

## Klassen en objecten

### Doelstellingen

Na dit hoofdstuk

- weet je dat een klasse een nieuwe datatype bepaalt
- besef je dat elk object eigen data bevat in z'n velden
- weet je hoe objecten d.m.v. constructoren geïnitialiseerd worden
- kan je zelf constructoren implementeren
- weet je wat een default constructor is
- ken je de verschillen tussen primitieve en complexe types
- kan je de geheugenlayout van objecten uittekenen
- weet je wat referenties zijn
- weet je wat een lege of `null` referentie is
- ken je het onderscheid tussen `public` en `private`, en kan je deze specifiers zinvol toepassen
- ken je getters (accessoren) en setters (mutatoren) en weet je ze zinvol te gebruiken
- kan je je eigen complexe types als parameter- of return-type gebruiken
- weet je wat de referentie `this` is en kan je er mee werken in je eigen code
- ken je het onderscheid tussen klasse- en objectmethoden

## 2.1 Inleiding

Uit de cursus Java Basisconcepten weet je reeds dat alle variabelen een welbepaalde scope hebben, aangeduid met accolades. Wanneer een variabele uit scope gaat, wordt het geheugen dat door die variabele werd ingenomen, vrijgegeven. Een variabele kan van een primitief type zijn (int, double, char ...) of van het objecttype (String, ...). Een variabele van een primitief type bevat effectief de waarde - dit is mogelijk omdat de grootte van de primitieven is vastgelegd in de taal. Een variabele van een objecttype daarentegen bevat eigenlijk een referentie naar een object. Wanneer een object variabele uit scope gaat, wordt enkel de referentie vrijgegeven, niet het object zelf. Dat object wordt later door de garbage collector opgekuist, namelijk wanneer geen enkele verwijzing meer bestaat naar dat object. In dit hoofdstuk gaan we dieper in op wat een object eigenlijk is, hoe het in het geheugen wordt bewaard, hoe objecten met elkaar communiceren. Een OO-programma bestaat uit een aantal objecten die met elkaar communiceren door boodschappen naar elkaar te sturen. De mogelijkheden zijn onbeperkt, en eens je dit beet hebt, heb je al een groot deel van het mysterie rond OO programmeren ontsluierd!

## 2.2 Klasse als datatype

Met de kennis opgedaan uit de cursus Java Basisconcepten is er geen mogelijkheid om gegevens van diverse soort samen te brengen tot één geheel. En dat is een beperking. Het zou bv. zeer praktisch zijn om diverse gegevens van een persoon samen te brengen: zijn naam en voornaam (String), zijn leeftijd (int) en zijn geslacht (boolean). We weten reeds dat we via arrays (rijen) een set van getallen, karakters, strings, ... als één geheel kunnen manipuleren. Denk maar aan het optellen van getallen uit een rij. Rijen zijn echter steeds gedefinieerd als een verzameling van gegevens van hetzelfde type. Het concept rijen is aldus onbruikbaar om er de gegevens van een persoon in te bewaren. Meer nog, het zou fijn zijn als we kunnen nagaan of 2 personen even oud zijn, of kijken of een persoon mannelijk of vrouwelijk is. We willen m.a.w. niet alleen meer complexe gegevensstructuren kunnen bouwen, we willen er ook bewerkingen op uit kunnen voeren.

Het Java gereedschap dat je hiervoor nodig hebt is een klasse : via de velden definieer je de diverse set van gegevens, via de methoden van de klasse bepaal je de gewenste functionaliteit. Velden in een klasse kunnen van het primitief type zijn of

*overloading  
is methoden van zelfde  
naam maar andere  
param.*

Eén soort methoden vervult een heel specifieke rol: **de constructoren**. Ze worden gebruikt om een object aan te maken van de klasse en kunnen enkel geactiveerd worden via het sleutelwoord **new**. De techniek van **overloading** laat toe om in één klasse meerdere constructoren te hebben.

De objectreferentie die je krijgt als resultaat van een constructor-oproep bevat de geheugenlocatie van het gecreëerde object. Via deze referentie wordt toegang tot het object mogelijk gemaakt!

Het object is een **instantie** van de klasse. De toestand van het object wordt bepaald door de waarden van de velden van de klasse, zijn gedrag door de objectmethoden die worden opgeroepen. De klasse zelf kan je zien als een **blueprint** voor een verzameling van gelijkaardige objecten.

## 2.3 Objecten als containers van data

Uit het vorige hoofdstuk kan je de indruk opgedaan hebben dat een klasse een soort vergaarbak van methoden is, en dat je objecten nodig hebt om de methoden uit die klasse aan te spreken. Hernemen we onze klasse Punt:

```
// Punt.java
public class Punt {
    public double berekenAfstand(int x1, int y1,
                                  int x2, int y2) {
        return Math.sqrt(Math.pow(x2 - x1, 2)
                        + Math.pow(y2 - y1, 2));
    }
}
```

Om de methode **berekenAfstand** in een andere klasse te kunnen gebruiken, hebben we een Punt-object nodig, bv.

```
Punt p = new Punt();
double afst = p.berekenAfstand(1, 2, 3, 0);
```

Zonder een object **p**, dat via **new** werd aangemaakt, kunnen we de methode **berekenAfstand** niet uitvoeren. Het object **p** wordt het oproepend object genoemd. De methode wordt namelijk aangeroepen via het oproepend object, gebruik makend van de **member-operator** (**.**) tussen beide.

Ok, op die manier kunnen we methoden uit een andere klasse aanspreken, maar hoe zit het dan met de velden? Daar hebben we stilletjes over gezwegen, maar eigenlijk is het systeem gelijkaardig. Stel dat we twee velden in onze klasse Punt zetten, één voor de x- en één voor de y-coördinaat:

```
// Punt.java
public class Punt {
    int x;
    int y;

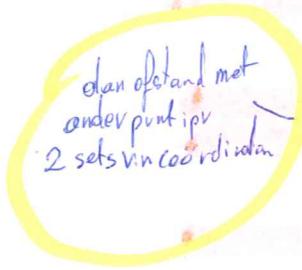
    ...
}
```

Uit de cursus Java Basisconcepten weet je reeds dat velden variabelen zijn op het niveau van de klasse en dat je die velden in elke methode van die klasse rechtstreeks kan aanspreken. Als we die variabelen echter buiten de klasse willen aanspreken, dan hebben we, analoog als voor methoden, ook een object nodig, p in onderstaand voorbeeld:

```
Punt p = new Punt();
p.x = 1;
p.y = 2;
```

Zonder een object p, dat via new werd aangemaakt, kunnen we de velden x en y niet aanspreken. Een veld wordt aangesproken via het oproepend object p, gebruik makend van de **member-operator (.)** tussen beide.

Als we het zo beschouwen, kunnen we de methode berekenAfstand dus eigenlijk herschrijven zodat er geen 4 maar slechts 2 parameters nodig zijn, we kunnen immers de x- en y-coördinaat van één van de punten in de velden stockeren.



```
// Punt.java
public class Punt {
    int x;
    int y;

    public double berekenAfstand(int andereX,
                                  int andereY) {
        return Math.sqrt(Math.pow(andereX - x, 2)
                        + Math.pow(andereY - y, 2));
    }
}
```

De oproepende code ziet er dan bv. als volgt uit:

```
Punt p = new Punt();
p.x = 1;
p.y = 2;
double afstand = p.berekenAfstand(3, 0);
```

Je zou nu de neiging kunnen hebben om meteen nog twee velden te voorzien, dan heeft de methode berekenAfstand helemaal geen parameters nodig, net zo gemakkelijk. Deze redenering zou misschien opgaan in dit beperkt voorbeeld, maar

niet alles wat je met een punt wilt doen is direct ook van toepassing op twee punten, en dan zit je daar met die velden die je eigenlijk niet nodig hebt.

Je kan de relevante set van velden voor een klasse best bepalen door na te gaan wat geldt als de karakteristieke data voor die klasse. Een punt wordt, althans in een tweedimensionaal stelsel, gekenmerkt door zijn x- en y-coordinaat.

Bovenstaande, gecombineerd met de wetenschap dat je meerdere objecten kan maken van een klasse, bieden een oplossing:

```
Punt p1 = new Punt();
p1.x = 1;
p1.y = 2;

Punt p2 = new Punt();
p2.x = 3;
p2.y = 0;

double afstand = p1.berekenAfstand(p2.x, p2.y);
```

Dit lijkt een zeer triviale uitbreiding op wat we al hadden, maar eigenlijk zet dit de toon voor een heel belangrijk idee: **elk object bevat eigen data in z'n velden.** Het veld **x** dat we via **p2** de waarde 3 geven is niet dezelfde **x** die via **p1** de waarde 1 heeft gekregen, dat zijn andere variabelen! **Zodra via new een nieuw object wordt gemaakt, worden voor de velden afzonderlijke variabelen in het leven geroepen die elk hun eigen waarden kunnen bevatten.** En al die velden/variabelen samen zijn via dat ene object aanspreekbaar. **Elk object is m.a.w. een container** waarin we een aantal gegevens kunnen bewaren, en welke gegevens dat zijn, dat bepalen de velden van de klasse. Nemen we even het voorbeeld van een Persoon:

```
// Persoon.java
public class Persoon {
    String naam;
    String voornaam;
    int leeftijd;
    boolean geslacht;
}
```

We kunnen nu twee Persoon-objecten aanmaken, en elk van die objecten andere waarden geven voor de velden:

```
Persoon pers1 = new Persoon();
pers1.naam = "Gruyaert";
pers1.voornaam = "Hans";
pers1.leeftijd = 32;
pers1.geslacht = false; // mannelijk

Persoon pers2 = new Persoon();
```

```

pers2.naam = "Van Assche";
pers2.voornaam = "Kristien";
pers2.leeftijd = 37;
pers2.geslacht = true; // vrouwelijk

```

Dit lijkt nog niet meteen zo verschrikkelijk nuttig, maar blijf er even bij, het zal nog duidelijk worden hoeveel je hier mee kunt doen. Wat er sowieso wel al omslachtig lijkt, is dat al die velden in afzonderlijke instructies waarden moeten krijgen. Het kan ons bovendien ook in de problemen brengen, want stel dat bepaalde velden niet werden geïnitialiseerd en er methoden net van die velden gebruik maken, wat levert dat dan als resultaat op?

```

// Persoon.java
public class Persoon {
    String naam;
    String voornaam;
    int leeftijd;
    boolean geslacht;

    public boolean isVolwassen() {
        return (leeftijd >= 18);
    }
}

// ergens anders
Persoon p = new Persoon();
if (p.isVolwassen()) {
    // doe iets
}

```

Er bestaan speciale methoden om net dat initialisatie-proces van objecten een-  
voudiger en veiliger te maken, deze methoden worden de *constructoren* genoemd.

## 2.4 Objecten initialiseren

Een constructor is een methode die gebruikt wordt om een object te initialiseren.

In concreto betekent dit dat de constructor ervoor verantwoordelijk is dat de velden van dat object een zinvolle beginwaarde krijgen. Constructoren zijn echter geen gewone methoden, in die zin dat er in hun signatuur geen return-type voorkomt, en dat de naam identiek moet zijn als die van de klasse, inclusief hoofdletter. Laat ons een voorbeeldje bekijken voor onze klasse Punt:

```

// Punt.java
public class Punt {

```

```

int x;
int y;

public Punt(int initX, int initY) {
    x = initX;
    y = initY;
}
}

```

Wat valt er op? De naam van de methode is gelijk aan die van de klasse, nl. **Punt**. Hoe zie je trouwens dat het hier om een methode gaat? Aan de ronde haakjes met de parameters die er tussen staan. Nog iets? Ja, tussen het sleutelwoord **public** en de naam staat er ook geen type, geen **void**, geen **int**, geen **String**, niets. En waarom hebben we hier net die parameters? Omdat we twee getallen nodig hebben om de velden te initialiseren.

Maar er is nog iets bijzonder aan een constructor. **De enige manier om ze op te roepen is via new**, bij creatie van het object m.a.w. En dat is ook logisch, want constructoren dienen net om objecten te initialiseren, een beginwaarde te geven, dat kunnen we dus maar één keer doen:

**Punt p = new Punt(1, 2);**

doet hetzelfde maar cleaner!

In plaats van niets tussen de haakjes, kunnen we vanaf nu twee getallen meegeven die de constructor zal gebruiken om de velden te initialiseren. Bovenstaande instructie is identiek aan de volgende code die we hadden, maar een stuk cleaner toch?

**Punt p = new Punt();  
p.x = 1;  
p.y = 2;**

We zijn er nog niet helemaal, want hoe komt het dat, tot één pagina geleden bij manier van spreken, het aanmaken van objecten altijd heeft gewerkt, zonder dat we van constructoren hadden gehoord, laat staan er zelf één hebben geschreven? Dat komt omdat **constructoren eigenlijk een noodzaak zijn bij de creatie en initialisatie van objecten**, en dat de **Java compiler zelf een standaard constructor zal voorzien in klassen waar er geen expliciet in voorkomt**. Die code zie je natuurlijk niet in je broncode omdat die er pas bij compilatie wordt toegevoegd.

Nu is het wel zo dat, van zodra je in een klasse wel expliciet een constructor definieert, de compiler dit niet meer doet. Van volgende klasse **Punt** kunnen we m.a.w een object aanmaken met de **constructor zonder argumenten (ook wel standaard constructor of default constructor genoemd)**:

**// Punt.java**

```

public class Punt {
    int x;
    int y;
}

// ergens anders
Punt p = new Punt(); // lukt wel

```

Wanneer we echter zelf een niet-default constructor voorzien, dan kunnen we de default constructor niet meer gebruiken:

```

// Punt.java
public class Punt {
    int x;
    int y;

    public Punt(int initX, int initY) {
        x = initX;
        y = initY;
    }
}

// ergens anders
Punt p = new Punt(); // lukt niet!

```

We kunnen desgewenst natuurlijk wel **zelf ook een default constructor schrijven!**

Java laat toe dat er in een klasse meerdere methoden met dezelfde naam voorkomen - dit wordt **method overloading** genoemd. De voorwaarde is dat de parameters in type en/of in aantal veranderen. Wanneer we dit toepassen op constructoren kunnen we dus eigenlijk nog een tweede, of een derde constructor voorzien, kwestie van indien nodig de keuze te hebben over hoe je een object wilt initialiseren.

```

// Punt.java
public class Punt {
    int x;
    int y;

    public Punt() {
        x = 0;
        y = 0;
    }

    public Punt(int initX, int initY) {
        x = initX;
        y = initY;
    }
}

```

```
// ergens anders
Punt p1 = new Punt(); // lukt!
Punt p2 = new Punt(1, 2); // lukt ook!
```

Laat ons bij wijze van voorbeeld nog eens 2 mogelijke constructoren bekijken voor de klasse Persoon:

```
// Persoon.java
public class Persoon {
    String naam;
    String voornaam;
    int leeftijd;
    boolean geslacht;

    public Persoon(String n, String v) {
        naam = n;
        voornaam = v;
        leeftijd = 18; // default leeftijd
        geslacht = true; // default geslacht
    }

    public Persoon(String n, String v,
                   int l, boolean g) {
        naam = n;
        voornaam = v;
        leeftijd = l;
        geslacht = g;
    }
}
```

Wanneer je echter de parameters dezelfde namen geeft als de velden, krijgen we volgende dubieuze code:

```
// Persoon.java
public class Persoon {
    String naam;
    String voornaam;
    ...

    public Persoon(String naam, String voornaam) {
        naam = naam;
        voornaam = voornaam;
        ...
    }

    ...
}
```

*Onvoorstelbaar!*

Omdat er **geen manier** is voor de compiler om onderscheid te maken tussen **velden en parameters**, zal de compiler steeds de parameters gebruiken. In bovenstaande code zal de constructor dus parameters aan zichzelf gelijkstellen en bijgevolg worden de **velden niet geïnitialiseerd!**

In Java wordt het **sleutelwoord this** gebruikt om de actieve context aan te spreken (zie ook sectie 2.9) Hiermee kunnen we aangeven dat we de velden van het aan te maken object een waarde willen geven:

```
// Persoon.java
public class Persoon {
    String naam;
    String voornaam;
    ...

    public Persoon(String naam, String voornaam) {
        this.naam = naam;
        this.voornaam = voornaam;
        ...
    }
    ...
}
```

De **naam** van het te initialiseren object wordt zo gelijkgesteld aan de parameterwaarde **naam**. De **voornaam** van het nieuw aan te maken object wordt ingesteld op de waarde van de gelijknamige parameter **voornaam**. Dit is een gangbare manier om objecten te initialiseren! *of geef param als naam => naam => Praktijk*

## 2.5 Primitieve vs complexe types

Een datatype bepaalt het soort gegevens die in een variabele kunnen bewaard worden: bij int zijn dat gehele getallen, bij String tekst, bij char letters, etc. In Java bepaalt **elke klasse een nieuw datatype**. D.w.z. dat we niet alleen variabelen kunnen declareren van een bestaand type, maar ook van onze eigen types, bv.

```
int getal;
String input;
Punt punt;
```

In Java wordt een duidelijk onderscheid gemaakt tussen primitieve types, ook wel waardetypes genoemd, en complexe types, ook wel objecttypes genoemd. Zoals je weet uit de cursus Java Basisconcepten zijn er **8 primitieve types in Java (of 9 als je void meetelt)**. Dat aantal ligt vast, er kunnen **geen nieuwe primitieve types**



aangemaakt worden. Bij de complexe types is dit anders, aangezien elke klasse een nieuw complex datatype creëert. Deze types worden complex genoemd omdat in één object verschillende gegevens kunnen gecombineerd worden, zoals we in de vorige paragraaf hebben gezien.

**De manier waarop primitieve en complexe types bewaard worden, de geheugenlayout, is ook anders.** Aangezien primitieve types slechts één gegeven bevatten, worden die rechtstreeks in de variabele gestockeerd. Door bv. een variabele van het type `int` te maken worden 32 bits gereserveerd waarin het geheel getal kan bewaard worden. **Bij complexe types wordt niet de waarde maar een referentie bewaard.** Een referentie is niets anders dan het adres van het stukje geheugen waar de gegevens van dat object te vinden zijn.

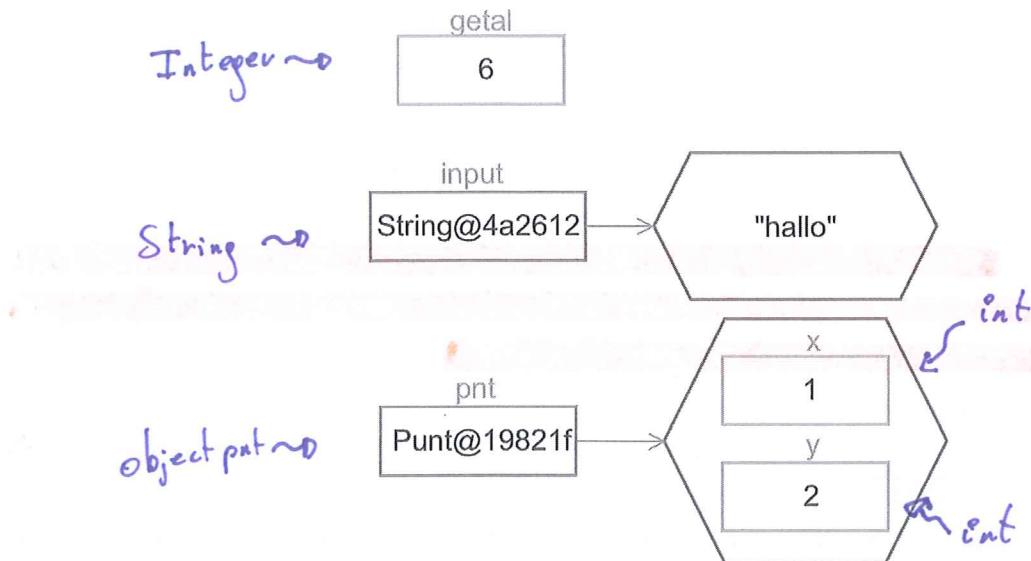
**De manier waarop we primitieve en complexe variabelen een waarde kunnen geven is hierdoor ook anders.** Primitieven kan je simpelweg initialiseren door er een waarde aan toe te kennen. Complex types moet je initialiseren d.m.v. een constructor. De **constructor zal twee dingen doen**: enerzijds **geheugenruimte voorzien voor de velden en die initialiseren**, anderzijds een **referentie naar dat geheugen in de variabele stoppen**. Bovenstaande variabelen kunnen bv. als volgt geïnitialiseerd worden. Merk op dat er **voor String een shortcut voorzien is aangezien dit type zo vaak gebruikt wordt.** ↪ `= " "; ipr new String();`

```
int getal = 6;
String input = new String("hallo");
// kan ook met String input = "hallo";
Punt punt = new Punt(1, 2);
```

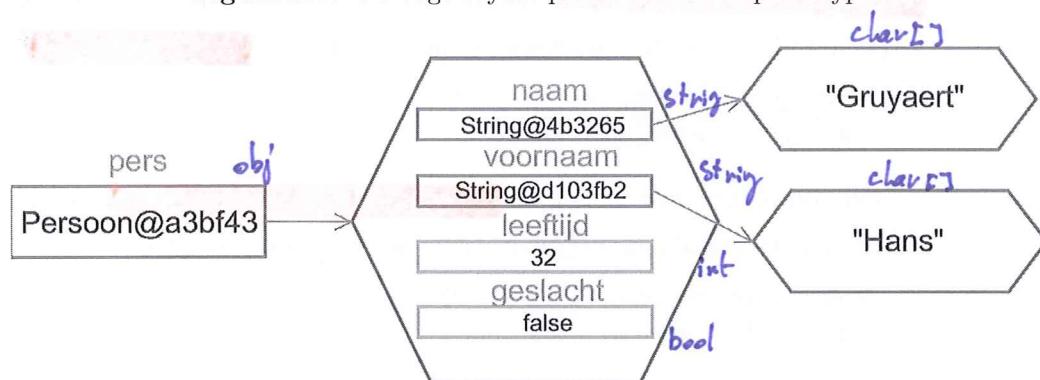
In figuur 2.1 zie je hoe de geheugenlayout er schematisch uitziet. Zoals je kunt zien bevat `getal` effectief de waarde 6, terwijl `input` en `punt` referenties bevatten naar hun respectievelijke inhoud.

De **gewone rechthoekjes zijn primitieven** en bevatten waarden, de **rechthoekjes waaruit een pijl vertrekt** bevatten referenties en de **zeshoeken zijn objecten**, die op hun beurt weer variabelen (velden) bevatten. Merk op dat we de interne opbouw van de klasse `String` niet kennen en er bijgevolg gewoon de tekstuele inhoud van tonen. **Intern is een String eigenlijk een char[]**, maar op de interne voorstelling van arrays komen we later nog terug, het belangrijkste om nu te weten is dat `String` ook een complex type is en bijgevolg met referenties werkt.

Maar wat leren we uit dit schema? Dat **de variabelen die wij aanmaken afhankelijk van hun type** waarden of referenties bevatten. En dat die referenties (in Java voorgesteld als een 6-digit hexadecimaal getal) objecten aanwijzen die, afhankelijk van het type van de velden van dat object, terug waarden of referenties bevatten. In



Figuur 2.1: Geheugenlayout primitieve vs complexe types



Figuur 2.2: Geheugenlayout van een Persoon object

het geval van `Punt` zijn de twee velden van het type `int`, maar kijken we bv. naar hoe een `Persoon` object er uitziet, dan wordt het schema iets complexer (zie figuur 2.2). **Velden kunnen op hun beurt immers ook van een complex type zijn.**

```
Persoon pers =
    new Persoon("Gruyaert", "Hans", 32, false);
```

Niet elke referentie wijst effectief naar een object. Er bestaat ook zoets als een lege referentie, een referentie die nergens naar wijst. In Java zeggen we dat die referentie de waarde `null` heeft (zie figuur 2.3). Je kan heel eenvoudig testen op een lege referentie met `==` en `!=`.

```
if (pnt != null) { bestaat pnt? ~ anders nullpointerExceptions
    // doe iets met pnt
}
```

**Figuur 2.3:** Een lege referentie

Noteer dat, wanneer je een variabele declareert van het objecttype, je hiermee enkel plaats reserveert voor de referentie, er is m.a.w. nog geen object aan gekoppeld. Onderstaande definities van pnt zijn equivalent:

```
Punt pnt = null;
```

```
Punt pnt;
```

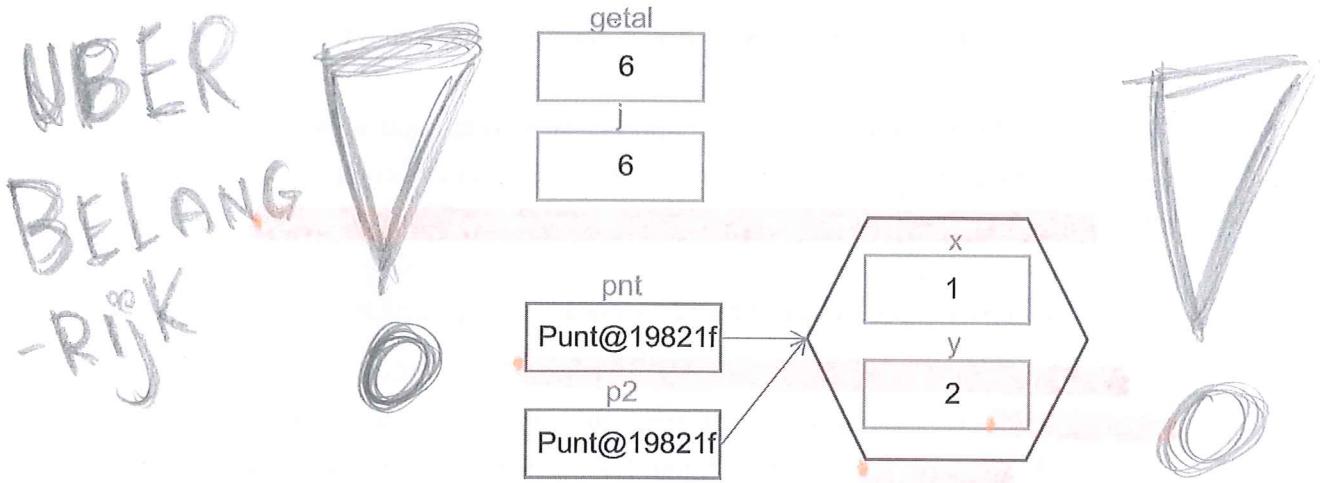
*Primitieve hebben std waarde 0 bub*

Variabelen zouden natuurlijk geen variabelen zijn indien we ze achteraf niet nog van waarde zouden kunnen veranderen. Ook hier is er een belangrijk verschil tussen primitieve en complexe types. Wanneer we aan een primitieve variabele de waarde van een andere variabele toekennen, dan wordt deze waarde gekopieerd.

```
int getal = 6;
int j = getal;
```

Bij complexe types is het de referentie die wordt gekopieerd, waardoor je dus eigenlijk twee variabelen krijgt die naar hetzelfde object wijzen.

```
Punt pnt = new Punt(1,2);
Punt p2 = pnt;
```

**Figuur 2.4:** Kopiëren van primitieve en complexe variabelen

Zoals je kunt zien in figuur 2.4 bevat *j* nu ook de waarde 6 en *p2* nu ook de referentie *Punt@19821f*. Eigenlijk wordt in beide gevallen de inhoud van de variabele

gekopieerd, alleen heeft dat kopiëren bij primitieve types een ander effect dan bij complexe types.

Wanneer je wijzigingen aanbrengt aan een object, dan doe je dit steeds via zijn referentie. Weet echter dat de kans bestaat dat nog andere variabelen refereren naar datzelfde object. Via die variabelen heb je dus ook toegang tot de gewijzigde inhoud.

```
pnt.x = 123;
System.out.println(p2.x); // geeft 123
```

## 2.6 Bewerkingen op complexe types

*(bewijst dat een type wordt pas echt interessant als we er ook bewerkingen op kunnen uitvoeren. Getallen stockeren in een int is leuk, maar ze ook kunnen optellen of delen maakt het nog veel leuker.)*

Door velden in een klasse op te nemen kunnen we complexe types bouwen, maar een type wordt pas echt interessant als we er ook bewerkingen op kunnen uitvoeren. Getallen stockeren in een int is leuk, maar ze ook kunnen optellen of delen maakt het nog veel leuker. Voor de primitieve types zijn er een aantal operatoren gedefinieerd die toelaten om er bv. mee te rekenen of ze te vergelijken. Bij complexe types ligt dat anders: daar heb je geen operatoren, daar is het met methoden te doen. Zo ken je bv. de methoden `charAt` en `toUpperCase` uit de klasse `String` die het mogelijk maken om een letter uit een string te halen of de string in hoofdletters om te zetten. Wat je ook weet is dat je methoden op strings kunt toepassen door ze via een . (ook wel de member-operator genoemd) op de variabele in kwestie uit te voeren, bv.

```
String s = "hallo";
char eersteLetter = s.charAt(0);
String hoofdletters = s.toUpperCase();
```

Voor onze eigen complexe types is dit niet anders: het zijn de methoden die we in onze klassen schrijven die het mogelijk maken om met die objecten ook effectief iets te doen. We hebben voorheen al de methode `isVolwassen` geschreven in de klasse `Persoon` om na te gaan of een persoon volwassen is, hierdoor kunnen we toch al iets nuttigs doen met ons type.

```
// Persoon.java
public class Persoon {
    String naam;
    String voornaam;
    int leeftijd;
    boolean geslacht;

    public Persoon(String naam, String voornaam,
                  int leeftijd, boolean geslacht) {
```

```

    ...
}

public boolean isVolwassen() {
    return (leeftijd >= 18);
}
}

// ergens anders
Persoon p = new Persoon("Devos", "Bob", 32, false);

if (p.isVolwassen()) {
    System.out.println("Proficiat!");
}

```

Je zou het bovenstaande natuurlijk ook anders kunnen schrijven:

```

if (p.leeftijd >= 18) {
    System.out.println("Proficiat!");
}

```

Dit is echter geen goede manier van werken. Door met methoden te werken heb je meer controle over wat er met de inhoud van je object gebeurt. Jij kiest als ontwerper van je klasse wat er met de objecten kan en mag gebeuren. **Dé manier om ongewenst gebruik van velden van buitenaf tegen te gaan is het veld privaat definiëren.** Door declaratie van een veld met toevoeging van het sleutelwoord **private** wordt de **toegang tot het betreffende veld beperkt tot de klasse zelf**. Enkel code die binnen de klasse is gesitueerd kan het veld nog aanspreken. De compiler detecteert ongeoorloofde toegang en geeft desgevallend een foutmelding:

```

// Persoon.java
public class Persoon {
    private String naam;
    private String voornaam;
    private int leeftijd;
    private boolean geslacht;

    ...
}

// ergens anders
if (p.leeftijd >= 18) { // compilatiefout!
    System.out.println("Proficiat!");
}

```

Ook de toegang tot methoden kan op gelijkaardige manier beperkt worden. Een eerder aangehaald voorbeeld hiervan is de methode `initComponents`, gegenereerd

bij grafische Netbeans toepassingen. In de keuze tussen **public** en **private** bestaat er geen regel die altijd opgaat, maar doorgaans zullen we velden **private** en methoden **public** maken.

Merk ook op dat het privaat maken van de velden meteen het gebruik van niet-standaard constructoren noodzakelijk maakt om objecten te kunnen initialiseren!

Via bijkomende publieke methodes kan je meer functionaliteit aan de klasse **Persoon** toevoegen, waardoor de buitenwereld meer met dat type kan doen, het als het ware rijker wordt. We kunnen bv. een methode schrijven om een persoon te laten verjaren, of om de volledige naam op te vragen, etc. Welke methoden je schrijft zal afhangen van wat je met de objecten van die klasse in je programma wil doen, of waar je in de toekomst denkt nood aan te hebben.

```
// Persoon.java
public class Persoon {
    ...
    public void verjaar() {
        leeftijd++;
    }

    public String geefVolledigeNaam() {
        return naam + " " + voornaam;
    }
}
```

Herinner je wel dat het resultaat van die methoden afhangt van het object waarmee je ze oproept - elk object heeft zijn eigen inhoud!

```
Persoon p1 =
    new Persoon("Gruyaert", "Hans", 32, false);
Persoon p2 =
    new Persoon("Van Assche", "Kristien", 36, false);

// geeft Gruyaert Hans als output
System.out.println(p1.geefVolledigeNaam());

// geeft Van Assche Kristien als output
System.out.println(p2.geefVolledigeNaam());
```

We kunnen op dezelfde manier methoden in de klasse **Punt** voorzien, bv. om een punt over de X-as of de Y-as met een bepaalde waarde te verschuiven. Merk op dat we de velden ondertussen ook hier **private** hebben gemaakt.

```
// Punt.java
public class Punt {
    private int x;
```

```

private int y;
...
public void verschuifX(int deltaX) {
    x += deltaX;
}
public void verschuifY(int deltaY) {
    y += deltaY;
}
}

```

In sommige gevallen is toegang tot de velden toch gewenst. In plaats van ze dan `public` te maken, is de courante praktijk om waar nodig getters en/of setters te voorzien. Een getter is een methode met als naam `getXXX`, waarbij `XXX` wordt vervangen door de naam van het veld dat opgevraagd moet kunnen worden. Analoog voor setters, waar de `setXXX` methoden dan gebruikt worden om velden een andere waarde te geven. Voor de klasse Punt zouden we volgende getters en setters kunnen

*Why Getters & Setters?*

- 1) Veiligheid - kan checken of de waarde toelaatbaar is
- 2) Korte - `readonly`?

```

// Punt.java
public class Punt {
    private int x;
    private int y;
    ...
    // getters
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    // setters
    public void setX(int nieuweX) {
        x = nieuweX;
    }
    public void setY(int nieuweY) {
        y = nieuweY;
    }
}

```

Het return-type van de getters wordt bepaald door het type van de onderliggende velden, idem voor het type van de parameters van de setters. Setters hebben altijd `void` als return.

**Getters en setters worden ook wel accessoren** (*to have access*, toegang hebben) **en mutatoren** (*to mutate*, wijzigen) **genoemd**.

Het voordeel van publieke methoden boven publiek toegankelijke velden wordt duidelijk met onderstaand voorbeeldje:

```
// Persoon.java
public class Persoon {
    ...
    private int leeftijd;

    public int getLeeftijd() {
        return this.leeftijd;
    }

    public void setLeeftijd(int leeftijd) {
        if (leeftijd >= 0) {
            this.leeftijd = leeftijd;
        }
    }
}

// ergens anders
Persoon p = new Persoon(...);
p.setLeeftijd(12);
```

De enige manier om de leeftijd van een persoon te wijzigen is door gebruik te maken van de publieke `setLeeftijd`-methode. **De gebruiker ziet de methode als een 'black-box'**, maar onderliggend is de methode **zó geïmplementeerd dat negatieve parameterwaarden als ongeldig worden beschouwd**. Hoe we de oproepende code hiervan op de hoogte brengen, zien we wat verder in dit hoofdstuk (zie sectie 2.11.3).

## 2.7 Eigen type voor parameters

We weten ondertussen dat we voor variabelen niet alleen de bestaande types maar ook onze eigen types kunnen gebruiken. Als we deze lijn doortrekken kunnen we die types dus ook gebruiken als parameter van een methode, of als returntype. **Stel dat we een methode willen om bij de x- en y-coördinaat van een punt de coördinaten van een ander punt op te tellen**. We kunnen hetvolgende proberen:

```
// Punt.java
```

```

public class Punt {
    private int x;
    private int y;

    ...

    public void telBij(int andereX, int andereY) {
        x += andereX;
        y += andereY;
    }
}

// ergens anders
Punt p1 = new Punt(1, 2);
Punt p2 = new Punt(3, 0);

p1.telBij(p2.x, p2.y);

```

Dit zou een mogelijke aanpak kunnen zijn, ware het niet dat `x` en `y` `private` zijn en bijgevolg niet meer toegankelijk buiten de klasse, en dat willen we zo houden. Alternatief kan zijn om getters te gebruiken die wel publiek zijn, maar het kan nog beter. We kunnen namelijk als parameter gewoon ons type `Punt` gebruiken, we geven m.a.w. een volledig `Punt`-object mee als argument aan de methode `telBij`. En aangezien de methode `telBij` in de klasse `Punt` zelf staat, zijn de velden van de parameter binnen de methode aanspreekbaar. En op dezelfde wijze kunnen we meteen ook een betere implementatie geven aan de eerder beschouwde methode `berekenAfstand`.

```

// Punt.java
public class Punt {
    private int x;
    private int y;

    ...

    public void telBij(Punt anderPunt) {
        this.x += anderPunt.x;
        this.y += anderPunt.y;
    }

    public double berekenAfstand(Punt anderPunt) {
        return Math.sqrt(Math.pow(anderPunt.x - x, 2)
                        + Math.pow(anderPunt.y - y, 2));
    }
}

```

```
// ergens anders
Punt p1 = new Punt(1, 2);
Punt p2 = new Punt(3, 0);

p1.telBij(p2);
double afstand = p1.berekenAfstand(p2);
```

Het merkwaardige aan dit stukje code is dat we in een methode uit een klasse tegelijkertijd met twee objecten van die klasse kunnen werken. Er is sowieso altijd het oproepend object, in ons geval p1, dit is het object waarmee de methode werd opgeroepen. Vanuit het standpunt van de methode zijn de velden van het oproepend object gewoon variabelen waarmee gewerkt kan worden, eventueel voorafgegaan door `this..`. Het is belangrijk om te beseffen dat de waarde van de velden bewaard blijft zolang het object bestaat. Op het moment dat we `telBij` via `p1` oproepen, kan `telBij` beschikken over de twee velden `x` en `y` met als respectievelijke waarden 1 en 2. Dat zijn de velden binnen de actuele context van de opgeroepen methode. Maar bij het oproepen van de methode `telBij` wordt een (ander) `Punt`-object als argument meegegeven, hier `p2`. `p2` is de actuele parameterwaarde voor de opgeroepen methode. Ook `p2` heeft twee velden `x` en `y` met als respectievelijke waarden 3 en 0. Gezien vanuit de code van `telBij` krijgt de parameter de naam `anderPunt`, dit is de formele parameternaam. De velden van dat punt zijn dan aanspreekbaar met `anderPunt.x` en `anderPunt.y`. Ingewikkeld? Best wel, maar geef het wat tijd, het loont de moeite om deze constructie onder de knie te krijgen.

Nemen we een tweede voorbeeld. We willen bv. nagaan of twee personen een gelijke familienaam hebben. De methode zou er als volgt uit kunnen zien:

```
// Persoon.java
public class Persoon {
    private String naam;
    ...

    public boolean heeftZelfdeNaamAls(Persoon p) {
        return naam.equals(p.naam);
    }
}

// ergens anders
Persoon p1 = new Persoon(...);
Persoon p2 = new Persoon(...);

if (p1.heeftZelfdeNaamAls(p2)) {
    ...
}
```

Hetzelfde systeem: de methode `heeftZelfdeNaamAls` heeft een parameter `p` van het type `Persoon`. Vermits de methode in de klasse `Persoon` staat zijn de velden van `p` aanspreekbaar via `p.naam`, `p.voornaam`, etc. De variabele `naam` (of `this.naam`) die gebruikt wordt, staat voor de naam van het oproepend object, hier `p1`. Met `p.naam` wordt de naam van het argument-object `p2` beschouwd.

## 2.8 Eigen type als return-type

We kunnen onze eigen types ook als return-type voor een methode gebruiken. Stel dat we de som van twee punten willen maken en dat dit een nieuw punt moet vormen. Een nieuw `Punt`-object maken doe je aan de hand van een constructor (al dan niet default). Constructoren kan je ook binnen een methode van een klasse gebruiken. De methode `berekenSom` zou er als volgt uit kunnen zien:

```
// Punt.java
public class Punt {
    ...

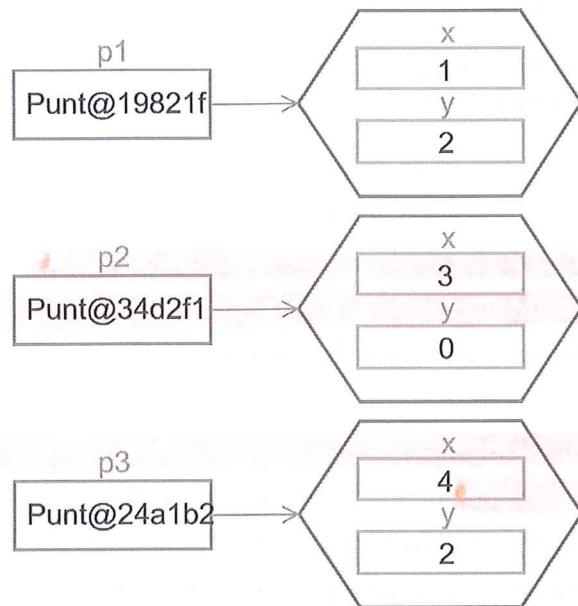
    public Punt berekenSom(Punt anderPunt) {
        Punt s = new Punt(x + anderPunt.x,
                           y + anderPunt.y);
        return s;
    }
}

// ergens anders
Punt p1 = new Punt(1, 2);
Punt p2 = new Punt(3, 0);

Punt p3 = p1.berekenSom(p2);
```

In de methode wordt een variabele `s` gedeclareerd van het type `Punt`. Aan `s` wordt vervolgens het resultaat van een nieuw aangemaakt `Punt`-object toegekend. Dit nieuwe punt is gecreëerd aan de hand van een niet-default constructor. De argumenten van de constructor zijn zó bepaald dat het nieuwe punt de som is van het huidige punt (context van het oproepend object) en het meegegeven punt: Als `x`-coördinaat wordt de som genomen van het `x`-veld van het oproepend object met het `x`-veld van het argument-object (formele parameter `anderPunt`), analoog voor de `y`-coördinaat. Het return-type van de methode `berekenSom` is `Punt`, dus de variabele `s` kan gereturneerd worden.

Noteer dat, aangezien Punt een objecttype is, het eigenlijk een referentie is die wordt toegekend aan de variabele p3. In figuur 2.5 zie je het geheugenschema nadat de methode berekenSom werd uitgevoerd met oproepend object p1, argument-object p2 en resulterend object p3. Voor p3 werd effectief een nieuw object gecreëerd.



Figuur 2.5: De som van twee punten gestockeerd in een nieuw object

## 2.9 Het oproepend object

Methoden uit een klasse moeten vanuit een andere klasse steeds via een object opgeroepen worden, daarom worden ze ook wel objectmethoden genoemd. Het object waarmee de methode wordt aangesproken hebben we het oproepend object genoemd. Het is dat object dat de context van de methode bepaalt. Nemen we er even terug volgende code bij:

```
// Punt.java
public class Punt {
    private int x;
    private int y;

    ...

    public void telBij(Punt anderPunt) {
        x += anderPunt.x;
        y += anderPunt.y;
    }
}
```

```

}

// ergens anders
Punt p1 = new Punt(1, 2);
Punt p2 = new Punt(3, 0);
Punt p3 = new Punt(5, 7);

p1.telBij(p2);
p3.telBij(p2);

```

De waarden van de velden `x` en `y` in de methode `telBij` worden bepaald door het object waarmee de methode werd opgeroepen, in bovenstaand voorbeeld is dat `p1`. **Roepen we de methode op via een ander object** (bv. `p3`), **dan is er een andere context voor de methode met mogelijk andere waarden voor de velden tot gevolg**, `p1` en `p3` vormen men in bovenstaande code de oproepende objecten voor de methode `telBij`.

**In Java kan je met het sleutelwoord `this` steeds de actieve context van een methode aanspreken.** In sectie 2.4 hebben we het sleutelwoord `this` reeds aangehaald om binnen een constructor de context van een aan te maken object op te roepen. Bij uitbreiding kan je ook voor gewone methoden via `this` de context van de methode oproepen. We kunnen de methode `telBij` bijgevolg ook als volgt schrijven:

```

public void telBij(Punt anderPunt) {
    this.x += anderPunt.x;
    this.y += anderPunt.y;
}

```

**De referentie `this` is niets anders dan een referentie naar het oproepend object**, deze zal dus telkens een andere waarde krijgen afhankelijk van de plaats vanwaar de methode wordt opgeroepen.

**Vaak is het gebruik van `this` optioneel**, zoals hierboven, maar er zijn situaties waarin dit niet zo is. Denk maar aan het eerder gegeven voorbeeld van de niet-standaard constructor met **parameternamen die gelijk zijn aan de veldnamen** van de constructor-context:

Om de **verwarring voor de compiler uit te sluiten**, hebben we ook daar gebruik gemaakt van de referentie `this`. De referentie is in dit geval de nieuw te creëren context!

```

// Punt.java
public class Punt {
    private int x;
    private int y;

    public Punt(int x, int y) {

```

Beter wel  
dan niet.

```

    this.x = x;
    this.y = y;
}
...
}

```

## 2.10 Klassemethoden

De meeste methoden die we schrijven in Java zijn objectmethoden en werken in op de objecten waarmee we ze oproepen, maar soms hebben we code die eigenlijk losstaat van eender welk object. Typisch is dat vrij receptmatige code die puur op basis van de input een bepaalde output kan realiseren, code die we bij manier van spreken in eender welke klasse zouden kunnen onderbrengen. Voor dat soort code bestaan er in **Java klassemethoden of statische methoden**.

Het klassieke voorbeeld van klassemethoden zijn de methoden uit de klasse `Math`. Je kan herkennen dat het klassemethoden zijn omdat we ze via de naam van de klasse oproepen, **we moeten niet eerst een `Math`-object creëren**:

```

double macht = Math.pow(2, 3);
double r = Math.random();
double d = Math.floor(3.4);
...

```

Om zelf een klassemethode te schrijven moet je het **sleutelwoord `static`** aan de methode-signatuur toevoegen. Stel dat we frequent de eerste letter van strings in een hoofdletter willen omzetten, terwijl de rest van de letters in kleine letters worden omgezet. We zouden deze code in onze klasse `Persoon` kunnen zetten, omdat we die omzetting kunnen gebruiken voor de namen. Maar dit is even goed een methode die we ergens anders ook kunnen gebruiken. **Vermits de code niet onlosmakelijk verbonden is met een persoon** (zoals bv. de methode `isVolwassen` dat wel is) maken we er een klassemethode van:

```

// Persoon.java
public class Persoon {
    private String naam;
    private String voornaam;
    private int leeftijd;
    private boolean geslacht;

    public static String capitalize(String s) {
        if (s == null) {
            return null;
        }
    }
}

```

```

        else if (s.length() == 0) {
            return "";
        }
        else if (s.length() == 1) {
            return s.toUpperCase();
        }

        return s.substring(0, 1).toUpperCase()
            + s.substring(1, s.length()).toLowerCase();
    }
}

```

Door `static` aan de signatuur toe te voegen hebben we van `capitalize` een klassemethode gemaakt. Merk wel op dat de velden van de klasse `Persoon` niet bruikbaar zijn in de klassemethode, naam of leeftijd kan je m.a.w. niet gebruiken. Wat uiteindelijk ook logisch is, want we roepen deze methode niet op via een object maar via de naam van de klasse:

```

String s = "hoTTentoTten";
System.out.println(Persoon.capitalize(s));

```

Het voelt zelfs een beetje raar aan om `Persoon` te gebruiken om deze code uit te voeren. Indien je dergelijke code hebt die nergens echt thuishoort, kan je een afzonderlijke `Helper` klasse aanleggen met daar je klassemethoden in, als een soort 'swiss army knife' met handige code.

Nog een laatste voorbeeldje om statische methoden te illustreren. In de cursus Java Basisconcepten hebben we de `klasse Input` als een soort 'black box' gebruikt voor dynamische invoer aan de commandolijn. We hebben deze klasse gewoon gebruikt, zonder ons veel vragen te stellen over de werking ervan:

```

String s = Input.readLine();
int i = Input.readInt();
double d = Input.readDouble();
...

```

Maar ondertussen ben je al zoveel wijzer geworden, en wordt het wel eens tijd om `Input` nader onder de loupe te nemen. Wat valt er trouwens al meteen op aan de manier waarop we de `readXXX` methoden oproepen? We gebruiken de klassenaam, het moeten dan wel klassemethoden zijn. Laat ons één van de methoden in detail bekijken:

```

public static String readLine() {
    return new Scanner(System.in).nextLine();
}

```

Vaak gebruikte techniek!

En effectief, static in de signatuur. Het eerste wat de methode doet is een nieuw Scanner-object initialiseren. De klasse Scanner is een eenvoudige tekst scanner, voorzien in het `java.util` package. Aan de constructor moet de bron opgegeven worden van waaruit gelezen moet worden, `System.in` staat voor standardin, het toetsenbord m.a.w. In Scanner zijn dan allerlei methoden gedefinieerd om de ingave om te zetten naar allerlei types. De basismethode is `nextLine()` die de ingave als een `String` teruggeeft.

## 2.11 Exceptions

### 2.11.1 Try-catch

Sommige code kan aanleiding geven tot een fout tijdens de uitvoering van het programma, we spreken van een exceptie (of uitzonderlijke toestand). Om er voor te zorgen dat je programma op dat moment niet gewoon crasht, voorziet Java een try-catch constructie. In het try-blok zet je de gevoelige code, de code die de exceptie kan genereren dus. In het catch-blok komt de code die moet uitgevoerd worden indien het effectief fout loopt, deze code komt dus niet aan bod indien alles goed gaat, dan gaat de uitvoer gewoon verder na catch.

Een andere methode uit de `java.util.Scanner` klasse is `nextDouble`, die, zoals de naam doet vermoeden, de ingelezen input als double teruggeeft. Uit de Java API leren we dat deze methode een `java.util.InputMismatchException` geeft indien de ingave niet om te zetten is naar een getal, bv. als er letters in zouden voorkomen. We zouden als volgt de ingave van een gebruiker kunnen beveiligen door gebruik te maken van try-catch:

```
import java.util.*;  
  
boolean ingaveOk;  
double getal = 0.0;  
  
do {  
    ingaveOk = true;  
  
    try {  
        System.out.println("Geef een getal in:");  
        Scanner sc = new Scanner(System.in);  
        getal = sc.nextDouble();  
    }  
    catch (InputMismatchException ie) {  
        System.out.println("Ongeldige ingave. Geef ");  
    }  
}  
while (!ingaveOk);
```

```

        → opnieuw in.");
        ingaveOk = false;
    }
} while (!ingaveOk);

```

Om te weten welke code welke exceptions kan opgooien moet je zoals gezegd de Java API raadplegen. We geven nog een voorbeeld: de statische methode `parseInt` uit de klasse `Integer`. Met deze methode kan je een `String` naar een `int` converteren. Aangezien niet elke string als getal representeerbaar is, gooit die methode potentieel een `NumberFormatException` op.

### 2.11.2 Checked en unchecked exceptions

Nu bestaan er in Java exceptions die je verplicht moet afhandelen met `try-catch` (**checked exceptions**), en exceptions waar dat niet verplicht is (**unchecked exceptions**). De `NumberFormatException` behoort tot die laatste groep. Beide stukjes code die volgen zullen dus compileren:

```

// met try-catch is beter:
String s = "34a";
int getal = 0;

try {
    getal = Integer.parseInt(s);
}
catch (NumberFormatException nfe) {
    System.out.println("Kan deze String niet naar int
        → converteren");
}

// maar zonder kan dus ook:
String s = "34a";
int getal = Integer.parseInt(s);

```

Alle unchecked exceptions zijn van het type `RuntimeException`, dat is de verzamelnaam voor exceptions die te maken hebben met de Java taal zelf. Andere voorbeelden van dit soort exceptions zijn de `IndexOutOfBoundsException`, de `ClassCastException` en de `NullPointerException`. De checked exceptions daarentegen hebben allemaal te maken met externe bronnen: netwerk, bestanden, databank, enz. In appendix A wordt de klasse `TextFile` besproken. Hiermee kan je tekstbestandjes lezen en schrijven. Eén van de methoden is `read`, deze methode kan een `IOException` opgooien en aangezien dat een checked exception is ben je verplicht om met `try-catch` te werken, anders genereert de compiler een fout.

```
String content = null;
```

```

try {
    content = TextFile.read("test.txt");
}
catch (IOException ioe) {
    System.out.println("Fout bij het lezen van het \n
        →tekstbestand");
}

```

### 2.11.3 Zelf exceptions opgooien

In sommige gevallen kan het zinvol zijn om **vanuit je eigen code exceptions op te gooien**. Dit kan je doen met het **throw** sleutelwoord. Een typische situatie zijn constructoren: in sommige gevallen kunnen er argumenten aan de constructor doorgegeven worden waarmee je niets kan doen, en wat dan? Je kan natuurlijk niet zo maar om het even welke exceptie gebruiken, voor de meest courante omstandigheden bestaan er wel klassen. Zo is er de `IllegalArgumentException`, een unchecked exceptie die je zelf kan genereren vanuit een methode om aan te geven dat er onbruikbare parameters werden doorgegeven.

Zo zouden we de constructor uit de klasse `Persoon` kunnen uitbreiden met een test op de leeftijd. Je kan immers geen negatieve leeftijd hebben.

```

public Persoon(String naam, String voornaam,
               int leeftijd, boolean geslacht) {
    if (leeftijd < 0) {
        throw new IllegalArgumentException("Negatieve \n
            →leeftijd is onmogelijk");
    }
    ...
}

```

**Vanuit de oproepende code kan je een mogelijke exceptie opvangen via de try-catch constructie.**

## 2.12 Probeer het uit!

Voor deze *Probeer het Uit* gaan we een programma schrijven om vierkantsvergelijkingen op te lossen. Laat ons even het geheugen oprissen en kijken wat Wikipedia zegt over deze vergelijkingen. Vierkantsvergelijkingen zijn tweedegraadsvergelijkingen van de vorm:

$$ax^2 + bx + c = 0$$

Een belangrijk getal bij het uitrekenen van deze vergelijkingen is de discriminant, die als volgt berekend wordt:

$$D = b^2 - 4ac$$

Het aantal oplossingen van de vergelijking hangt af van de waarde van  $D$ :

- als  $D > 0$  zijn er twee reële oplossingen  $x_1$  en  $x_2$
- als  $D = 0$  zijn er twee gelijke reële oplossingen  $x_1 = x_2$
- als  $D < 0$  zijn er geen reële oplossingen

De oplossingen zelf tenslotte kunnen bepaald worden met volgende formule:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$

En nu de applicatie. We willen zoals steeds de logica en de gebruikersinteractie scheiden. Voor de logica kunnen we een klasse **Vierkantsvergelijking** schrijven met daarin de nodige functionaliteit om deze vergelijkingen uit te rekenen. Vervolgens schrijven we een klasse **Gui** om de interactie met de gebruiker te verzorgen: die moet de waarden voor  $a$ ,  $b$  en  $c$  kunnen ingeven in tekstvelden, en door te klikken op een knop worden de oplossingen (indien die er zijn) weergegeven.

We kunnen **Vierkantsvergelijking** bouwen rond 3 velden van het type **double**:  $a$ ,  $b$  en  $c$ . Denk er aan: velden maken we **private**! We hebben meteen ook een constructor voorzien voor de initialisatie:

```
// Vierkantsvergelijking.java
public class Vierkantsvergelijking {
    private double a;
    private double b;
    private double c;
```

```

public Vierkantsvergelijking(double a, double b,
    →double c) {
    this.a = a;
    this.b = b;
    this.c = c;
}
}

```

Wat willen we kunnen doen met deze vergelijkingen? De **discriminant uitrekenen** en **de oplossingen bepalen**. De methoden zijn vrij triviaal, dus we geven direct de code. We hebben voor een **double[] als return-type** voor de methode `losOp` gekozen, zo kunnen we ook twee oplossingen teruggeven indien die er zijn.

```

// Vierkantsvergelijking.java
public class Vierkantsvergelijking {
    ...
    public double zoekDiscriminant() {
        return Math.pow(b, 2) - 4 * a * c;
    }

    public double[] losOp() {
        double[] res = null;
        double d = zoekDiscriminant();

        if (d > 0) {
            res = new double[2];
            res[0] = (-b - Math.sqrt(d)) / (2 * a);
            res[1] = (-b + Math.sqrt(d)) / (2 * a);
        } else if (d == 0) {
            res = new double[1];
            res[0] = -b / (2 * a);
        }

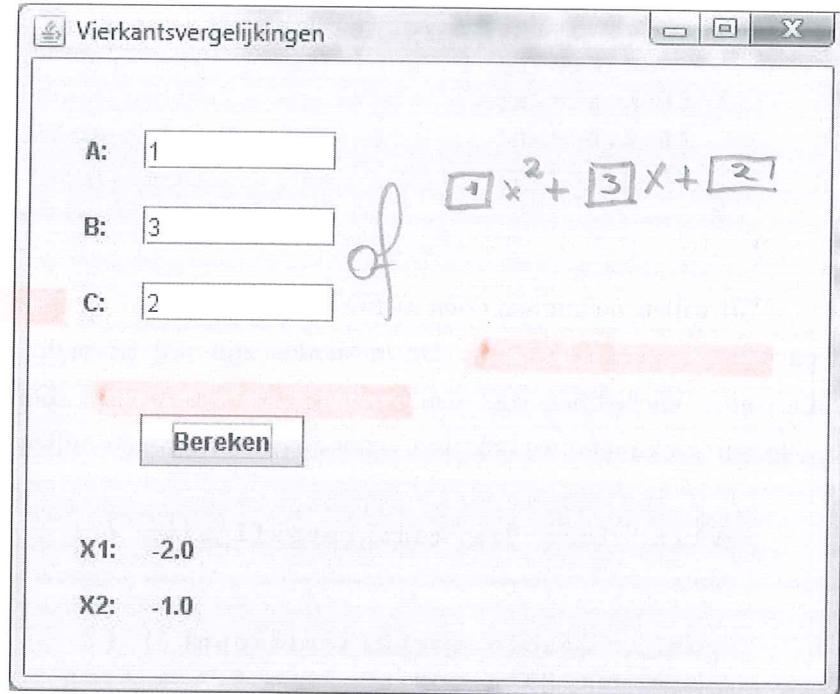
        return res;
    }
}

```

Zie  
Wiskunst!

Merk op: **indien de vergelijking geen oplossingen heeft**, zal de methode `losOp` een **null-referentie als resultaat teruggeven**. Hier zullen we op moeten letten bij het gebruik van deze methode. Dit soort eigenaardigheden van een methode beschrijf je best in de **documentatie**!

Voor de interactie met de gebruiker voegen we een nieuwe `JFrame` toe die we `GUI` noemen (zie figuur 2.6). Het enige wat we moeten doen om de GUI te laten werken is het `actionPerformed` event implementeren van de **Bereken**-knop.



Figuur 2.6: De GUI

Om de vergelijking uit te rekenen moeten we een Vierkantsvergelijking-object initialiseren met de waarden uit de tekstvakken (`jTextFieldA`, `jTextFieldB` en `jTextFieldC`). Aangezien `getText` een String teruggeeft, is een bijkomende conversie nodig naar `double` (met `Double.parseDouble()`).

De methode `losOp` geeft ons een array met de oplossingen, die we in twee labels `JLabelOplossing1` en `JLabelOplossing2` weergeven. Indien er één of geen oplossing is, geven we \*\*\* weer.

```
// in Gui.java
private void jButtonBerekenActionPerformed(...) {
    double a = Double.parseDouble(jTextFieldA.getText());
    double b = Double.parseDouble(jTextFieldB.getText());
    double c = Double.parseDouble(jTextFieldC.getText());

    Vierkantsvergelijking vkvg = new Vierkantsvergelijking(a, b, c);
    double[] oplossingen = vkvg.losOp();

    // geen oplossingen
    if (oplossingen == null) {
        jLabelOplossing1.setText("***");
    }
}
```

```

        jLabelOplossing2.setText("***");
    }
    // 1 oplossing
    else if (oplossingen.length == 1) {
        jLabelOplossing1.setText(oplossingen[0] + "");
        jLabelOplossing2.setText("***");
    }
    // 2 oplossingen
    else if (oplossingen.length == 2) {
        jLabelOplossing1.setText(oplossingen[0] + "");
        jLabelOplossing2.setText(oplossingen[1] + "");
    }
}

```

---

## 2.13 Oefeningen

1. Ontwerp een **klasse Boom**. Een boom heeft een **bepaalde hoogte**. Voorzie in je klasse volgende methoden:

- een default (boom van lengte 3m) en niet-default constructor
- methoden om de boom te doen groeien (er komt 25cm bij de hoogte bij) en te snoeien (er gaat 25cm van de hoogte af)
- een methode om de huidige hoogte van de boom op te vragen

Zorg er voor dat een boom **niet kleiner dan 25cm kan worden**, en **niet groter dan 550cm**. Voorzie **constanten** voor deze waarden in de klasse.

Voeg een **derde klasse** (naast Main en Boom dus) **Console toe** met daarin een methode **testBoom**. Initialiseer een **boom-object** van **275cm**. Laat de boom a.d.h.v. een **for-lus** 12 keer groeien en vervolgens 12 keer snoeien. Geef telkens de huidige hoogte van de boom weer.

Teken de **geheugenlayout** van dit object. (Zie p. 45)

2. Kies geschikte velden en types voor een nieuw complex type **Bankrekening**. We willen het **rekeningnummer** (in de vorm XXX-XXXXXXX-XX), het **saldo** en de **naam** van de houder bijhouden. Schrijf een gepaste constructor voor deze klasse. Geef de instructie waarmee u een nieuw object kunt aanmaken, en teken de **geheugenlayout** van dit object.

Schrijf vervolgens methoden om **geld te storten op de rekening**, en **om geld af te halen**. Merk op dat u **geen geld kunt afhalen wat niet op de rekening staat**.

Voeg een klasse **Console** toe met daarin een methode **testBankrekening**. In die methode maak je een nieuw Bankrekening-object aan op basis van de gegevens van uw eigen bankkaart. **Stort 500 euro** op de rekening en probeer **vervolgens 800 euro af te halen**. Lukt het om vanuit je code een gepaste **foutmelding** te geven?

- Schrijf een klasse **Tijd** dat de tijd in uren en minuten bijhoudt. De klasse heeft slechts één constructor die de uren en de minuten meekrijgt:

```
public Tijd(int uren, int minuten) {  
    ...  
}
```

Voeg ook een methode toe om bij de huidige tijd een andere op te tellen:

```
public void telTijdBij(Tijd t) {  
    ...  
}
```

Voeg tenslotte ook **getters toe** aan de klasse.

Schrijf een klasse **Console** met daarin een methode **testTijd**. Maak 2 **Tijd-objecten** aan: eentje met de huidige tijd, en eentje met 2u55m. Tel het tweede object bij het eerste op. Vraag de uren en de minuten van het eerste object a.d.h.v. de getters op. Welke tijd krijg je? Worden de minuten correct naar uren omgerekend? *let op max 24u! (tip modulo)*

- Implementeer alle **getters en setters voor de klasse Persoon**.
- Implementeer de klasse **MeerkeuzeVraag**. Via de accessoren **getJuisteAntwoord**, **getVraag** en **getAntwoorden** kunnen resp. het juiste antwoord, de vraag en de antwoorden opgevraagd worden. De laatste methode maakt gebruik van de private methode **randomize**, die de antwoorden-array willekeurig door elkaar haalt zodat de antwoorden iedere keer in een andere volgorde worden teruggeven (let erop dat de index van het correcte antwoord mee moet aangepast worden!). De methode **controleer** kan gebruikt worden om te controleren of een gegeven antwoord het juiste is.

*Geef als vb de termen.*

De klasse MeerkeuzeVraag
<pre>private String vraag private String[] antwoorden private int juisteIndex  public MeerkeuzeVraag(String vraag, String[] antwoorden, int juisteIndex) public boolean controleer(int antwoord) public String getJuisteAntwoord() public String getVraag() public String[] getAntwoorden() private void randomize()</pre>

Schrijf een console applicatie waarin je de gebruiker een meerkeuzevraag stelt. De gebruiker moet antwoorden door het nummer van het antwoord in te geven en krijgt dan te zien of het correct was of niet. **Herhaal de vraag zolang de gebruiker een foutief antwoord geeft.** Worden de mogelijke antwoorden iedere keer in een andere volgorde weergegeven en kan je nog steeds de vraag correct beantwoorden?

6. Voeg aan de klasse **Bankrekening** uit oefening 2 een klassemethode **isGeldigRekNr** toe waarmee voor een gegeven **String** wordt bepaald of het een geldig rekeningnummer is of niet. Waarop moet je allemaal letten bij die controle? Gebruik deze methode in de constructor van de klasse, en **genereer een `IllegalArgumentException`** i.g.v. van een ongeldig rekeningnummer.
7. Schrijf een **klassemethode concat(char[] rij1, char[] rij2)** om twee rijen aan elkaar te kleven. Breng deze methode onder in een afzonderlijke klasse **Helper**. **Let erop dat**
  - **rij1** en **rij2** referenties zijn, en **bijgevolg null kunnen zijn**
  - de methode een **nieuwe rij als resultaat moet teruggeven** met als lengte de som van de lengtes van de twee gegeven rijen
- Test uw methode uit.
8. Schrijf een **klassemethode** om een 4-digit getal te encrypteren. Ga hiervoor als volgt te werk. Vermeerder elke digit met 7 en behoud enkel de rest na deling van dat resultaat door 10. Verwissel vervolgens de eerste met de derde digit, alsook de tweede met de vierde digit. Dit geeft je de geöncrypteerde waarde. Breng uw statische methode **onder in een klasse Helper** die je in een package **algemeen** onderbrengt. Test je klassemethode door ze vanuit een andere klasse in een andere package op te roepen.

9. Implementeer de klasse `Datum` zoals aangegeven in onderstaand overzicht. In de constructor van je klasse roep je de `private methode isGeldig` op. Zo zorg je ervoor dat objecten met ongeldige datum niet aangemaakt zullen worden. De constructor genereert in dat geval een `IllegalArgumentException`.

De methode `isSchrikkeljaar` geeft een boolean terug die aangeeft of het actuele `Datum` object in een schrikkeljaar ligt (`true`) of niet (`false`).

De methode `geefWeekVanJaar` zal een getal tussen 1 en 52 teruggeven. Het is een antwoord op de vraag: In welke week van het jaar ligt het actuele `Datum` object?

De methode `geefDagVanWeek` zal een getal tussen 0 en 6 (0=zondag) teruggeven. Als je weet dat 1 januari 1900 een maandag was, dan kan je aan de hand van het aantal verstreken dagen sinds die dag bepalen welke dag van de week het actuele `Datum` object is.

De methode `geefDagVanJaar` zal een getal tussen 1 en 365 of 366 teruggeven. Het is een antwoord op de vraag: De hoeveelste dag van het jaar is het actuele `Datum` object?

De methode `geefAantalDagenInJaar` zal steeds de getalwaarde 365 of 366 teruggeven. Er zijn twee versies van deze methode:

- een `publieke niet-statische methode` heeft geen parameters en geeft van het actuele `Datum` object het totaal aantal dagen in het actuele jaar terug (365 of 366).
- een `private statische methode` bepaalt hoeveel dagen er zijn in het opgegeven jaartal. Deze versie zal je kunnen gebruiken om de correcte dag van de week te berekenen.

Test alle functionaliteit van je klasse grondig uit.

De klasse <code>Datum</code>
<code>private int dag</code>
<code>private int maand</code>
<code>private int jaar</code>
<code>public Datum(int dag, int maand, int jaar)</code>
<code>private boolean isGeldig()</code>
<code>public boolean isSchrikkeljaar()</code>
<code>public int geefWeekVanJaar()</code>
<code>public int geefDagVanWeek()</code>
<code>public int geefDagVanJaar()</code>
<code>public int geefAantalDagenInJaar()</code>
<code>private static int geefAantalDagenInJaar(int jaar)</code>

10. Implementeer de klasse **Breuk** met alle methoden zoals in onderstaand overzicht. Let in het bijzonder op de private methode **vereenvoudig**. Deze zal je gebruiken om de breuk die het resultaat is van een bewerking te vereenvoudigen.

Om een breuk te vereenvoudigen zal je de grootste gemene deler van de teller en de noemer van de breuk moeten bepalen. Voorzie een **Helper** klasse en plaats er de publieke **statische methoden ggd en kgv**, zoals aangegeven op onderstaande figuur.

Schrijf vervolgens een **GUI** waarmee je via 4 tekstvelden 2 breuken kan ingeven. Voorzie 4 knoppen, eentje voor elke bewerking, en een label om het uiteindelijke resultaat in weer te geven.

De klasse Breuk
private int teller
private int noemer
public int getTeller()
public int getNoemer()
public Breuk()
public Breuk(int teller, int noemer)
public Breuk plus(Breuk ander)
public Breuk min(Breuk ander)
public Breuk maal(Breuk ander)
public Breuk deel(Breuk ander)
private void vereenvoudig()

De klasse Helper
public static int ggd(int a, int b)
public static int kgv(int a, int b)