt is: het is niet nodig dat elk object zijn eigen kopie heeft van deze informatie, maar dit soort informatie mag "centraal" in de klasse bijgehouden worden
- de aangroeiïnterest op een spaarrekening: het is de bedoeling dat ALLE klanten van de bank

DEZELFDE interestvoet krijgen, dus het is best dat deze op één plaats bijgehouden wordts

Voordelen bij het gebruik:

- minder geheugenruimte nodig bij gedeelde variabelen: zelfs al zijn er 1000 objecten, dan nog moet de informatie maar 1 keer bijgehouden worden!
- minder tijd nodig om de variabele te wijzigen: je wijzigt de informatie op één plaats en je bent verzekerd dat alle objecten van de wijziging op de hoogte gaan zijn en op die manier kan je dus ook geen fouten maken doordat het op één plaats al gewijzigd is en op een andere nog niet

Net zoals alle andere attributen wordt een static klassevariabele bijna altijd als private (en dus enkel toegankelijk binnen de klasse zelf) gedefinieerd. Uitzonderlijk kan het ook public gedefinieerd zijn, zoals bijvoorbeeld de klassevariabele PI. Je kan dan de waarde ook in andere klassen opvragen zonder dat je een getter nodig hebt. Je gebruikt daarvoor een object van de klasse of de klassenaam, gevolgd door een punt. Bijvoorbeeld: Math.PI

Wanneer er nog geen objecten van de klasse bestaan, dan gebruik je de naam van de klasse. Voor private attributen zal een public static methode (getter) nodig zijn om de waarde op te vragen.

Voorbeeld:

```
package domein;

public class VoorbeeldStatic
{
    private static int aantal;
    private final int id;

    public VoorbeeldStatic()
    {
        id = ++aantal;
    }

    public static int getAantal()
    {
        return aantal;
    }

    public int getId()
    {
        return id;
    }
}
```

```java
package cui;

import domein.VoorbeeldStatic;

public class VoorbeeldStaticApplicatie
{
    public static void main(String[] args)
    {
        System.out.printf("Aantal objecten: %d%n", VoorbeeldStatic.getAantal());

        VoorbeeldStatic obj1 = new VoorbeeldStatic();
        VoorbeeldStatic obj2 = new VoorbeeldStatic();
        VoorbeeldStatic obj3 = new VoorbeeldStatic();

        System.out.printf("ID van object 1 is %d%n", obj1.getId());
        System.out.printf("ID van object 2 is %d%n", obj2.getId());
        System.out.printf("ID van object 3 is %d%n", obj3.getId());

        // kan ook via een ander object opgevraagd worden of via de klassenaam
        System.out.printf("Aantal objecten: %d%n", obj2.getAantal());
    }
}
```

Dit geeft als output:

```
Aantal objecten: 0
ID van object 1 is 1
ID van object 2 is 2
ID van object 3 is 3
Aantal objecten: 3
```