Java traineeship

Methods en encapsulation

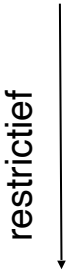
Leerdoelen

- Het bereik van variabelen definiëren
- De levenscyclus van een object uitleggen
- Methoden maken met primitief en object argumenten en retourwaarden
- Overbelaste methoden en constructors maken
- Lezen en schrijven naar objectvelden
- Methoden aanroepen op objecten
- Encapsulation principes toepassen op een klas

Scope van variabelen

Het bereik van een variabele specificeert de levensduur en de zichtbaarheid ervan. We gaan vandaag de reikwijdte van variabelen behandelen, inclusief de domeinen waarin ze toegankelijk zijn.

Dit zijn de beschikbare variabelenbereiken:



- Lokale variabelen (ook bekend als methode-lokale variabelen)
- Methode parameters (ook wel methodeargumenten genoemd)
- Instantie variabelen (ook wel attributen, velden en niet-statische variabelen genoemd)
- Statische variabelen

Locale variabelen

Lokale variabelen worden gedefinieerd binnen een methode.

Ze kunnen worden gedefinieerd in code constructies zoals if-else-constructies, for-loop-constructies of switch-instructies.

Meestal gebruik je lokale variabelen om de tussenresultaten van een berekening op te slaan. Vergeleken met de andere drie eerder genoemde variabele scopes hebben ze de kortste scope (levensduur).

```
public class BubbleSort{
    public static void main(String a[]){
    int variabele1 = 10;

    //code...

    public void mijnMethod(){
      int localeVariabele = 20;

      //code...

    }

    if(variabele1 > 5){
      boolean localeVariabele = true;

      //code...

    }

  }
}
```

Locale variabelen

Laten we eens kijken naar een voorbeeld

```
class Student {
    private double cijfer1, cijfer2, cijfer3; // instance variables
    private double hoogsteCijfer = 100;       // instance variables

    public double getGemiddelde() {
        double gem = 0;                       // Locale variable gem
        gem = ((cijfer1 + cijfer2 + cijfer3) / (hoogsteCijfer * 3)) * 100;
        return gem;
    }
    public void setGemiddelde(double waarde) {
        gem = waarde;                         // code compileert hier niet
                                              // gem is niet accesible in deze class
    }
}
```

Hoe lossen we het probleem op?

Locale variabelen

Vuistregel:

Het bereik van een **lokale variabele** hangt af van **de locatie** van zijn declaratie **binnen een methode**. De reikwijdte van lokale variabelen die zijn gedefinieerd binnen een loop-, if-else- of switchconstructie of binnen een codeblok (gemarkeerd met {}) is beperkt tot deze constructies.

Lokale variabelen die **buiten** een van deze constructies zijn gedefinieerd, zijn **toegankelijk voor de hele methode**.

Methode parameters

De variabelen die waarden in een methode-definitie accepteren, worden **methode parameters** genoemd.
Ze zijn alleen toegankelijk in de methode die ze definieert:

```
class Telefoon {
    private boolean getest;
    public void setGetest(boolean waarde) { // Methode parameter waarde is alleen toegankelijk in methode
setGetest
        getest = waarde;
    }
    public boolean isGetest() { // Variabele waarde is niet toegankelijk in methode isGetest
        waarde = false; // deze regel code zal niet compileren.
        return getest;
    }
}
```

Wat gaat hier mis?

Instantie variabelen

Instantie is een andere naam voor een **object**. Er is dus een instantie variabele beschikbaar voor de levensduur van een object.

Een instantie variabele wordt gedeclareerd binnen een klasse, buiten alle methoden. Het is toegankelijk voor alle instantie (of niet-statische) methoden die in een klasse zijn gedefinieerd.

Instantie variabelen

```
class Telefoon {
    private boolean getest; // instance variable
    public void setGetest(boolean waarde) {
        this.getest = waarde; // waarde is ook hier toegankelijk en kan worden
aangepast.
    }
    public boolean isGetest() {
        return this.getest; // Waarde kan ook teruggegeven worden.
    }
}
```

Statische variabelen

Een **Statische variabele** wordt gedefinieerd door het sleutelwoord **static** te gebruiken.

Een klasse variabele **behoort tot een klasse**, niet tot individuele objecten van de klasse.

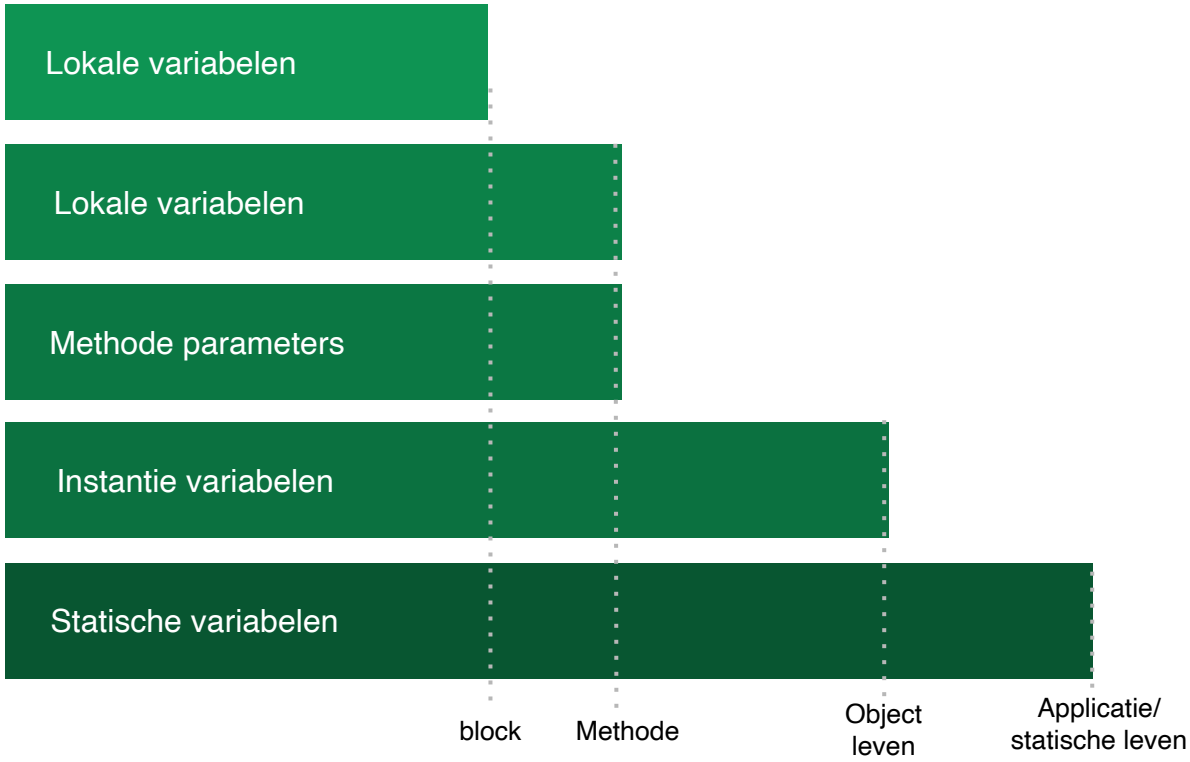
Een klasse variabele **wordt gedeeld** door alle objecten.

Objecten hebben **geen afzonderlijke kopie** van de klasse variabelen.

```java
package com.mobiel;
class TestTelefoon {
    public static void main(String[] args) {

        // Geeft toegang tot de statische variabele door de naam van de klasse te
gebruiken.
        // Het is toegankelijk zelfs voordat een van de objecten van de klasse
bestaat.
        Telefoon.isActief = false;
        Telefoon t1 = new Telefoon();
        Telefoon t2 = new Telefoon();
        System.out.println(t1.isActief); // print false
        System.out.println(t2.isActief); // print false

        // Een verandering in de waarde van deze variabele wordt weergegeven
        // wanneer de variabele wordt benaderd via objecten of klassenaam.
        t1.isActief = true;
        System.out.println(t1.isActief); // print true
        System.out.println(t2.isActief); // print true
        System.out.println(Telefoon.isActief); // print true
    }
}
```

```java
package com.mobiel;
class Telefoon {
    static boolean isActief = true;
}
```

De reikwijdte van variabelen vergeleken



De levensloop van een object

In tegenstelling tot sommige andere programmeertalen, zoals C, staat Java niet toe om zelf geheugen toe te wijzen of ongedaan te maken wanneer je objecten maakt of vernietigt.

Java beheert het geheugen voor het toewijzen van objecten en het terugwinnen van het geheugen dat wordt ingenomen door ongebruikte objecten.

De taak van het terugwinnen van ongebruikt geheugen wordt verzorgd door **Java's garbage collector**, een *thread* met lage prioriteit. Het wordt periodiek uitgevoerd en maakt ruimte vrij die wordt ingenomen door ongebruikte objecten.

De levensloop van een object

De levensloop van een object begint wanneer het is gemaakt en duurt totdat het buiten het bereik valt of niet langer wordt verwezen door een variabele.

Wanneer een object toegankelijk is, kan er door een variabele naar worden verwezen en andere klassen kunnen het gebruiken door zijn methoden aan te roepen en zijn variabelen te benaderen.

Een object is aangemaakt

Een object is aangemaakt wanneer je de trefwoordoperator **new** gebruikt. Je kan een referentievariabele initialiseren met dit object.

```
class Persoon {}
class ObjectLevensloop {
    Persoon persoon;
}
```

Let op:
In de code hierboven zijn er geen objecten van de klasse Persoon gemaakt in de klasse ObjectLevensloop.

We declareren alleen een variabele van het type Persoon. Een object wordt gemaakt wanneer een referentievariabele wordt geïnitialiseerd.

Een object is aangemaakt

Dus… Een object wordt gemaakt wanneer een referentievariabele wordt geïnitialiseerd:

```
class ObjectLevensloop2 {
    Persoon persoon = new Persoon();
}
```

Let op:
De String klasse in Java wijkt af van deze manier van aanmaken van objecten. String reference variables kunnen ook worden geïnitialiseerd door string literal values te gebruiken.

```
class ObjectLevensloop3 {
    String obj1 = new
String("obj1");
    String obj2 = "obj2";
}
```

Een object is toegankelijk

Nadat een object is gemaakt, kan het worden geopend met behulp van de referentievariabele. Het blijft toegankelijk totdat het buiten het bereik valt of de referentievariabele expliciet op **null** wordt gezet.

Als je een ander object opnieuw toewijst aan een geïnitialiseerde referentievariabele, wordt het vorige object ook ontoegankelijk vanuit die variabele.

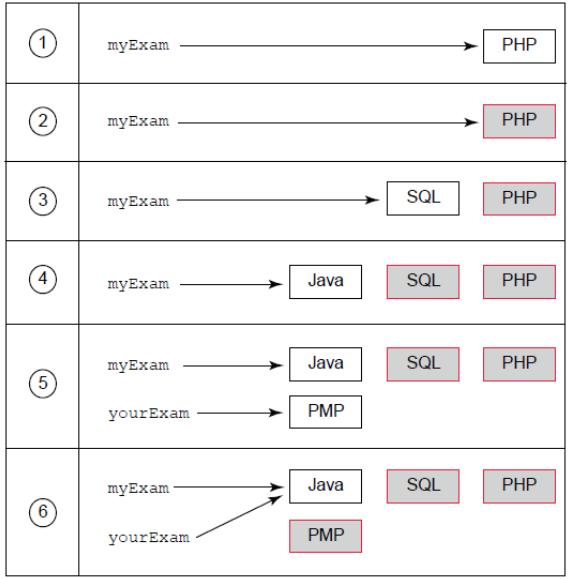
```java
class Examen {
    String naam;
    public void setNaam(String naam) {
        this.naam = naam;
    }
}

class ObjectLeven1 {
    public static void main(String[] args) {
        Examen examen = new Examen(); // object wordt aangemaakt
        examen.setNaam("Java");       // method wordt aangeroepen
        examen = null;                // object wordt verwijderd
        examen = new Examen();        // object wordt opnieuw aangemaakt
        examen.setNaam("PHP");        // method wordt aangeroepen
    }
}
```

1. We creëren een referentie variabele **examen** en initialiseren het met het object **Examen**
2. We roepen SetNaam op het object waarnaar wordt verwezen door de variabele examen
3. Door de variabele op **null** te zetten hebben geen referentie meer naar het object
1. We maken opnieuw een referentie aan en wijzen deze toe aan dezelfde variabele
2. We roepen opnieuw een methode aan.

Als **4** een ander object van de klas Examen maakt en dit toewijst aan de variabele myExam, wat gebeurt er dan met het eerste object dat door **1** is gemaakt?

Omdat het eerste object niet langer toegankelijk is met een variabele, wordt het door Java als afval beschouwd en in aanmerking genomen om door de **garbage collector** van Java naar de vuilnisbak te worden gestuurd.

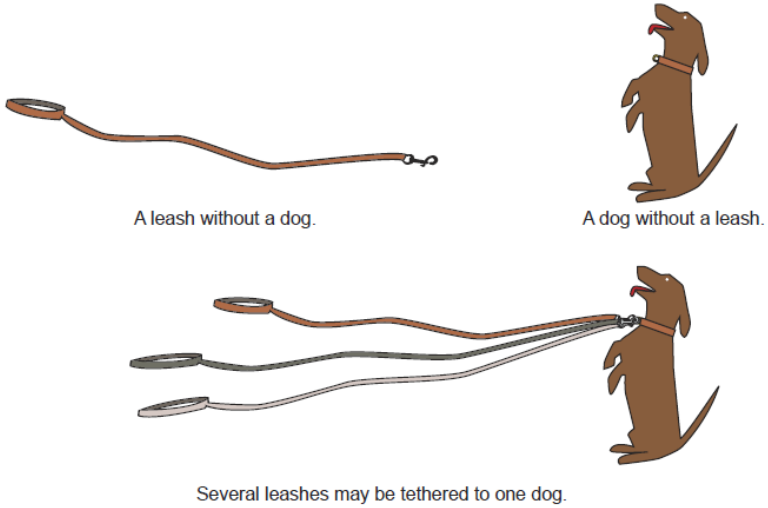
```java
class Examen {
    String naam;
    public Examen(String naam) {
        this.naam = naam;
    }
}

class ObjectLeven2 {
    public static void main(String[] args) {
        Examen mijnExamen = new Examen("PHP");
        mijnExamen = null;
        mijnExamen.setNaam("SQL");
        mijnExamen.setNaam("JAVA");

        Examen jouwExamen = new Examen("PMP");
        jouwExamen = examen
    }
}
```



Wanneer wordt een object opgeruimd?

Je kan alleen bepalen welke objecten **in aanmerking komen** voor garbage collection.

Je kan namelijk nooit bepalen **wanneer** een bepaald object wordt ingezameld. Een gebruiker kan de uitvoering van een garbage collector niet controleren of bepalen. Het wordt bestuurd door de JVM.



De scope van variabelen

Testje:

Voorspel de uitkomst van het onderstaande Java programma:

```
public class Test {
    static int x = 11;
    private int y = 33;
    public void method1(int x) {
        Test t = new Test();
        this.x = 22;
        y = 44;

        System.out.println("Test.x: " + Test.x);
        System.out.println("t.x: " + t.x);
        System.out.println("t.y: " + t.y);
        System.out.println("y: " + y);
    }
```

```
public static void main(String args[]) {
        Test t = new Test();
        t.method1(5);
    }
}
```

Methodes

Dan gaan we nu over naar het gebruik van methodes. Je hebt deze hiervoor al zien langskomen, maar nu zullen we wat verder inzoomen op hoe ze werken.

```
class Telefoon {
    public boolean stuurSMS(String nummer, String bericht) {
       boolean berichtStatus = false;
       if (verstuurd(nummer, bericht)) {
           berichtStatus = true;
       }
        return berichtStatus;
    }

    // De rest van de code van de Telefoon klasse.
}
```

In een methode geven we allereerst een return waarde mee. Vervolgens geven we parameters mee. We kunnen ook *geen* parameters meegeven.

Verder kunnen we binnen een methode ook andere methodes aanroepen.

We sluiten de methode altijd af met een **return** statement.

Parameters

Methodeparameters zijn de variabelen die voorkomen in de definitie van een methode en specificeren het type en aantal waarden dat een methode kan accepteren.

Als je nog niet weet hoeveel parameters je mee wilt geven heb je daar een handig trucje voor in Java:

Het ellipsis (...) dat volgt op het gegevenstype geeft aan dat de methodeparameter dagen een array of meerdere door komma's gescheiden waarden kan worden doorgegeven.
Let op: Deze type parameter kan alleen als achterste parameter.
`public int vakantieDagen(int... dagen, String jaar)`
Zal niet compileren

```
class Werknemer {
    public int vakantieDagen(int... dagen) {
        int totaal = 0;
        for (int dag : dagen) {
            totaal += dag;
        }
        return totaal;
    }
}
```

De return statement

Een return-statement wordt gebruikt om een methode te verlaten, met of zonder waarde. Voor methoden die een retourtype definiëren, moet de return-instructie onmiddellijk worden gevolgd door een returnwaarde.

Voor methoden die geen waarde retourneren, kan de return-instructie worden gebruikt zonder een returnwaarde om een methode af te sluiten.

```
double berekenGemiddelde(int a, int b, int c)
{
    int gemiddelde;
    gemiddelde = (a + b + c) / 3;
    return gemiddelde;
}
```

```
double berekenGemiddelde(int a, int b, int c) {
    return (a + b + c) / 3;
}
```

Beide manieren zijn goed hier. Enkel is de tweede methode efficienter

De return statement

Hier zijn enkele punten om op te letten bij het definiëren van een return statement:

- Voor een methode die een waarde retourneert, moet de return-instructie onmiddellijk worden gevolgd door een waarde.
- Voor een methode die geen waarde retourneert (retourtype is void), mag de return-instructie niet worden gevolgd door een retourwaarde.
- Als de compiler vaststelt dat een return-instructie niet de laatste instructie is die in een methode wordt uitgevoerd, kan de methode niet worden gecompileerd.

Oefenen met methodes

Opdracht:

Schrijf een methode genaamd `exponent(int base, int exp)` die een int waarde teruggeeft tot de macht van exp.

```
Geef de basis: 3
Geef de exponent: 4
3 tot de macht 4 is: 81
```

Opdracht:

Schrijf een booleaanse methode met de naam isOdd() in een klasse met de naam OddEvenTest, die een int als invoer neemt en true retourneert als deze oneven is.

Method overloading

Function overloading (of method overloading in Java) is het hebben van meerdere definities van dezelfde functie / methode binnen een klasse.

Aan de hand van de argumenten zoekt Java de definitie die het ‘beste’ past bij de aanroep. Bijvoorbeeld:

```
static int max(int n1, int n2) {
    if (n1 > n2) {
        return n1;
    } else {
        return n2;
    }
}
```

```
static double max(double d1, double d2) {
    if (d1 > d2) {
        return d1;
    } else {
        return d2;
    }
}
```

Voorbeeld aanroep

Als de types van de argumenten exact overeen komen, is het eenvoudig te bepalen welke implementatie uitgevoerd zal worden.

```java
static int max(int n1, int n2) {
    if (n1 > n2) {
        return n1;
    } else {
        return n2;
    }
}
```

```java
static double max(double d1, double d2) {
    if (d1 > d2) {
        return d1;
    } else {
        return d2;
    }
}
```

```java
int i;
double d;
i = max(1, 2);
d = max(1.0, 2.0);
d = max(1.0, 2); // welke?
```

Java zal altijd **upcasten** van primitieve datatypen toepassen om een match te vinden

Dubbelzinnige method overloading

```
public class TestMethodOverload {
    static public void main(String[] args) {
        int i;
        i = max(1, 2.0); // Roep max(int, double) aan
        i = max(1.0, 2); // Roep max(double, int) aan
        i = max(1, 2);   // dubbelzinnig
    }

    static int max(int i, double d) {
        if (i > d) return i
        else return (int) d;
    }

    static int max(double d, int i) {
        if (i > d) return i;
        else return (int) d;
    }
}
```

In dit geval zal de compiler klagen over ambiguiteit

Method overloading met verschillend return type?

```java
public class Opgave1 {
    public static void main(String[] args){
        methode1(2);
        methode1(3);
    }
    static int methode1(int i) {
        return i + 1;
    }
    static void methode1(int i) {
        System.out.println(i);
    }
}
```

```
$ javac Opgave1.java
Opgave1.java:9:methode1(int)is alreadydefined in
Opgave1

static void methode1 (int i) {
            ^

1 error
```

Method Overloading

Quiz:

Wat is de uitvoer van de onderstaande Code?

Wat gebeurt er wanneer we casten naar: int, float, double, long & char?

```
public class MainClass {
   static void method(Integer i) {
        System.out.println(1);
    }

    static void method(Double d) {
        System.out.println(2);
    }

    static void method(Number n) {
        System.out.println(4);
    }

    static void method(Object o) {
        System.out.println(5);
    }

    public static void main(String[] args) {
        method((short)12);
    }
}
```

Constructors

Met behulp van een constructor kunnen we gegevens in het object laden direct nadat het is gemaakt.

Hiervoor maken we een methode aan die dezelfde naam het als het object zelf, en geen return type heeft.

Constructors

```
class Punt {
    double x, y;

    public Punt(double getal1, double getal2) {
        x = getal1;
        y = getal2;
    }
}
```

En dan kun je een Punt object maken op deze manier:

```
Punt p1 = new Punt(1, 2);
```

Standaard constructor vervalt

Als we zelf een constructor definiëren vervalt de standaard constructor van Java

```
Error: constructor Punt in class Punt cannot be applied to given types;
        Punt punt = new Punt();
```

Constructors zonder argumenten

```
class Punt {
    double x, y;

    public Punt() {
        x = 12;
        y = 13;
    }
}
```

En dan zal ieder nieuw punt-object standaard 12 en 13 voor de x en y waarde krijgen.

```
Punt p1 = new Punt();
```

Standaardwaarden voor klasse en instantievariabelen

```
class Punt {
    double x = 12;
    double y = 13;
}
Punt p1 = new Punt();
```

Dit zou overigens hetzelfde effect hebben…

Overschaduwing van instantievariabelen

Zoals we net al zagen hebben we wel eens dat we parameters dezelfde naam willen geven als de instantievariabelen:

```
class Punt {
    double x, y;
    public Punt(double x, double y)
{
        x = x;
        y = y;
    }
}
```

Maar in dit geval overschaduwen de parameters de instantievariabelen :(

Overschaduwing oplossen met behulp van het this keyword

Dit kunnen we oplossen door de speciale variabele **this**. Welke een referentie naar het eigen object bevat.

```
class Punt {
    double x, y;
    public Punt(double x, double y)
{
        this.x = x;
        this.y = y;
    }
}
```

Constructor overloading

Net zoals we meerdere definities van dezelfde methode kunnen hebben, kunnen we ook meedere definities van een constructor hebben:

Contrcutor overloading

```
class Punt {
    double x, y;
    public Punt(double x, double y)
{
        this.x = x;
        this.y = y;
    }
    public Punt(int x, int y) {
        this.x = x + 0.5;
        this.y = y + 0.5;
    }
}
```

Constructor chaining

- Vaak lijken constructoren erg op elkaar en zorgt de strategie van de vorige slide voor veel dubbele code.
- Dit kunnen we oplossen door in een constructor een andere constructor aan te roepen.
- Op deze manier ketenen we constructoren aan elkaar vast: **constructor chaining**.
- Een andere implementatie van de constructor kun je aanroepen met behulp van: *this(parameters)*.

Constructor chaining (2)

```
class Punt {
    double x, y;
    public Punt(double x, double y)
{
        this.x = x;
        this.y = y;
    }
    public Punt(int x, int y) {
        this(x + 0.5, y + 0.5);
    }
}
```

Constructor chaining (2)

Als je gebruik maakt van constructor chaining dan ben je verplicht de andere constructor direct aan te roepen als eerste commando van de constructor:

```
class Punt {
    double x, y;
    public Punt(double x, double y)
{
        this.x = x;
        this.y = y;
    }
    public Punt(int x, int y) {
        double newX = x + 0.5;
        this(x + 0.5, y + 0.5);
    }
}
```

Error: call to this must be first statement in constructor

Constructor chaining

Quiz: Wat is de uitvoer van deze code?

```java
class A {
    public A(int i) {
        System.out.println(1);
    }

    public A() {
        this(10);
        System.out.println(2);
    }

    void A() {
        A(10);
        System.out.println(3);
    }

    void A(int i) {
        System.out.println(4);
    }
}

public class MainClass {
    public static void main(String[] args) {
        new A().A();
    }
}
```

Information hiding

Als we voorheen een object aanmaakte dan was het mogelijk om alle variabelen en methoden daarvan uit te lezen, te veranderen of aan te roepen.

```
class TestObject {
    int i = 3;
}

TestObject obj = new TestObject();
System.out.println(obj.i);
```

Dit is alleen niet altijd wenselijk, het is conceptueel gezien beter om te verbergen hoe dingen in een klasse/object worden opgeslagen: **information hiding.**

Information hiding

Bij onderstaande objecten bijvoorbeeld is voor twee verschillende manieren gekozen om te representeren of iets wel of niet vertrouwelijk is.

```
class Document {
    boolean vertrouwelijk = true;
}

class Document {
    String vertrouwelijk = "zeer vertrouwelijk";
}
```

Stel dat we om wat voor een reden dan ook een 3e staat willen toevoegen en daarom van de boolean een String willen maken, moet op alle plekken waar ”vertrouwelijk” wordt gebruikt de code worden aangepast.

Information hiding

Een manier om dit op te lossen is door deze instantievariabele **Private** te maken. Dan is deze niet meer van buitenaf benaderbaar.

```
class Document {
    private String vertrouwelijk = "zeer vertrouwelijk";
}

// verderop:
new Document().vertrouwelijk.equals(..);
```

```
error: vertrouwelijk has private access in Document
      System.out.println(obj.vertrouwelijk);
1 error
```

Information hiding

Als we nu alsnog willen dat de programmeur kan aanpassen of het wel of niet vertrouwelijk is, kunnen we **public** een **getter** en **setter**-methode implementeren:

```
class Document {
    private String vertrouwelijk = "zeer vertrouwelijk";
    public boolean getVertrouwelijk() {
        return ! vertrouwelijk.equals("openbaar");
    }
    public void setVertrouwelijk(boolean isVertrouwelijk) {
        geheim = isVertrouwelijk ? "zeer vertrouwelijk" : "openbaar";
    }
}
```

Dus: Toegang tot objectvelden krijgen

Een objectveld is een andere naam voor een instantievariabele gedefinieerd in een klasse.

```
class Ster {
    double leeftijd; // instantie variabele
    public void setLeeftijd(double nieuweLeeftijd) { // setter methode
        leeftijd = nieuweLeeftijd;
    }
    public double getLeeftijd() { // getter methode
        return leeftijd;
    }
}
```

Toegang tot objectvelden krijgen

In het voorgaande voorbeeld definieren we eerst een instantievariabele, **sterleeftijd**.

Hierna definieren we een settermethode, **setLeeftijd**. Een setter (of mutator) methode wordt gebruikt om de waarde van een variabele in te stellen.

Vervolgens definieert een getter (of accessor) methode, **getLeeftijd**. Een gettermethode wordt gebruikt om de waarde van een variabele op te halen.

In dit voorbeeld is het objectveld **sterLeeftijd**, niet **leeftijd** of **nieuwLeeftijd**. De naam van een objectveld wordt niet bepaald door de naam van de getter- of settermethoden.

Objecten en primitieven doorgeven aan methoden

Laten we ten slotte eens kijken naar het verschil tussen het doorgeven van object referenties en primitieven aan een methode. We beginnen met het doorgeven van primitieven (wat zijn dat ook al weer?) aan methoden.

De waarde van een primitief datatype wordt gekopieerd en doorgegeven aan een methode. Daarom verandert de variabele waarvan de waarde is gekopieerd niet:

Primitieven doorgeven aan methoden

De waarde van een primitief datatype wordt gekopieerd en doorgegeven aan een methode. Daarom verandert de variabele waarvan de waarde is gekopieerd niet:

```
class Werknemer {
    int leeftijd;
    void pasLeeftijdAan(int a) {
        a = a + 1;
        System.out.println(a);
    }
}

class Office {
    public static void main(String args[]) {
        Werknemer w = new Werknemer();
        System.out.println(w.leeftijd);
        w.pasLeeftijdAan(w.leeftijd);
        System.out.println(w.leeftijd);
    }
}
```

```
0
1
0
```

Object referenties doorgeven aan methoden

Over het algemeen zijn er twee gevallen waar we rekening mee moeten houden:

1. Wanneer een methode de objectverwijzing die eraan is doorgegeven opnieuw toewijst aan een andere variabele.
2. Wanneer een methode de status wijzigt van de objectverwijzing die eraan is doorgegeven

Laten we eerst naar het eerste geval kijken

Object referenties doorgeven aan methoden

Wanneer je een objectverwijzing doorgeeft aan een methode, kan de methode deze aan een andere variabele toewijzen. In dit geval blijft de staat van het object, die aan de methode is doorgegeven, intact.

Wanneer aan een methode een referentiewaarde wordt doorgegeven, wordt een kopie van de referentie (**dat wil zeggen het geheugenadres**) doorgegeven aan de aangeroepen methode.

De aanroeper kan doen wat hij wil met zijn kopie zonder ooit de originele referentie te veranderen.

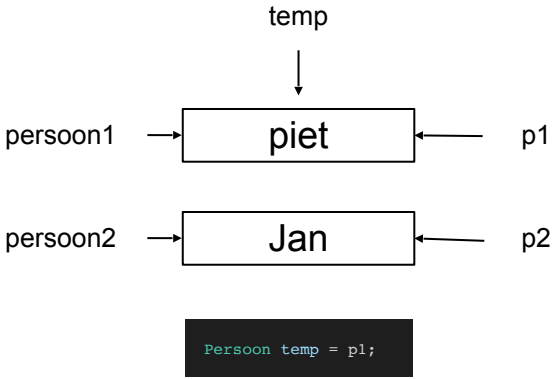
Object referenties doorgeven aan methoden

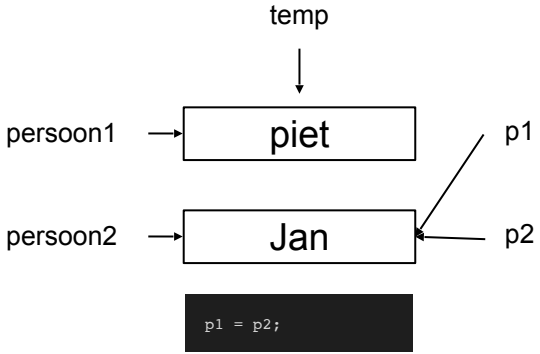
```
class Persoon {
    private String naam;
    Persoon(String nieuweNaam) {
        naam = nieuweNaam;
    }
    public String getNaam() {
        return naam;
    }
    public void setNaam(String nieuweNaam) {
        naam = nieuweNaam;
    }
}
```

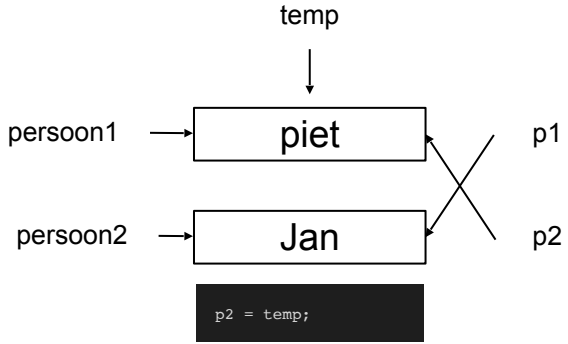
```
class Test {
    public static void wissel(Persoon p1, Persoon p2) {
        Persoon temp = p1;
        p1 = p2;
        p2 = temp;
    }

    public static void main(String args[]) {
        Persoon p1 = new Persoon("Piet");
        Persoon p2 = new Persoon("Jan");
        System.out.println("Voor wissel: " + p1.getNaam() + " " +
p2.getNaam());
        wissel(p1, p2);
        System.out.println("Na wissel: " + p1.getNaam() + " " + p2.getNaam());
    }
}
```

Uitkomst niet wat je verwachtte? Laten we eens kijken hoe het werkt…

Object referenties doorgeven aan methoden







Zoals je in figuur hierboven kunt zien, verwijzen de referentievariabelen persoon1 en persoon2 nog steeds naar de objecten die ze aan de methode swap hebben doorgegeven. Omdat er geen wijziging is aangebracht in de waarden van de objecten waarnaar wordt verwezen door de variabelen persoon1 en persoon2

We printen daarom tweemaal eerst Piet en dan Jan

Object referenties doorgeven aan methoden

Laten we eens kijken hoe een methode de status van een object kan veranderen, zodat de gewijzigde status toegankelijk is in de aanroepende methode. Laten we dezelfde code gebruiken als bij het andere voorbeeld

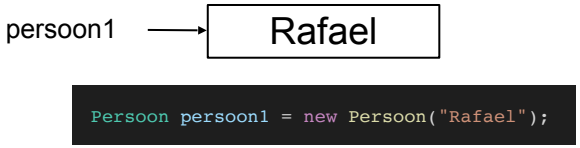
Object referenties doorgeven aan methoden

```
class Persoon {
    private String naam;
    Persoon(String nieuweNaam) {
        naam = nieuweNaam;
    }
    public String getNaam() {
        return naam;
    }
    public void setNaam(String nieuweNaam) {
        naam = nieuweNaam;
    }
}
```

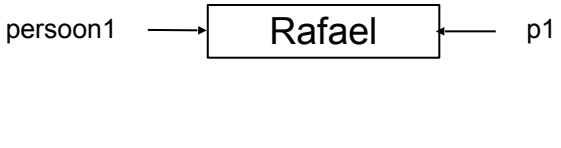
```
class Test {
    public static void resetNaam(Persoon p1) {
        p1.setNaam("Rodrigo");
    }

    public static void main(String[] args) {
        Persoon persoon1 = new Persoon("Rafael");
        System.out.println(persoon1.getNaam());
        resetNaam(persoon1);
        System.out.println(persoon1.getNaam());
    }
}
```

Uitkomst niet wat je verwachtte? Laten we eens kijken hoe het werkt…

Object referenties doorgeven aan methoden




```
Persoon persoon1 = new Persoon("Rafael");
```




```
p1.setNaam("Rodrigo");
```

De methode `resetNaam` accepteert het object waarnaar persoon1 verwijst en wijst het toe aan de methodeparameter p1.

Nu verwijzen beide variabelen, person1 en p1, naar hetzelfde object. `p1.setNaam("Rodrigo")` wijzigt de waarde van het object waarnaar wordt verwezen door de variabele p1.

Omdat de variabele persoon1 ook naar hetzelfde object verwijst, retourneert `persoon1.getNaam()` de nieuwe naam, Rodrigo, in de methode main.

```
$ java Eindopdracht_hoofdstuk3
Geef een natuurlijk getal: 5
De eerste 5 Lucas-getallen:
2 1 3 4 7

$ java Eindopdracht_hoofdstuk3
Geef een natuurlijk getal: -6
Getal negatief, fout

$ java Eindopdracht_hoofdstuk3
Geef een natuurlijk getal: 60
Getal te groot, past niet
```

Eindopdracht

Het programma vraagt de gebruiker een natuurlijk getal n en print vervolgens de eerste n zogenaamde Lucas-getallen. Deze getallen worden gegeven door de reeks:

2 1 3 4 7 11 18 …

Het eerste Lucas-getal is 2, het tweede is 1. Daarna krijg je het volgende getal telkens door de twee voorgaande getallen bij elkaar op te tellen. In je programma moet je testen of het door de gebruiker ingetypte getal wel positief is. Verder kunnen de getallen van de Lucas-reeks zo groot worden dat ze niet meer passen in een **int**.

Bouw in je programma een test in, zodat bij een te grote waarde van n niets geprint wordt.

Leerdoelen

- ✓ Het bereik van variabelen definiëren
- ✓ De levenscyclus van een object uitleggen
- ✓ Methoden maken met primitief en object argumenten en retourwaarden
- ✓ Overbelaste methoden en constructors maken
- ✓ Lezen en schrijven naar objectvelden
- ✓ Methoden aanroepen op objecten
- ✓ Encapsulation principes toepassen op een klas

Vragen?

- E-mail mij op joris.vanbreugel@code-cafe.nl!

- Join de Code-Café community op discord!