Java summary: Basics

Comments

// Everything to the end of the line is ignored.
/* Everything (possibly many lines) is ignored until a */
/** Used for automatic HTML documentation generation by the javadoc program. */

Types

Types are divided into two categories. There are 8 ***primitive*** *types: boolean (true and false); char; byte, short, int, and long (integers); float and double (floating point).* ***Object*** *types are created whenever a class is defined.* ***Arrays*** *are objects.*

Integer Types

Type	Bytes	Range	Literals
byte	1	-128..+127	none
short	2	-32,768..+32,767	none
int	4	-2,147,483,648..+2,147,483,647	23, 0xAF
long	8	-9,223,372,036,854,775,808..	23L, 0xAFL

Operators: arithmetic, comparison, bitwise, assignment.

Floating-point Types

The floating-point types are float and double. Calculations may produce NaN (Not a Number, for example after a division by zero) or +/- infinity.

Type	Bytes	Range	Accuracy	Literals
float	4	-3.4E38..+3.4E38	6-7 digits	3.14F 6.02e23F
double	8	-1.7E308..+1.7E308	14-15 digits	3.14 6.02e23

Operators: arithmetic, comparison, assignment.

char Type

char type is stored as an unsigned number (integer type) in two bytes (range 0..65,535).
Literals:

- 'A' (single character)
- Unicode '\uxxxx' where x is a hexadecimal digit. *Eg* '\u0041'
- *Escape* combinations: '\n' newline, '\\' backslash, '\'' single quote, '\"' double quote, '\r' carriage return, '\t' tab, '\b' backspace, '\f' form feed.

boolean Type

boolean is used to store the values ***true*** *or* ***false****. The storage is unspecified.*
Operators: logical, ==, !=, assignment.

Identifiers (variable, class or method names)

- Start identifiers with an alphabetic character (a-z or A-Z), and continue with alphabetic, numeric (0-9), or '_' (underscore) characters. Do not use $.
- Do not use keywords (see below).

Standard Naming:

- Class and Method names should start with an uppercase letter
- Variable names should start with a lowercase letter, start class attributes with *m* and class attributes with an *s*.
- Constants should be all uppercase with underscores between words.
- Second words in a name should start with an uppercase letter.

Keywords (defining the Java language)

assert boolean break byte case catch char class const continue default do double else extends false final finally float for goto if implements import instanceof int interface long native new null package private protected public return short static strictfp super switch synchronized this throw throws transient true try void volatile while

Variables

Variables may be *local*, *attributes* (field), or *class variables*. Parameters are local variables that are assigned values when the method is called.

	local in method	**attribute**	**class**
Where declared	In a method.	In class, but not in a method.	In class using `static` keyword.
Initial value	Must assign a value before using.	Zero for numbers, null for objects, false for boolean. Can be initialized in constructor.	Zero/null/false or initialized with static initializer.
Visibility	Only in the same method.	All methods in the same package. If it is declared `public`, other classes can see it. If it is `protected`, only child classes, and for `private`, only this class.	Same as fields.
Lifetime	Created when the method is called. Destroyed when the method returns.	Created when an instance of the class (object) is created with `new`. Destroyed after there are no more references to the object (garbage collection).	Program lifetime.

Java summary: Expressions

Parentheses () have three uses:

1. Grouping to control order of evaluation, or for clarity.
   Eg, (a + b) * (c - d)
2. After a method name to enclose parameters. Eg, x = sum(a, b);
3. Around a type name to form a *cast operator*. Eg, i = (int)x;

Order of evaluation

1. Higher precedence are done before lower precedence (see table).
2. Left to right among equal precedence except: unary, assignment and conditional operators.

Abbreviations used below

i, j - integer (int, long, short, byte, char).
m, n - numeric values (integers, double, or float).
b, c - boolean; x, y - any primitive or object type.
s, t - String; a - array; o - object; co - class or object

Operator Precedence

`. [] (args) post ++ --`
`! ~ unary + - pre ++ --`
`(type) new`
`* / %`
`+ -`
`<< >> >>>`
`< <= > >= instanceof`
`== !=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`= += -= etc`
	Remember only:
	1. unary operators
	2. * / %
	3. + -
	4. *comparisons*
	5. && ||
	6. = *assignments*
	Use () for all others

Arithmetic Operators

The result of arithmetic operators is double if either operand is double, else float if either operand is float, else long if either operand is long, else int.

n + m	Addition. *Eg* 7+5 is 12, 3 + 0.14 is 3.14
n - m	Subtraction
n * m	Multiplication. *Eg* 3 * 6 is 18
n / m	Division. *Eg* 3.0 / 2 is 1.5 , 3 / 2 is 1
n % m	Remainder (Mod) after division of n by m. *Eg* 7 % 3 is 1
++i	Add 1 to i *before* using the value in the current expression
--i	As above for subtraction
i++	Add 1 to i *after* using the value in the current expression
i--	As above for subtraction

Comparing Primitive Values

The result of all comparisons is boolean (true or false).

== != < <= > >=

Logical Operators

The operands must be boolean. The result is boolean.

b && c	Conditional "and". true if both operands are true, otherwise false. Short circuit evaluation. *Eg* (false && anything) is false.
b || c	Conditional "or". true if either operand is true, otherwise false. Short circuit evaluation. *Eg* (true || anything) is true.
!b	true if b is false, false if b is true.
b & c	"And" which always evaluate both operands (*not* short circuit).
b | c	"Or" which always evaluate both operands (*not* short circuit).
b ^ c	"Xor" Same as b != c

Assignment Operators

=	Left-hand-side must be a variable (*an lvalue*).
+= -= *= …	All binary operators (except && and ||) can be combined with assignment. *Eg* a += 1 is the same as a = a + 1

Bitwise Operators

Bitwise operators operate on bits of integers. Result is int.

i & j	Bitwise AND: 1 if both bits are 1. 5 & 3 is 1 (5=101, 3=011).
i | j	Bitwise OR: 1 if either bit is 1. 5 | 3 is 7 (7=111).
i ^ j	Bitwise EXOR: 1 if bits are different. 5 ^ 3 is 6 (6=110).
~i	Bitwise NOT: 0 -> 1, 1 -> 0
i << j	Bits in i are shifted j bits to the left, zeros inserted on right. 5 << 2 is 20.
i >> j	Bits in i are shifted j bits to the right. Sign bits inserted on left. 5 >> 2 is 1.
i >>> j	Bits in i are shifted j bits to the right. Zeros inserted on left.

Casts

(*t*)x	Casts variable x to type t

Java summary: Statements

Control statements contain *blocks* of statements enclosed in curly braces {}. If only 1 statement, the braces may be omitted.

if Statement

```
//----- if statement with only a true clause
   if (expression) {
       statements
   }
//----- if statement with true and false clause
   if (expression) {
       statements
   }else{
       statements
   }
//----- if statements with many parallel tests
   if (expression1) {
       statements
   }else if (expression2) {
       statements
   }else if (expression3) {
       statements
   . . .
   }else {
       statements // if no expression was true
   }
```

switch Statement

The effect of the **switch** statement is to choose some statements to execute depending on the *integer value* of an expression. The **break** statement exits from the switch statement. If there is no break at the end of a case, execution continues in the next case (fall-through)!!

```
   switch (expr) {
      case c1:
            statements
            break;
      case c2:
            statements // if expr == c2
                       // no break, so the next statements are also executed!!
      case c2, c3, c4:  // multiple values
            statements
            break;
      . . .
      default:
            statements // if none of above
   }
```

while

```
   while (testExpression) {
       statements
   }
```

for

The for first executes the `initialStmt`, then the `testExpr`, if true executes the statements. Before testing the `testExpr` again, it executes the `incrementStmt`.

```
   for (initialStmt; testExpr; incrementStmt) {
       statements
   }
```

do

do {
statements
} **while** (*testExpression*);

Other loop controls

```
break;       //exit innermost loop or switch
break label; //exit from loop label
continue;    //start next loop iteration
continue label; //start next loop label
```

Label a statement:

```
outerloop:
for (...)
  for ()
     break outerloop;  // exits the outer for loop
```

Method Return

```
return;      //exits a void method
return expr; //exits method and returns value
```

try...catch exceptions (exception handling)

```
try {
    . . . // statements that might cause exceptions
}catch (exception-type x) {
    . . . // statements to handle exception
}
```

Java summary: Arrays & ArrayLists

For storing multiple values of the same type. For arrays, the size must be specified at creation; while ArrayLists dynamically extend their size when necessary.

Declaring an array

- Example: `int[] a;`
- Is a reference, like an object

Array allocation

- Allocating memory for an array
  - Example: `a = new int[100]; (fixed size)`
  - an array is an object
- all elements are set to zero (numeric), null (object types) or false (boolean)
- Anonymous array -- Example: `int[] a = new int[] {1,2}; -> array of length 2`
  - `String[] words = new String[]{"bla", "blo", "bli"};`

Accessing elements

with Subscripts
-- Example: `a[i]`

- Subscript ranges always start at zero
- Java always checks Subscript legality, between 0 and size. If not, Java throws ArrayIndexOutOfBoundsException

Array length & iteration

-- Example: `a.length`

- typical statement for looping over an array

```
for (int i = 0;i < a.length;i++){
   a[i]=10;
}
```

- Alternative for looping over an array when value should not be changed

```
for (int x : a){
   System.out.println(x+", ");
}
```

Two-dimensional arrays

- `int[][] a = new int[2][3];  // Two rows and three columns.`
- a[1][1] = 5;

ArrayList

- For dynamically extendable arrays: use *ArrayLists* (or *Vector* which is the same) of the java.util package.

```
ArrayList<int> v = new ArrayList<int>();
v.add(2);
for(int i = 0;i < v.size(); i++){
  System.out.println("Element " + i + "=" + v.get(i));
}
```

- Alternative iteration also works:

```
for(int x: v){
   System.out.println(x+", ");
}
```

Java's spelregels

Deze moet je kennen voor het theorie-examen.

0. **Java** bestaat enkel uit *klasses* [`class`].
1. Een **klasse** bestaat uit *attributen* en *methodes*.
2. Een **attribuut** is van het enkelvoudige type [`int, float, double, boolean, char`] of het is een object van een welbepaalde klasse (attribuut is dan een object).
   - Een niet-ingevulde object-variabele heeft de waarde `null`.
3. Een **methode** is zoals een procedure (of ook functie genoemd) met een *header* ('eerste lijn' van de methode, met zijn *parameters*) en een *implementatie*.
   - Er zijn 2 speciale methodes: de *constructor* (met zelfde naam als klasse) die opgeroepen wordt bij 't aanmaken van een object en de `main` (nooit meer dan 1 per klasse) die het begin van een programma aangeeft.
   - Je mag meerdere methodes met dezelfde naam hebben, zolang het type of aantal parameters verschilt
4. Een **object** is een *instantiatie* van een klasse [`new`], waarbij elk object eigen waarden voor elk attribuut heeft. Bij het instantiëren wordt de constructor van de klasse uitgevoerd.
   - De **default constructor** heeft geen parameters en een lege implementatie. Als er geen andere constructor bestaat, wordt de default constructor automatisch door Java aangemaakt.
   - Met `super(…)` op de *eerste lijn van je constructor* roep je één van de constructors van de superklasse op. Indien afwezig, wordt de default constructor opgeroepen. Deze moet dan wel bestaan voor de superklasse.
   - Op de *eerste lijn van een constructor* kan met `this(…)` een andere constructor van dezelfde klasse opgeroepen worden.
5. De **implementatie** van een methode bestaat uit *statements* en *lokale variabelen* (enkelvoudig of objecten).
   - Een **lokale variabele** bestaat enkel binnen een methode, net zoals de parameters van de methode.
   - De statements hebben toegang tot alle attributen en methodes van het eigen object en die van object variabelen via de '`.`' operator.
6. Een klasse kan **erven** van 1 concrete of abstracte *superklasse* [`extends`].
   - De *subklasse* erft alle attributen en methodes van de superklasse over.
   - De *subklasse* mag attributen en/of methodes toevoegen.
   - In de constructor roep je met super(x) op de eerste lijn de constructor van de superklasse op. Als je die weg laat, wordt de default constructor opgeroepen (alsof er super()) zou staan. Deze moet natuurlijk wel bestaan.
   - Methodes met dezelfde header als een methode van de superklasse *overschrijven* die methode, bij een object zal deze nieuwe methode opgeroepen worden ipv. die van de superklasse. De methode van de superklasse roep je op met `super.m()`.
7. Een **abstracte methode** heeft *geen implementatie* (deel tussen accolades), enkel een header [`abstract`]. De eerste lijn eindigt met een punt-komma.
8. Een **interface** is een klasse met *enkel abstracte methodes* [`interface`].
   - Een interface heeft geen constructor en geen attributen (al zijn statische attributen wel mogelijk).
   - Een klasse mag meerdere interfaces als superklasse hebben [`implements`]: "de klasse implementeert de interface".
   - Een concrete klasse moet alle abstracte methodes implementeren. Anders blijft het een abstracte klasse en kan je er geen objecten van maken.
9. Een **abstracte klasse** heeft minstens 1 *abstracte methode* [`abstract`]
   - Een abstracte klasse zit tussen een gewone klasse en een interface in: ze heeft zowel concrete methodes en attributen als abstracte methodes. Voor de abstracte methode moet je het keywoord `abstract` zetten.
   - Een abstracte klasse kan *niet geïnstantieerd* worden (object van kunnen maken).
   - Een subklasse moet *alle abstracte methodes* van alle superklasses *overschrijven en implementeren* om geïnstantieerd te kunnen worden, het wordt dan een **concrete klasse**.
   - Een interface is een speciale abstracte klasse, want alle methoden zijn abstract.

10. **Statische attributen en methodes** bestaan op klasseniveau [`static`] en worden opgeroepen via de klasse (`ClassA.attribuut` of `ClassA.method(…)`).
    - Een statisch attribuut heeft maar 1 waarde per klasse.
    - Een statische methode heeft enkel toegang tot statische attributen van de klasse!
11. **Finale** attributen of lokale variabelen zijn constantes die je dus niet kan veranderen [`final`].
12. **Inner klasses** zijn klasses gedefinieerd in een andere, 'outer' klasse.
    - Een inner klasse kan enkel aangesproken worden door die 'outer' klasse.
    - Een inner klasse heeft toegankelijk tot het object waarbinnen het gemaakt is.
13. Een **package** groepeert klasses [`package`], die *geïmporteerd* kunnen worden in andere klasses buiten de package [`import`].
    - Een klasse die je wilt exporteren moet je als `public` declareren in een *file* met dezelfde naam.
    - Binnen een package hebben alle klasses zonder meer toegang tot elkaar.
14. **Visibiliteit** van methodes en attributen:
    - `public` : voor iedereen toegankelijk.
    - *Zonder keyword*: vrij toegankelijk binnen package.
    - `protected` : toegankelijk binnen package én voor subklasses.
    - `private` : enkel toegankelijk voor de klasse zelf.

Met deze cursus hopen we dat jullie van start kunnen met Java en object-georiënteerd programmeren.

VUB, januari 2019

Beerend Ceulemans, David Blinder, Jan Lemeire

Pijlers van object-georiënteerde programmeertalen.

Wat in vetgedrukt staat, komt uit het 2e deel van de cursus (‘Algoritmes en Datastructuren’).

Gebruik en nut begrijpen als toegepast op voorbeelden zoals in de cursus.

I. Encapsulatie

- **2.3 ArrayList p. 12**
- **3.1 Stapel-datastructuur p. 13**
- **6.2 Java’s LinkedList p. 56**

II. Overerving (inheritance)

- 1.1.3 Studentvoorbeeld p. 20
- 1.3 Vriendenvoorbeeld p. 36
- 1.4.1 MyPanel p. 41
- 1.4.3 PainComponent overschrijven p. 44
- **4.3 FunctieMetAfgeleide-interface p. 25**

III. Polymorfisme en abstractie

- 1.2.4 Set p. 31
- 1.2.5 Map p. 33
- 1.4.2 EventListener p. 43
- 1.6.4 Abstracte klassen p. 58
- **4.2 Functie-interface p. 22**
- **5.2.2 Backtracking & Breadth-first p. 32**
- **Addendum bij hoofdstuk 5 (zie website, is optioneel)**
    - **Abstract zoekalgoritme**
    - **Vergelijking van zoekalgoritmes**

Hoofdstuk 0.

Van Python tot Java.

In dit eerste hoofdstuk maken we de overstap van Python naar Java. We bespreken de verschillen en geven wat achtergrondinformatie. In het volgende hoofdstuk gaan we dieper in op de object-georiënteerde aspecten van de taal. Beide hoofdstukken bespreken niet de hele taal en alle mogelijkheden en details van Java. Daarvoor heb je referentieboeken en het internet.
Python heeft me kunnen overtuigen als een *praktische* taal waarin je *snel* iets kunt programmeren. Je zult zien dat in Java sommige dingen wat omslachtiger zijn. Python probeert alles heel simpel te maken. Anderzijds vind ik Python niet de beste keuze voor grote programmeerprojecten. Daarvoor is Java beter geschikt. Ik kom hier later nog op terug.
De verschillen tussen Python en Java wijzen op belangrijke eigenschappen van programmeertalen en informatica in het algemeen. De verschillen op een rijtje:

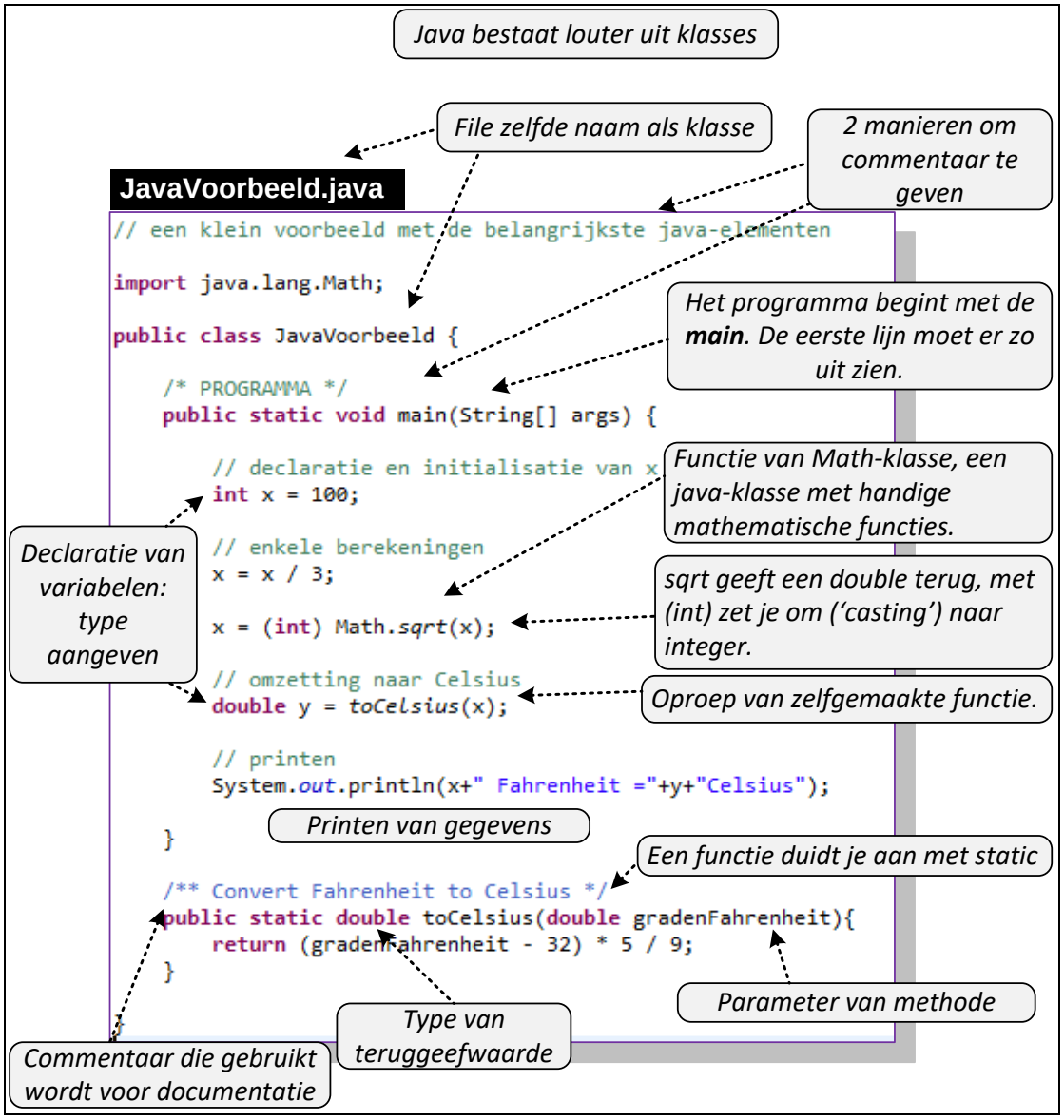
1. Object-georiënteerde taal.
2. public static void main
3. Puntkomma's en accolades
4. System.out.println
5. Typeren: sterk en statisch
6. Arrays en ArrayLists
7. Verschil tussen letters en woorden
8. De *for*-lus
9. Varia

We bespreken nu deze verschillen in detail, maar we starten met enkele programmavoorbeelden. Beschouw het volgende Pythonprogramma:

Hetzelfde in Python

```
'Functions for working with temperatures.'

from math import sqrt

def to_celsius(t):
    'Convert Fahrenheit to Celsius.'
    return (t - 32.0) * 5.0 / 9.0

x = 100
x = x / 3
x = sqrt(x)

y = to_celsius(x)

print str(x)+" Fahrenheit = "+str(y)+" Celsius"
```

Er zijn 2 variabelen x en y, met een functie die een waarde omzet in een andere. Hetzelfde programma in Java geeft het volgende: (de belangrijkste Java-aspecten zijn aangeduid)



Als tweede voorbeeld, probeer het volgende Javaprogramma te begrijpen. Stel bijvoorbeeld dat de gebruiker 24 en 9 intikt (of 8 en 14). Wat is het resultaat van het programma? Probeer te raden wat het programma doet. We noemen het een algoritme. Een 'slimme' manier om iets te doen noemen we een **algoritme**. Een algoritme beschrijft stap voor stap wat je moet doen en kan dan worden geïmplementeerd in een programmeertaal.

```
1  import java.util.Scanner;
2
3  public class GHI {
4
5       /** PROGRAMMA */
6       public static void main(String[] args) {
7
8            System.out.println("Geef 2 getallen: ");
9
10           Scanner scanner = new Scanner(System.in); //
11 om input in te lezen
12           int a = scanner.nextInt();
13           int b = scanner.nextInt();
14
15           int ghi = berekenGHI(a, b);
16           System.out.println("Het resultaat na het
17 ingeven van "+a+" en "+b+" is "+ghi);
18      }
19
20      public static int berekenGHI(int x, int y){
21           while (x != y){
22                if (x > y)
23                     x = x-y;
24                else
25                     y = y-x;
26           }
27           return x;
28      }
29 }
```

Hetzelfde algoritme in Python geeft de volgende code:

```
1  def berekenGHI(x, y):
2      while x != y:
3          if x > y:
4              x = x-y
5          else:
6              y = y-x
7      return x
8
9  s = input("Geef 2 getallen: ")
10 a = int(s)
11 s = input()
12 b = int(s)
13 ggd = berekenGHI(a, b)
14 print("GGD van", a, "en", b, "is", ggd)
```

De Javacode is bijna dubbel zo lang. Dat is ook het uitdrukkelijke doel van Python: een eenvoudige, snelle programmeertaal zonder poespas. Laat ons de verschillen één-voor-één bespreken.

0.1. Object-georiënteerde taal

Java is een puur object-georiënteerde taal. Hier komen we nog uitgebreid op terug in het volgend hoofdstuk. Voorlopig moet je enkel weten dat een programma opgebouwd wordt met objecten die gedefinieerd worden door *klasses*. Een klasse definieer je als *class GrootstGemeneDeler* en die zet je in een Java-file met dezelfde naam. Hier: *GrootstGemeneDeler.java*.

In Python kan je ook objecten definiëren en gebruiken, maar je kan ook zonder. Het is geen puur object-georiënteerde taal zoals Java. In Java moet alles binnen een klasse gedefinieerd worden, ook functies. Deze worden daarom **methodes** genoemd, ze geven ons de operaties die we kunnen doen op en met het object in kwestie.

Een **statische methode** staat echter los van objecten. Ze krijgt het keywoord *static* voor hun definitie. Zoals *GrootstGemeneDeler* in ons voorbeeld. Ze kan worden gezien als een algemene methode die iets vereenvoudigt, berekent of uitvoert. De methode is van overal oproepbaar, er is geen object voor nodig. Statische methodes zijn wel gegroepeerd in een klasse, maar dit is in feite om een logische opdeling van de methodes te maken. Een klasse groepeert methodes die om hetzelfde gaan. Zo biedt de *Math* klasse (van de java.lang package) de voornaamste mathematische functies, zoals de logaritme, de worteltrekking of de cosinus. We noemen statische methodes dan ook functies, ze nemen inputwaarden en berekenen een outputwaarde.

0.2 “public static void main”

In Python kan je elke py-file uitvoeren als een apart programma. Je kunt anderzijds ook een Python-file importeren in een andere file en gebruik maken van zijn functies en variabelen.
In Java zet je je code in een.*java-file* waarbij je in elke file 1 klasse definieert. Je kunt elke file ook uitvoeren als een apart programma. Daarvoor definieer je een *main-methode* in de klasse met als header *public static void main(String[] args)*, zie lijn 6. Bij het uitvoeren wordt de code van de methode uitgevoerd. Als je op run duwt wordt de *main* van de geselecteerde klasse uitgevoerd. De instructies van de *main* zullen stap-voor-stap uitgevoerd worden en na het uitvoeren van alle instructies stopt het programma.

Zonder de *main*-methode kan je de file niet uitvoeren als een apart programma. Je kan de klasse wel gebruiken in andere files, zoals we veel zullen doen. Voorlopig hoef je ook niet te weten wat ‘public’, ‘static’ en ‘args’ betekenen. ‘void’ leggen we binnenkort uit. Onthoud dat de eerste lijn van de methode zo moet zijn. De voorbeeldcode heeft twee statische methodes of functies: de *main* en *berekenGrootstGemeneDeler*. In Python werkten we vooral met functies die we definiëren met *def*.

0.3. Puntkomma’s en accolades

In Java zet je puntkomma’s en accolades. Puntkomma’s zet je na een enkelvoudig statement (lijnen 8, 10, 11, 12 en 14) en accolades bij statements waarbij een blok van statements hoort, zoals een methode (lijnen 6 en 19) of een *while* (lijn 21). Met

accolades geef je begin en einde van een blok aan. In Python doen we dit door in te springen (*indentation* in het engels).

In Java is het inspringen niet verplicht zoals in Python, eigenlijk spelen spaties en tabs geen rol. Daartegenover staat dat je accolades moet gebruiken om een groep van statements (‘block’) aan te duiden die samen horen. Het is wel ten zeerste aan te raden om ook in te springen. Zo is je code leesbaarder. Wat heeft Python gedaan: inspringen verplicht gemaakt. Zo konden de accolades afgeschaft worden. In Python zet je een dubbelpunt.

In Python moet je telkens een nieuwe lijn starten. In Java moet dit in feite niet. Maar door dit te verplichten kon Python de puntkomma’s ook afschaffen.

0.4 System.out.println

Zoals te zien op lijn 15 print je tekst naar console met *System.out.println*. Tussen haakjes zet je de tekst. Je mag stukjes *concateneren* met een plus-teken en dit stukjes mogen ook getallen of zelfs objecten zijn (hier komen we nog op terug). In Python moet je eerst alles omzetten naar String. Java doet dit automatisch.
Tekst inlezen doe je het gemakkelijkst via een *Scanner*-object. Zo’n object maak je aan met *new Scanner(System.in)*. Met *System.in* geef je aan dat je de tekst inleest van de standaardinput (console dus). *System.out* duidt op de standaardoutput. Het *Scanner*-object *scanner* kunnen we dan vragen naar een integer met *nextInt()*. Deze methode zal de ingedrukte tekens omzetten naar een getal. In Python gebeurt dit met *x = int(s)*.

0.5 Typeren: sterk en statisch

Het grootste verschil tussen de twee talen is dat Java eist dat je het type van elke variabele expliciet aanduidt door deze te declareren.

In Python wordt het type van een variabele bepaald bij het toekennen van een waarde. Als je *x=4* schrijft geef je te kennen dat *x* een integer is. In het vervolg van het programma wordt x als een integer behandeld. Alhoewel, je kunt in Python *x* een nieuw type geven bij een nieuwe toekenning. Met *x = ‘bla’* geef je aan dat *x* een string geworden is. Een variabele kan tijdens het programma van type veranderen, **dynamische typering** genoemd.

Dit kan in Java niet. In Java geef je expliciet het type aan en dit moet hetzelfde blijven tijdens het verloop van het programma. Dit wordt **statisch typeren** genoemd.
Je moet dus eerst *int x;* schrijven om *x* te declareren als een integer. Je mag *x* ook direct een waarde geven, *int x=4;*. *x* blijft vanaf dan een integer. *x = ‘bla’* geeft een compilatiefout: de compiler checkt het type van *x* en geeft aan dat dit niet kan.

Python is dus weer gemakkelijker. Ze doet niet zo moeilijk, je kan een variabele een nieuw type geven. Java wel dus. Programmeurs wereldwijd hebben hier al heel veel over gedebatteerd. En die debatten kunnen hoog oplopen. Ook binnen de VUB gebeurde dit toen we een keuze moesten maken over de aan te leren programmeertaal.

Mijns inziens valt er wel wat te zeggen voor sterke, statische typering met type checking van de compiler. Laat me dit even uitleggen, het is nuttig en misschien hebben jullie hier nog helemaal niet bij stilgestaan.

In Python declareer je een functie als volgt:

```
def berekenGrootstGemeneDeler(x, y):
```

Dezelfde functiedeclaratie wordt in Java:

```
public static int berekenGrootstGemeneDeler(int x, int
y){
```

Het '*public*' keyword mag je voorlopig negeren. Python heeft ervoor gezorgd dat die keywords niet nodig zijn. In Java kun je echter niet zonder…

Het belangrijke verschil is dat we de types van de parameters expliciet moeten declareren. We geven dus aan dat *x* en *y* beide integers zijn. Ook declareer je het type van de teruggeefwaarde: het type zet je voor de naam van de methode. Hier: *int*. Als je methode niets teruggeeft geef je dit aan met het *void*-keyword. Zie lijn 6 van de code voor de *main*. Dit duidt er dus op dat de *main* geen waarde teruggeeft.

De *header* of *hoofdding* van een methode, de 'eerste lijn', bepaalt hoe je de methode moet gebruiken. Ze geeft de *naam* van de methode, de *input parameters* en de *teruggeefwaarde* van de methode. Een methode kan iets of niets teruggeven. De *grootsteGemeneDeler* geeft een integerwaarde terug. *println* van *System.out* geeft niet terug, ze print enkel iets op het scherm.

Python is dus weer simpeler dan Java, maar mijns inziens heeft Java hier een streepje voor. Zeker als je grote programmeerprojecten maakt en samenwerkt met andere programmeurs. Ik verklaar me nader.

De Pythonfunctie kan je oproepen met om het even wat voor parameterwaarden. Je kan ze oproepen met tekst: *berekenGrootstGemeneDeler("bla", "boe")*. De code crasht dan op lijn 7 omdat Python niet weet hoe twee strings van elkaar af te trekken. Je kunt de functie ook oproepen met reële getallen: *berekenGrootstGemeneDeler(4.5, 5.5)* geeft *0.5* als resultaat. Maar het werkt niet voor alle reële getallen, als je 4.5 en 5.1 meegeeft kom je in een oneindige lus terecht!

Natuurlijk heeft dit allemaal weinig zin, de grootst gemene deler wordt enkel gedefinieerd voor natuurlijke getallen. Dit zie je dan ook in Java: je wordt verplicht een integer mee te geven. Strings of reële getallen kan niet, de compiler zal een fout geven zodat je het programma zelfs niet kunt uitvoeren. Door de types expliciet aan te geven, weet je als programmeur wat je moet meegeven. In Python moet je dit opzoeken in de documentatie. Waardoor je code weer langer wordt… In Java ben je dus zeker dat je niets fout kunt doen: de compiler checkt de types. Je programma zal ook niet crashen (als de functie geen fouten bevat tenminste).

Bij mij zorgt dit altijd voor de nodige geruststelling als ik andermans code gebruik: je kunt met een gerust hart de publieke methodes (*public*-keywoord) gebruiken zolang je maar de juiste parametertypes meegeeft. De programmeur heeft ervoor gezorgd dat er

dan niets fout kan gaan. Java is dus geschikter voor grotere projecten en hergebruik van code via bibliotheken.

Merk op dat dit nog niet geldt voor de code van grootst gemene deler: een integer kan ook negatief zijn (gehele getallen), maar dan werkt de functie niet. Daarvoor moet je eerst nog checken of de gebruiker geen negatief getal meegeeft. In dat geval geef je een fout met de melding dat enkel positieve getallen toegelaten zijn. Dat is ook wat je in Python zou moeten doen: testen of de inputwaarden correct zijn.

0.6 Arrays en ArrayLists

De *lists* van Python worden in Java vervangen door *arrays* en *ArrayLists*. Python heeft hier ook weer gekozen voor het gemak van de programmeur. Ze heeft de *ArrayList* van Python verheven tot de standaardlijst, terwijl in Java de standaardlijst gegeven is door de *array*. Maar omdat de array het nadeel heeft van niet uitbreidbaar te zijn, is die in Python eruit gehaald.

Hoofdstuk 2 van deel II is helemaal gewijd aan *arrays*. Op het einde van dat hoofdstuk komen we nog even terug op deze vergelijking.

Java biedt standaard geen *tupels* aan (lijsten waarvan de elementen niet aanpasbaar zijn) zoals Python dat wel doet. Hier een samenvatting van de 4 datastructuren:

Java Array	**Java ArrayList**	**Python list**	**Python tuple**
Vaste grootte	*Flexibele grootte*	*Flexibele grootte*	*Vaste grootte*
Grootte te specifieren	*Initiële grootte is facultatief*	*Niet nodig*	*Bij initialisatie*
int[] array = new int[5];	ArrayList<Integer> arrayList = **new ArrayList<Integer>();**	**list=[]**	*Creatie en initialisatie samen*
array = **new int[]{1, 2,3};**	*Initialisatie niet mogelijk*	**list=[1, 2, 3]**	**tuple = (1, 2, 3, 4, 5)**
x = array[2];	x = arrayList.get(2);	**x=list[2]**	**x=tuple[2]**
array[1] = 5;	arrayList.set(1, 5);	**list[1]=5**	*Veranderen niet mogelijk*
Toevoegen niet mogelijk	arrayList.add(7);	**list.append(7)**	*Niet mogelijk*
array.length	arrayList.size()	**len(list)**	**len(tuple)**

0.7 Verschil tussen letters en woorden

In Python wordt geen onderscheid gemaakt tussen letters en woorden. Je kan een variabele een letterwaarde geven, bvb x= 'a', en op dezelfde manier een woordwaarde (String), bvb x = 'blabla'. Merk ook op dat je enkelvoudige of dubbele quotes mag gebruiken: x= "a" en x = "blabla" geeft hetzelfde resultaat.
In Java wordt er wel expliciet een verschil gemaakt tussen een letter (*char*) en een zin (*String*). Een letter is een *enkelvoudig basistype* en definieer je met enkele quotes: char x = 'a'. Een zin is een object dat je definieert als String x = "blabla" die je initialiseert met dubbele quotes.
Zoals we later zullen zien is een String een lijst (array) van letters. Door er een klasse van te maken zie je die lijst niet meer, de lijst/array is verstopt in het object. Je moet je er dan ook geen zorgen over maken. In het hoofdstuk over arrays gaan we hier verder op in. In Python is de String zelfs een basisklasse en Python vereenvoudigt het gebruik ervan verder door geen onderscheid te maken met letters. Java toont dus iets meer van de onderliggende bouwstenen: namelijk een enkelvoudige waarde (letter) die je samenstelt tot woorden of zinnen met behulp van arrays/lijsten.

0.8 De *for*-lus

Neem de for-lus van Python:

```
totaal = 0
for x in list:
    totaal += x
```

Deze wordt in Java:

```
int tot = 0;
for(int v: array)
      tot += v;
```

De array mag ook een *ArrayList* zijn. Elke klasse die een lijst (Java-*Collection*) voorstelt kan je op die manier doorlopen.

De *range*-functie van Python bestaat helaas niet in Java. De functie is handig om gestructureerde lijsten op te bouwen en wordt ook veel gebruikt in *for*-lussen. Java heeft echter een andere, belangrijke basisvorm voor een *for*-lus waarmee je hetzelfde resultaat krijgt.

```
for(int i=0;i<100;i+=2){
      System.out.print(i+", ");
}
```

Alle even getallen kleiner dan 100 worden geprint (0 ook).

In Python zou je dit schrijven als:

```
for i in range(0, 100, 2):
      print i, ','
```

De *for*-lus van Java is flexibeler de Python-constructie met de *range*-functie. De drie delen van de *for*-lus, **initialisatie** – **eindconditie** – **increment**, mogen eender welk statement bevatten.

Merk op dat de eerste code in de basisvorm van de *for*-lus het volgende wordt:

```
int tot = 0;
for(int i=0;i<array.length; i++)
      tot += array[i];
```

De eerste vorm is dus iets eleganter.

0.9 Varia

- In de meeste programmeertalen kan een functie maar één waarde teruggeven. Niet zoals in Python waar je meerdere waarden mag teruggeven. Je doet dit simpelweg met *return x, y*. In Java kan dat dus niet. Ik heb dit steeds een beperking gevonden. Om meerdere dingen terug te geven in Java, moet je de resultaten in een array zetten of in een object. Maar dit is in sommige gevallen iets te omslachtig ten opzichte van Python. In 6.6.1 zien we een functie waar twee teruggeefwaarden handig zou zijn.
- `do{ … } while(test);` bestaat niet in Python. Voert iteratief de statements uit zolang de test true is.
- je hebt ook de `switch … case` statement
- `x+=y;` is exact hetzelfde (maar korter) als `x=x+y;`
- `x++;` is exact hetzelfde (maar korter) als `x=x+1;`
- hetzelfde als bovenstaande geldt voor de `-`.
- `x = y > 2 ? 3 : 4;` is exact hetzelfde (maar korter) als

```
if (y > 2)
  x= 3;
else
  x = 4;
```

  - Algemeen: `a ? b : c` geeft `b` terug als `a` waar is, anders `c`
- `Enumeratie:` zie 1.2 waar we Faculteit als een enumeratie definiëren

Tot slot de basistypes van Java:

Type	Grootte	Minimum	Maximum	Precisie
byte	8 bits	-128	127	
short	16 bits	-32 768	32 767	
int	32 bits	-2^{31}	$2^{31}-1$	
long	64 bits	-2^{63}	$2^{63}-1$	
char	16 bits	0	$2^{16}-1$	
float	32 bits	-3.4×10^{38}	3.4×10^{38}	6-9 cijfers
double	64 bits	-1.7×10^{308}	1.7×10^{308}	15-17 cijfers

Hoofdstuk 1.

Object-georiënteerd programmeren met Java.

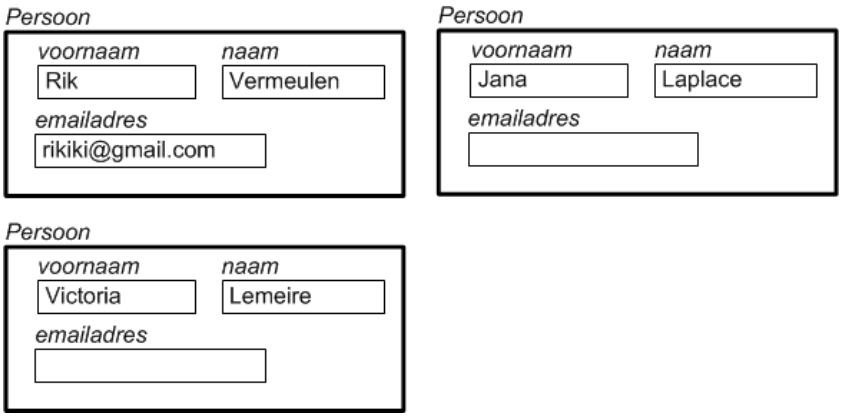
In dit hoofdstuk leggen we de essentie van object-georiënteerd programmeren uit a.d.h.v. Java. De essentie is dezelfde voor alle programmeertalen, al zijn er kleine verschillen tussen de verschillende programmeertalen. De Javaregels van object-georiënteerdheid vind je aan het begin van dit boek. Hier proberen we het inzicht bij te brengen.

In de laatste paragrafen van dit hoofdstuk gaan we na wat er gebeurt als we een programma uitvoeren. Dit is belangrijk om efficiënte programma's te kunnen maken. Je moet weten wat er precies intern in de processor gebeurt, zodat je de juiste keuzes maakt. Maar bovenal om je programma te *debuggen*: het opsporen van fouten. Dat kan een zware klus zijn. Hiervoor moet je nagaan of het programma doet wat het moet doen. Daarvoor moet je dus eerst begrijpen wat je programma normaliter zou moeten doen.

De laatste paragrafen kan je alvast lezen, maar om ze volledig te begrijpen ga je beter verder met de volgende hoofdstukken om daarna, met de opgedane kennis, terug te grijpen naar dit hoofdstuk. Eenmaal je dit begrijpt, heb je de essentie van programmeren onder de knie en kan je er gedurende je carrière als ingenieur steeds naar terug grijpen. Waarschijnlijk zal je nog heel wat moeten leren over programmeren, maar met deze cursus heb je de basis op zak. Daar ben ik van overtuigd.

1.1. Klassen, attributen, methodes, objecten

Wat je tot nu toe geleerd hebt in Python is een *procedurele taal*. Je programma bestond uit een collectie procedures of functies (wat hetzelfde is). Het concept functie is het fundament van deze talen. Nu gaan we overschakelen op een object-georiënteerde talen die georganiseerd is rond ... *objecten*. Hier zien we 3 objecten waarbij elk object een Persoon voorstelt met zijn desbetreffende eigenschappen.



1.1.1 Klassen & objecten

In eerste instantie is een object een collectie van gegevens die bij elkaar horen omdat ze iets zeggen over eenzelfde ding, het *object*. We noemen de gegevens de eigenschappen of *attributen* van het object. Neem het voorbeeld van de universiteit die een applicatie aanmaakt. Ze zal de gegevens van elke student groeperen in een object. Eén object per student.

Een *klasse* definieert hoe een object eruitziet. Elke klasse definieert een bepaald type object. Je programmeert dus de klassen waarmee je dan objecten aanmaakt. Zie regel 1 van Java's spelregels: "*Java bestaat enkel uit klasses*". Het definieert de eigenschappen of *attributen* van het object en de dingen die je met het object kunt doen, zijn operaties of *methoden*. Dit is regel 2: "*Een klasse bestaat uit attributen en methodes*".

De klasse *Persoon* hebben we 3 attributen gegeven en 4 methodes.

```
public class Persoon {

	//======================= ATTRIBUTEN ============================//
	String voornaam, naam;
	String emailadres;

	//======================= CONSTRUCTORS ==========================//
	Persoon(String voornaam, String naam){
		this.voornaam = voornaam;
		this.naam = naam;
	}
	Persoon(String voornaam, String naam, String emailadres){
		this.voornaam = voornaam;
		this.naam = naam;
		this.emailadres = emailadres;
	}

	//======================= METHODES ==============================//
	public void maakDefaultEmailadres(String domeinVanProvider){
		emailadres = voornaam+"."+naam+"@"+domeinVanProvider;
	}

	public String toString(){
	return voornaam+" "+naam+(emailadres==null?"": " ["+emailadres+"]");
	}
}
```

maakDefaultEmailadres() is een gewone methode die voor een persoon een standaard emailadres aanmaakt, gegeven de mailprovider. De methode gebruikt dus 2 attributen van het object om een 3^{e} in te vullen. Da's de bedoeling van object-georiënteerd programmeren: de methodes bepalen de operaties die je op de objecten kunt uitvoeren (de dingen die je met een object kan doen). In procedurele talen zijn de functies 'losse dingen', hier hangen we ze vast aan de gegevens waarop ze werken.

De *toString()* gebruik je om het object te printen: het zet de gegevens van het object in een string die je dan kan printen of naar file schrijven. Ook Java zal deze methode

gebruiken om een object te printen! Het is een goed idee om deze te definiëren voor elke klasse, om het debuggen van je programma te vergemakkelijken.

De 2 eerste methodes, de *constructors*, verdienen wat extra aandacht. Een constructor heeft dezelfde naam als de klasse. Neem een klasse *Persoon* met eigenschappen naam, voornaam en emailadres. We wensen dat naam en voornaam ingevuld worden bij creatie van een *Persoon*-object. Dit doen we door deze als parameters te definiëren in de constructor. Om bij de constructie ook het emailadres in te vullen, voorzien we een tweede constructor waarbij het emailadres als derde parameter geldt. Zo kan de gebruiker kiezen tussen beide constructors, afhankelijk van welke parameters die hij wil meegeven.

Als je geen constructor definieert maakt Java de *default constructor* aan, deze heeft geen argumenten en initialiseert alle attributen met een 0 (strings met “”). Maar van zodra je zelf een constructor aanmaakt, vervalt de default constructor.

Hier zie je hoe we objecten aanmaken en gebruiken (via de methodes of direct de attributen aanspreken). De eerste lijn moet je als volgt interpreteren: met “`new Persoon("Rik", "Vermeulen", "rikiki@gmail.com")`” wordt een nieuw object aangemaakt. Dit kennen we toe aan een variabele: “`rik`”. Het type van de variabele is de klasse “`Persoon`”.

```
public class PersoonMain {

	/** PROGRAMMA */
	public static void main(String[] args) {

		Persoon rik = new Persoon("Rik", "Vermeulen",
"rikiki@gmail.com");
		Persoon jana = new Persoon("Jana", "Laplace");
		Persoon victoria = new Persoon("Victoria", "Lemeire");

		victoria.voornaam = "Vicky";
		jana.maakDefaultEmailadres("hotmail.com");
		System.out.println("Adres aangemaakt: "+jana.emailadres);

		System.out.println("Rik: "+rik);
		System.out.println("Jana: "+jana);
		System.out.println("Victoria: "+victoria);
	}
}
```

Deze code is te vinden in **package *objectgeoriënteerd***.

1.1.2 Een uitbreiding

Overerving is 1 van de 3 pijlers van object-georiënteerd programmeren. Zie Java-spelregel 6.

Wat gebeurt er nu bij overerving? Neem een applicatie van de universiteit. Deze moet de gegevens van alle studenten bij houden. We gaan dus een Student-klasse implementeren. Een student is echter ook een persoon zoals in 1.1. De *Student*-klasse laten we overerven van de *Persoon*-klasse. Een *Student*-object zal dus alle eigenschappen en methoden overerven van *Persoon*. Dit doen we als volgt:

```
public class Student extends Persoon {

	enum Faculteit {IR, WE, GF, LK, LW, ES, RC, PE};

	int rolnummer;
	Faculteit faculteit = Faculteit.IR; // default waarde voor alle
nieuwe objecten

	Student(String voornaam, String naam, int rolnr, Faculteit fac){
		super(voornaam, naam);
		this.rolnummer=rolnr;
		this.faculteit = fac;
	}
}
```

Naast de attributen die overgeërfd worden van Persoon, voegt de Student-klasse het rolnummer en de faculteit toe. Aangezien de faculteiten een vaste lijst is, definiëren we dit met een enumeratie (`enum`). Een variabele van het type `Faculteit` zal één van de opgenomen waarden aannemen (en geen andere).

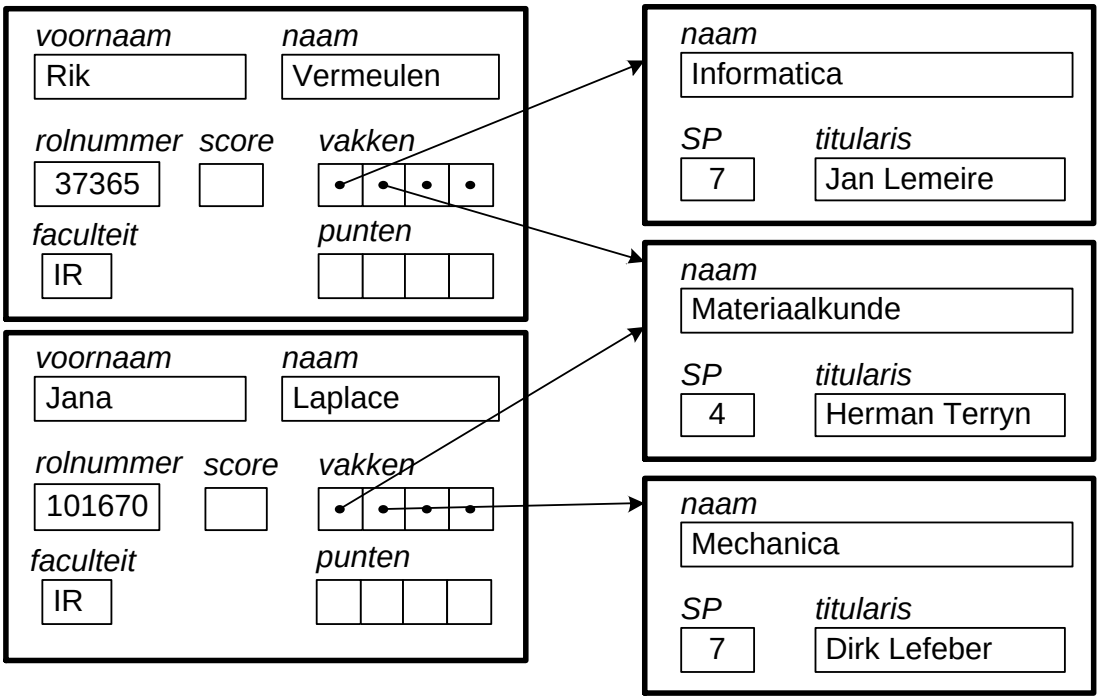
Student heeft een constructor, maar als je een student-object aanmaakt, maak je in feite ook een *Persoon*-object aan en moet er ook1 van de bestaande constructors van *Persoon* opgeroepen worden. In onze code vraagt de constructor van de Student-klasse dat voornaam, naam en rolnummer meegegeven worden. Voornaam en naam zijn echter attributen van de superklasse Persoon, deze worden ingevuld door het oproepen van de constructor van Persoon met `super(voornaam, naam);` het gereserveerde woord *super* slaat dus op een constructor van de superklasse. Deze moet altijd op de eerste lijn van de constructor worden opgeroepen. Als je dit niet zet, zal Java de superconstructor van de superklasse oproepen. Deze moet dan wel bestaan, anders zal de compiler dit niet accepteren (zoals hier bij *Persoon*).

1.1.3. Object-attributen

Tot hiertoe waren alle attributen van het *primitieve type* (zoals getallen en booleans) of strings. Maar attributen kunnen zelf ook objecten zijn. Dan moeten we dat attribuut-object zien als iets buiten het object. Het is een ander object in de ‘wereld’, het attribuut is een verwijzing (*referentie*) naar dat object.

We gaan ook de vakken bijhouden die elke student volgt. Een vak gaan we echter ook opslaan als een object, aangezien het meerdere attributen heeft die kunnen veranderen

in de tijd. Elke student kan meerdere vakken volgen. Die slaan we op in een lijst die bij de student hoort. Stel dat we de volgende gegevensstructuur willen krijgen.



Dit resulteert in de volgende klassedefinities (deze vind je in de **package objectgeoriënteerd** in de code die je kan downloaden van de website):

```
public class Student extends Persoon
{
	enum Faculteit {IR, WE, GF, LK, LW, ES, RC, PE};

	int rolnummer;
	Faculteit faculteit = Faculteit.IR;
	ArrayList<Vak> vakken;
	Map<Vak, Integer> punten;
	float score;

	Student(String voornaam, String naam, int rolnummer, Faculteit
faculteit){
		super(voornaam, naam);
		this.rolnummer=rolnummer;
		this.faculteit = faculteit;
		vakken = new ArrayList<Vak>();
		punten = new HashMap<Vak, Integer>();
	}
```

```
public class Vak {
	String naam, titularis;
	int SP;
	Vak(String naam, String titularis, int SP){
		this.naam = naam;
		this.titularis = titularis;
		this.SP = SP;
	}
}
```

Een ArrayList is een lijst om objecten van een bepaald type bij te houden. Dit type duidt je aan tussen <...>. Met een Map verbind je aan een Vak een getal (integer): zo houden we de punten bij van elk vak. Merk op dat we in de constructor een ArrayList- en een Mapobject aanmaken met new. Pas dan wordt er een object toegekend aan de attributen vakken en punten. De javaklassen ArrayList en Map bespreken we in detail in sectie 1.2.

Hieronder de code waarmee we de objecten van het schema aanmaken. De objecten maken we weer aan met new. De vakken en punten worden aan de lijst en map toegevoegd via het aanroepen van methoden.

```
/** PROGRAMMA */
public static void main(String[] args) {
	Student rik = new Student("Rik", "Vermeulen", 37365, Faculteit.IR);
	Student jana = new Student("Jana", "Laplace", 101670, Faculteit.WE);

	Vak informatica = new Vak("Informatica", "Jan Lemeire", 7);
	Vak materiaalkunde = new Vak("Materiaalkunde", "Herman Terryn", 4);
	Vak mechanica = new Vak("Mechanica", "Dirk Lefeber", 7);

	rik.vakken.add(informatica);
	rik.punten.put(informatica, 14);
	rik.vakken.add(materiaalkunde);
	rik.punten.put(materiaalkunde, 12);
	rik.vakken.add(mechanica);
	rik.punten.put(mechanica, 17);

	jana.vakken.add(informatica);
	jana.punten.put(informatica, 16);
	jana.vakken.add(mechanica);
	jana.punten.put(mechanica, 13);
}
```

Zoals we reeds zagen zijn attributen maar de helft van de object-georiënteerde taal. Het groeperen van gegevens van dingen is ook mogelijk in procedurele talen. Daar noemen we het *records* of *structs* (zoals in de taal C). De ware kracht van object-georiënteerde talen is het principe om de functies/procedures van de procedurele talen aan objecten te hangen. Een functie zal immers iets te maken hebben met de gegevens van een object. Waarom kunnen we die functie dan niet bij het object plaatsen? Zo'n functie wordt dan een ***methode***.
In ons voorbeeld: een student kan zich inschrijven voor een cursus. Een professor kan hem punten geven. Dat zijn 'operaties' op een studentobject en kunnen dus om die reden aan het object geplakt worden. Een methode definiëren we samen met de definitie van het object (in de klasse).
De volgende methode berekent de score van een student; een gewogen gemiddelde van de punten van alle vakken. De punten van een vak worden uit de map gehaald via de *get*-methode.

```
int berekenTotaal(){
        score=0;
        int totaalSP=0;
        for(Vak vak: vakken){
                score += punten.get(vak) * vak.SP;
                totaalSP += vak.SP;
        }
        score /= totaalSP;
        return score;
}
```

Een methode wordt pas uitgevoerd als we ze oproepen. Dit gebeurt door de volgende code aan de *main* toe te voegen:

```
float scoreRik = rik.berekenTotaalScore();
float scoreJana = jana.berekenTotaalScore();
System.out.println(rik+" behaalde "+scoreRik+"/20");
System.out.println(jana+" behaalde "+scoreJana+"/20");
```

Nu gaan we `ObjectOefening1` oplossen. Deze oefeningen staan achteraan dit hoofdstuk.

1.1.4. Variabelen en scope/visibiliteit

Bekijk het volgende (**package *voorbeelden***). Het toont de vier mogelijke soorten variabelen.

```
public class MyList {

        /** PROGRAMMA */
        public static void main(String[] args) {
                MyList list = new MyList();

                for(int i=0;i<10;i++)
                        list.add(i % 5);
                Arrays.sort(list.array, 0, list.k);
                System.out.println("Max is : "+list.getMax());
        }

        static int DEFAULT_N = 100;

        int[] array;
        int n, k;

        MyList(int n){
                array = new int[n];
                this.n = n;
                k=0;
        }
        MyList(){
                this(DEFAULT_N);
        }
        void add(int x){
                array[k] = x;
                k++;
```

lokale variabele

actuele parameter

gebruik v. attribuut

statische variabele

attribuut

formele parameter

```
            if (k > n)
                  System.out.println("Array is full");
      }
      int getMax(){
            int max = array[0];
            for(int i=1;i<k;i++)
                  if (max < array[i])
                        max = array[i];
            return max;
      }
}
```

Een *lokale variabele* is een variabele die is gedeclareerd in een methode (statisch of niet statisch). We noemen die lokaal omdat ze enkel in die methode gebruikt kan worden. Waar je een variabele overal kunt gebruiken, wordt het de *scope* genoemd. Voor een lokale variabele is de scope dus de methode zelf. Een lokale variabele kun je overal in een methode declareren, en daarna kun je ze gebruiken. Niet vòòr de declaratie dus.
Een *parameter* is een variabele waarmee je de inputwaarden van een methode meegeeft. In het hoofdding van de methode worden de parameters gedefinieerd. We noemen dat de **formele parameters**. Formele parameters kunnen net als lokale variabelen enkel binnen een methode worden gebruikt. Als je een methode oproept, moet je voor elke formele parameter een waarde meegeven. Dit worden de **actuele parameters** genoemd. Het gebruik van parameters om gegevens uit te wisselen tussen het hoofdprogramma en methodes of functies, is een veilige en gebruikersvriendelijke manier. Een functie wordt op een abstracte manier geschreven. Bij het schrijven ken je de precieze waarden van alle variabelen nog niet. Een algoritme bedenk je als een algemene oplossing. Dit is heel logisch, maar het veroorzaakt bij de beginnende programmeur in de praktijk toch heel wat verwarring.

Een **attribuut** is een eigenschap van een object. Ze behoort tot een object. Haar gebruiken doe je via het object. Zie de vierde lijn in de *main*: attributen *array* en *k* bereik je via het object. In eclipse worden die netjes in het blauw weergegeven.
Een *statische variabele* is een variabele die binnen een klasse gedefinieerd is en die je gebruikt via de klasse, net als een statische methode. Een statische methode noemen we een functie omdat je die overal kunt gebruiken. Ook statische variabelen zijn overal toegankelijk. Je kunt ze dan ook zien als *globale variabelen*. Het gebruik ervan wordt wel afgeraden.

1.2. Libraryklassen en interfaces

Misschien wel de beste introductie tot klassen en objecten is het leren gebruiken van de klassen uit de Javabibliotheek. Hiermee introduceren we ook de concepten *interface*, *generics* en *encapsulatie*.

Zie Java spelregels (p. 5) voor de definities en regels.

1.2.1. Scanner

De klasse “Scanner” stelt je in staat om String te ontleden (dit wordt “*parsing*” genoemd). Hiermee kan je gebruikersinvoer en/of externe tekstbestanden verwerken. Het aanmaken doe je als volgt:

```
Scanner sc = new Scanner(System.in);                // Gebruikersinvoer
Scanner sc = new Scanner(new File("myfile.txt"));   // Tekstbestand
```

Vervolgens kun je op het Scanner-object methodes oproepen die de meegegeven tekst opdeelt in Strings, die op hun beurt kunnen worden omgezet in andere types; een voorbeeld waarbij we de invoer van de gebruiker omzetten naar een kommagetal (double):

```
import java.util.Scanner;

public class ScannerExample {

	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);

		System.out.print("Geef een invoer:");
		double num = scanner.nextDouble();

		System.out.println("De vierkantswortel van " + num + "is: " +
Math.sqrt(num));
		scanner.close();
	}
}
```

Hier volgen een aantal nuttige methodes van Scanner:

Constructor Summary

Scanner(**InputStream** source) Constructs a new Scanner that produces values scanned from the specified input stream.
Scanner(**Path** source) Constructs a new Scanner that produces values scanned from the specified file.
Scanner(**String** source) Constructs a new Scanner that produces values scanned from the specified string.

Method Summary

void	**close**() Closes this scanner.
String	**findInLine**(**String** pattern) Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
String	**findWithinHorizon**(**Pattern** pattern, int horizon)

	Attempts to find the next occurrence of the specified pattern.
boolean	**hasNext**() Returns true if this scanner has another token in its input.
boolean	**hasNext**(**String** pattern) Returns true if the next token matches the pattern constructed from the specified string.
boolean	**hasNextBoolean**() Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false".
boolean	**hasNextDouble**() Returns true if the next token in this scanner's input can be interpreted as a double value using the **nextDouble()** method.
boolean	**hasNextFloat**() Returns true if the next token in this scanner's input can be interpreted as a float value using the **nextFloat()** method.
boolean	**hasNextInt**() Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the **nextInt()** method.
boolean	**hasNextLine**() Returns true if there is another line in the input of this scanner.
boolean	**hasNextLong**() Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the **nextLong()** method.
String	**next**() Finds and returns the next complete token from this scanner.
boolean	**nextBoolean**() Scans the next token of the input into a boolean value and returns that value.
double	**nextDouble**() Scans the next token of the input as a double.
float	**nextFloat**() Scans the next token of the input as a float.
int	**nextInt**() Scans the next token of the input as an int.
String	**nextLine**() Advances this scanner past the current line and returns the input that was skipped.

1.2.2. Random

Voor vele toepassingen zoals simulaties, spelletjes en bepaalde algoritmes, hebben we willekeurige getallen nodig. Dit kunnen we realiseren met de klasse Random: deze zal willekeurige getallen genereren volgens een bepaalde distributie.

Klassieke computers zijn echter deterministisch: het is niet mogelijk om zomaar willekeurige getallen te genereren. Daarom wordt gebruik gemaakt van "pseudo-random number generators" (PRNG). Gegeven een startwaarde (ook wel de *seed* genaamd), zal de PRNG een serie van goedgekozen operaties toepassen op de seed. Dit gaat resulteren in een nieuw getal, welke opnieuw gebruikt wordt als invoer voor de PRNG. Goed ontworpen PRNG zullen een successie van getallen produceren die niet te onderscheiden zijn van een echte willekeurige sequentie. Hier een voorbeeld van een PRNG (in Python):

```
nSeed = 5323 # initial starting seed
def PRNG():
    global nSeed

    # Take the current seed and generate a new value from it
    nSeed = 8253729 * nSeed + 2396403
    # Take the seed and return a value between 0 and 32767
    return nSeed  % 32767
```

Dit betekent echter dat als je een constante seed instelt, de serie geproduceerde getallen steeds dezelfde zal zijn (deterministisch) wanneer je je programma opnieuw opstart. Soms is dit wenselijk, maar meestal niet: daarom wordt als seed vaak iets tijdgebonden gebruikt, bv. de huidige tijd in milliseconden uitgedrukt.

Hieronder een voorbeeld:

```
import java.util.Random;

public class Randomtest {
   public static void main(String[] args) {
       // de seed wordt automatisch bepaald (op basis van de huidige tijd)
       Random rand = new Random();
       for (int i=0; i<8; i++)
            System.out.print(rand.nextInt(1000) + ", ");
       System.out.println();

       rand = new Random(42); // vaste seed
       for (int i=0; i<8; i++)
            System.out.print(rand.nextInt(1000) + ", ");
       System.out.println();

       rand = new Random(42); // vaste seed (dezelfde!)
       for (int i=0; i<8; i++)
             System.out.print(rand.nextInt(1000) + ", ");
       System.out.println();
   }
}
```

```
Uitvoer:
200, 593, 612, 198, 427, 419, 276, 790,
130, 763, 248, 884, 970, 525, 505, 918,
130, 763, 248, 884, 970, 525, 505, 918,
```

Een aantal methodes van Random die goed van pas komen. Je kan dus willekeurig gehele getallen opvragen, maar ook kommagetallen tussen 0 en 1 of volgens een Gaussiaanse distributie, enzovoorts.

Constructor Summary

Random()	Creates a new random number generator.
Random(long seed)	Creates a new random number generator using a single long seed.

Method Summary

boolean	**nextBoolean**() Returns the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence.
void	**nextBytes**(byte[] bytes) Generates random bytes and places them into a user-supplied byte array.
double	**nextDouble**() Returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence.
float	**nextFloat**() Returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence.
double	**nextGaussian**() Returns the next pseudorandom, Gaussian ("normally") distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.
int	**nextInt**() Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence.
int	**nextInt**(int bound) Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.
long	**nextLong**() Returns the next pseudorandom, uniformly distributed long value from this random number generator's sequence.

void	setSeed(long seed) Sets the seed of this random number generator using a single long seed.

1.2.3. De ArrayList

De ArrayList is een extensie van "List"; deze beschrijft een lijst om elementen bij te houden.

Het aanmaken van een *ArrayList* van integers gebeurt zo:

```
ArrayList<Integer> array = new ArrayList<Integer>();
```

Met de constructor geef je tussen de '<' en '>' het type van de elementen mee. Het type van de elementen is dus een *parameter* die je als gebruiker meegeeft. Dit heet in Java ***generics***. Men noemt het ook een *template*: de *ArrayList* is gedefinieerd, onafhankelijk van het type van de elementen. Je programmeert een soort template die dan met het feitelijke type wordt geïnstantieerd. We gaan hier nog veel gebruik van maken. Voor deze generics moet je wel de klassevariant van het primitief type gebruiken:

Primitief type	Klasse
int	Integer
float	Float
double	Double
char	Char
boolean	Boolean

De documentatie toont ons nog een tweede constructor. Eentje die we de initiële capaciteit kunnen meegeven. Voor de eerste constructor, de default constructor, wordt een initiële capaciteit van 10 gebruikt:

Constructor Summary

ArrayList() Constructs an empty list with an initial capacity of ten.
ArrayList(int initialCapacity) Constructs an empty list with the specified initial capacity.

Als we na het aanmaken de grootte testen met *size()* geeft Java nog steeds 0 aan! Java heeft een initiële geheugencapaciteit gealloceerd, maar gaat die pas gebruiken als je erom vraagt. Die capaciteit wordt gevuld met de elementen die je toevoegt. Dat doe je met *add(E e)*. *E* is het type dat je meegaf met de constructor; in ons voorbeeldje *Integer*. Met het opvullen vergroot *ArrayList* de 'zichtbare' grootte. Intern heeft hij meer geheugen ter beschikking, *ArrayList* houdt bij hoeveel elementen je hebt toegevoegd en laat dit zien met *size()*. Dat is in feite heel handig. Als de *ArrayList* op een gegeven

moment te weinig geheugen heeft, zal hij intern een grotere array aanmaken. Als gebruiker merk je daar echter niets van, behalve dat zo'n vergroting toch wat tijd kost.

Hier de belangrijkste methoden van de *ArrayList*:

Method Summary	
boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns `true` if this list contains no elements.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.

Elementen toevoegen doe je dus met *add(E e)*. Met *add(int index, E e)* voeg je een element toe op een bepaalde plaats, een *insert* dus. Elementen opvragen doe je met *get()*. Je kunt ook checken of een element aanwezig is (*contains*), of zijn index opvragen (*indexOf*). Een bepaald element op een bepaalde positie vervangen door een ander doe je met *set*. Elementen verwijder je met *remove*.

Hieronder volgt een voorbeeld van het gebruik van een ArrayList die een sequentie uitrekent volgens "het vermoeden van Collatz". Gegeven een natuurlijk getal n_0 als startwaarde, bereken de sequentie van n_i volgens de onderstaande regel totdat je 1 bekomt:

$$n_{i+1} = \begin{cases} {}^{n_i}/_2 & \text{als } n_i \text{ even} \\ 3n_i + 1 & \text{als } n_i \text{ oneven} \end{cases}$$

Het is nog niet bewezen dat de sequentie uiteindelijk 1 wordt voor *elk* natuurlijk getal; het is dus ook niet geweten hoe lang de sequentie zal zijn, wat een ArrayList zeer geschikt maakt voor deze opgave.

```
List<Integer> collatz = new ArrayList<Integer>();
int n = 77; // startwaarde
```

```
while (n != 1)
{
	collatz.add(n);
	if (n % 2 == 0) {
		n /= 2;		// even geval
	}
	else {
		n = 3*n + 1; // oneven geval
	}
}

collatz.add(1);
System.out.println(collatz);
```

1.2.4. Een verzameling: 'set'

De Set-klasse van Java stelt een wiskundige verzameling voor. Ze is als volgt gedefinieerd (zie Javadocumentatie):
"*A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element.*"

Dit zijn de belangrijkste methoden:

Method Summary	
boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns true if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
boolean	**isEmpty**() Returns true if this set contains no elements.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).

boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).

De typische bewerkingen van sets kunnen uitgevoerd worden met de gegeven methods:
- De unie: (a ∪ b) bevat alle elementen die tot a of b behoren. Hiervoor voeren we een *AddAll* uit.
- De intersectie of doorsnede: (a ∩ b) bevat alle elementen die tot a en b behoren. Hiervoor voeren we een *RetainAll* uit.
- Het verschil: (a \ b) bevat alle elementen die tot a en niet tot b behoren. Hiervoor voeren we een *RemoveAll* uit.
- Het symmetrische verschil bevat alle elementen die tot een verzameling en niet tot de andere verzamelingen behoren. Hiervoor bestaat niet één methode. We maken de unie van de twee verschillen: (a \ b) ∪ (b \ a).

Hierbij komt ook nog de booleaanse operator *contains* die toelaat na te gaan of een element al of niet aanwezig is in een verzameling.

Let op: de *Set* is een ***interface***, geen echte klasse (zie Javaregels). Ze definieert het 'contract' van de klasse, het zegt wat een *Set* allemaal moet kunnen. Dat zijn de methodes die we gaan gebruiken. Een interface legt al die methodes vast zonder te zeggen hoe ze uitgevoerd moeten worden. De code ziet er als volgt uit:

```
public interface Set<E> extends Collection<E> {

    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean retainAll(Collection<?> c);
    boolean removeAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```

Om een set-object aan te maken, moeten we een 'echte' klasse kiezen die de *Set*-interface implementeert, en die dus voor alle bovenstaande methodes een implementatie voorziet. Twee implementaties die je kunt gebruiken, zijn *TreeSet* en *HashSet*. Momenteel moet je enkel weten dat deze twee OK zijn, je kunt ze gebruiken en je weet dat ze aan het *Set*-contract voldoen. In hoofdstuk 8 komen we hier in het kort op terug. Dan weet je ook wat een *Tree* en een *Hashfunctie* zijn.

Om het gebruik van sets verder te illustreren, hebben we de zeef van Erathostenes geschreven met een verzameling die alle priemgetallen bij houdt:

```
	public static void main(String[] args) {
		System.out.print("Geef range: ");
		Scanner scanner = new Scanner(System.in);
		int N = scanner.nextInt();

		Set<Integer> getallen = new HashSet<Integer>();
		for(int i=2;i<N;i++) // vanaf 2
			getallen.add(i);

		// de zeef
		for(int i=2;i<N;i++){
			// doe voor alle tot dan toe gevonden priemgetallen
			if (getallen.contains(i)){
			// veelvouden van priemgetal zijn geen priemgetal
				int m = i * 2;
				while(m < N){
					getallen.remove(m);
					m = m + i;
				}
			}
		}
		// print array
		System.out.print("Er zijn "+getallen.size()+"
priemgetallen kleiner dan "+N+": "+getallen);
	}
```

In hoofdstuk 2 zien we een versie geschreven met arrays: we onthouden voor elk getal een *boolean* die aanduidt of een getal een priemgetal is. Hier houden we de priemgetallen zelf bij. We vullen eerst de set met alle getallen en verwijderen vervolgens de veelvouden van de reeds gevonden priemgetallen.
Merk op dat ik ook het aantal priemgetallen afprint. Deze wordt gegeven door *size()*. Terwijl in de array-implementatie je zelf dat aantal moet berekenen door de true's te tellen in de array.

1.2.5. Mappen

Een laatste algemene en zeer nuttige datastructuur is een **map**.
In een map link je dingen (*objecten* in een object-georiënteerde taal) aan elkaar. Meestal link je objecten van één soort met objecten van een andere soort. Een voorbeeld: je wilt een project linken aan een student. In Excel doe je dit zo:

Ida, Jennifer	Oprolbare brug
Laurent, Maarten	Ultragrabber
Manuka, Stani	Moving Tower Ahmedabad
Mitichashvili, Tornike	piramide
Nguyen, Kim	trap Villa Savoye
Pelicaen, Erik	achthoekstructuur
Perez Sotomayor, Lucia	Origami

Hier zijn de studenten alfabetisch geordend. Ik kan dus een student zoeken en zijn project afleiden uit de tabel.

In Java noemen we het eerste veld de *key* en het tweede de *value*. Java laat toe dat je elk mogelijk type kunt gebruiken. De Java Map is als volgt gedefinieerd (een interface!):

Interface Map<K,V>

Met Type Parameters:

`K` - the type of *keys* maintained by this map
`V` - the type of mapped *values*

Dit maakt gebruik van generics met twee parameters, K en V. De methodes van de interface geven de dingen aan die je met een map kunt doen. De belangrijkste zijn *put* en *get*, waarmee je *key-value* koppels toevoegt en afhaalt. De *get* gebeurt aan de hand van de *key*: je geeft een *key* op en de map geeft je de *value* terug. Met *containsKey()*ga je na of een *key* in de map aanwezig is.

Hieronder vind je de definities van alle methodes:

`void`	**`clear`**`()` Removes all of the mappings from this map (optional operation).
`boolean`	**`containsKey`**`(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	**`containsValue`**`(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	**`entrySet`**`()` Returns a `Set` view of the mappings contained in this map.
`boolean`	**`equals`**`(Object o)` Compares the specified object with this map for equality.
`V`	**`get`**`(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`int`	**`hashCode`**`()` Returns the hash code value for this map.
`boolean`	**`isEmpty`**`()` Returns `true` if this map contains no key-value mappings.
`Set<K>`	**`keySet`**`()` Returns a `Set` view of the keys contained in this map.
`V`	**`put`**`(K key, V value)` Associates the specified value with the specified key in this map (optional operation).
`void`	**`putAll`**`(Map<? extends K,? extends V> m)` Copies all of the mappings from the specified map to this map (optional operation).

V	**remove**(Object key) Removes the mapping for a key from this map if it is present (optional operation).
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a Collection view of the values contained in this map.

Als je een Map wilt aanmaken moet je een concrete implementatie van deze interface kiezen. Kies de HashMap of TreeMap. Later in de cursus leggen we uit hoe deze werken.

In hoofdstuk 4 zien we een meer geavanceerd voorbeeld van interfaces (Functies).

1.2.6. Generics

Stel dat je een methode wil maken die alle elementen van een array uitprint. Afhankelijk van het type van de elementen zou je dan een verschillende methode kunnen aanmaken:

```
void printArray( Integer[] input )
void printArray( String[] input )
void printArray( Student[] input )
…
```

Dit is natuurlijk zeer redundant en onpraktisch. Een oplossing voor dit probleem is het gebruik van “generics” (ook soms wel templates genoemd): hier gaan we een methode (of een klasse) declareren waarin één (of meerdere) types generisch zijn (zoals een jokerteken waarin elke klasse kan worden ingevuld). Dit wordt gedaan m.b.v. “<” en “>” symbolen:

```
public <T> void printArray( T[] input )
{
    for ( T element : input ){
          System.out.print( element + ", ");
    }
    System.out.println();
}
```

In het bovenstaande voorbeeld is het “jokerteken” gelijk aan “T” (je mag hier een willekeurige string voor invullen). De bovenstaande methode zal werken voor elke array wat zijn type ook, zolang tenminste de methode “toString()” gedefinieerd is.

Naast methodes kan je verder ook klassen definiëren met generics. Dit wordt vaak gebruikt voor “containerklassen”, namelijk klassen die gemaakt zijn om meerdere objecten op een gestructureerde manier bij te houden, zoals: matrices, tupels, lijsten, sets, maps, enz. (een aantal voorbeelden volgen in de komende subsecties).
Een voorbeeld van zo’n zelfgemaakte klasse is:

```
public class MyPair<X> {

    private X first, second;

    public MyPair(X first, X second) {
        this.first = first;
        this.second = second;
    }

    public X get_first() { return first; }
    public X get_second() { return second; }

    public String toString() {
        return "{" + first + ", " + second + "}";
    }
}
```

Deze klasse stelt een paar voor die twee elementen van hetzelfde type bevat. Als je bv. een MyPair wil aanmaken met twee integers 7 en 8, typ je:

```
MyPair<Integer> pair = new MyPair<Integer>(7,8);
```

Noot: je kan bij generics geen primitieve types gebruiken als parameter. Hiervoor moet je gebruik maken van de klassenvariant van het primitieve type:

Primitief type	Klasse
int	Integer
float	Float
double	Double
char	Char
boolean	Boolean

Er kan nog veel meer gedaan worden met generics, maar dit zal niet verder behandeld worden in deze cursus.

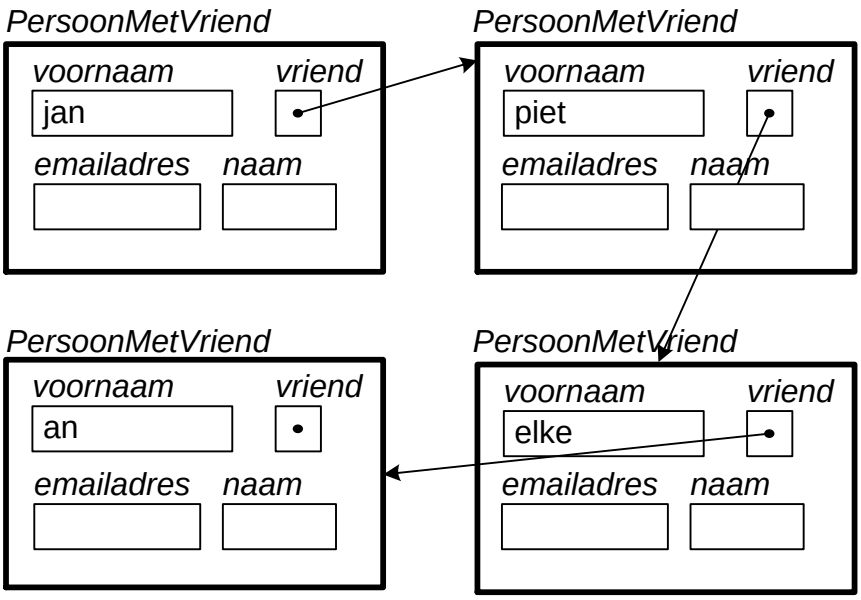
1.3. Overerving en Overschrijven: Vrienden-voorbeeld

In het volgend voorbeeld illustreren we de mogelijkheid om overgeërfde methodes te overschrijven (veranderen). Stel dat we de vriendschapsverbanden willen weergeven (zoals bij sociale netwerksites). Om de code eenvoudig te houden gaan we er even van uit dat elke persoon maar 1 vriend heeft. Dit is eenvoudig uit te breiden naar meerdere vrienden d.m.v. ArrayLists, maar dit zou de code complexer maken.

We maken een nieuwe klasse die het attribuut vriend toevoegt aan de klasse Persoon (zie **package *objectgeorienteerd***):

```
public class PersoonMetVriend extends Persoon {

	//========== ATTRIBUTEN =============//
	PersoonMetVriend vriend; // een persoon heeft maar 1 vriend

	//========= CONSTRUCTOR =============//
	PersoonMetVriend(String voornaam){
		super(voornaam, ""); // oproep constructor van Persoon
		vriend = null; // heeft bij default geen vriend
	}
	PersoonMetVriend(String voornaam, String naam){
		super(voornaam, naam); // oproep constructor van Persoon
		vriend = null; // heeft bij default geen vriend
	}
	//========= METHODES ============//
	boolean kenJeDiePersoonViaVia(PersoonMetVriend persoon){
		if (persoon == vriend)
			return true;
		else if (vriend != null)
			return vriend.kenJeDiePersoonViaVia(persoon);
		else
			return false;
	}

	public String toString(){
		return voornaam+"("+vriend+")";
	}
}
```

We voorzien 2 constructors: eentje waarbij je enkel de voornaam geeft en eentje waarbij je ook de (familie)naam meegeeft. In het eerste geval blijft de naam blanco. Merk op dat per default het attribuut vriend op *null* staat, maw de persoon heeft geen vriend. Vervolgens willen we het volgende vriendennetwerk bouwen:



Dit gebeurt adhv de volgende code:

```
public class OefeningVrienden {
	//======================== PROGRAMMA ============================//
	public static void main(String[] args) {
		PersoonMetVriend jan = new PersoonMetVriend("jan");
		PersoonMetVriend piet = new PersoonMetVriend("piet");
		PersoonMetVriend elke = new PersoonMetVriend("elke");
		PersoonMetVriend an = new PersoonMetVriend("an");

		jan.vriend = piet;
		piet.vriend = elke;
		elke.vriend = an;

		boolean kentJanAn = jan.kenJeDiePersoonViaVia(an);
		System.out.println(jan+" kent "+an+"? "+kentJanAn);
	}
}
```

We maken eerst 4 personen aan en vervolgens vullen we het attribuut *vriend* in. Op de twee laatste lijntjes gaan we iets doen met het sociaal netwerk. We gaan nagaan of een persoon gelinkt is met een ander persoon via zijn vrienden. Hiervoor hebben we de methode *kenJeDiePersoonViaVia* toegevoegd aan de klasse *PersoonMetVriend*. De methode *kenJeDiePersoonViaVia* begint met na te gaan of de persoon die we zoeken gelijk is aan de vriend. Als dit niet het geval is en de persoon een vriend heeft, roepen we de methode recursief op om na te gaan of de vriend de gezochte persoon kent. Indien geen vriend, geven we *false* terug.

De bovenstaande code `boolean kentJanAn = jan.kenJeDiePersoonViaVia(an);` zal de methode eerst toepassen op object *jan*, vervolgens op *piet* en tot slot op object *elke*. *Elke* heeft *an* als vriend en geeft dus *true* terug. Bij het terugkeren uit de recursie geven we die *true* door en de variabele `kentJanAn` krijgt de waarde *true*.

We voegen ook een *toString*-methode toe om de objecten netjes af te printen.

We gaan twee varianten aanmaken om de mogelijkheden van overerving te tonen. We creëren een persoon die steeds ontkent de persoon te kennen (*ontkenner*) en een persoon die liegt (*leugenaar*).

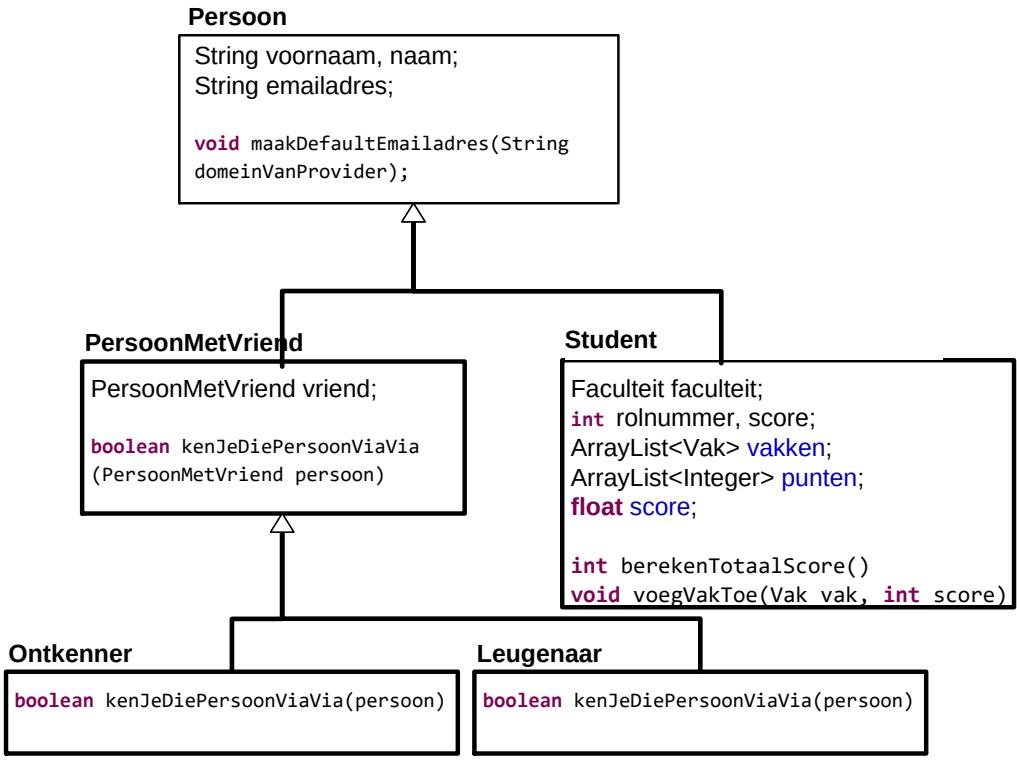
```
public class Ontkenner extends PersoonMetVriend {

	Ontkenner(String naam) {
		super(naam);
	}
	boolean kenJeDiePersoonViaVia(PersoonMetVriend vriend){
		return false;
	}
}
```

We voegen een constructor toe en overschrijven de methode `kenJeDiePersoonViaVia`. Deze methode geeft altijd *false* terug. Dit is alles wat we moeten programmeren, alle andere functionaliteiten erft de klasse over van *PersoonMetVriend*.

```
public class Leugenaar extends PersoonMetVriend {

	Leugenaar(String naam) {
		super(naam);
	}
	boolean kenJeDiePersoonViaVia(PersoonMetVriend vriend){
		return !super.kenJeDiePersoonViaVia(vriend);
	}
}
```

Een leugenaar verdraait de waarheid. Daarvoor wordt de originele code opgeroepen: de methode *kenJeDiePersoonViaVia* van *PersoonMetVriend*. Dit doe je met super, het is namelijk de methode van de superklasse. Het resultaat wordt omgekeerd.

Het volgend schema toont de klassehiërarchie van de gemaakte klassen.



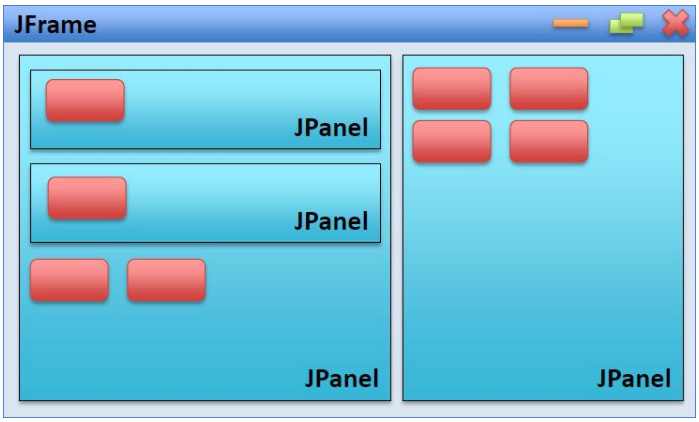
Nu gaan we `KlasseOefening1 & KlasseOefening2 & InterfaceOefening` oplossen. Deze staan achteraan dit hoofdstuk.

1.4 GUI

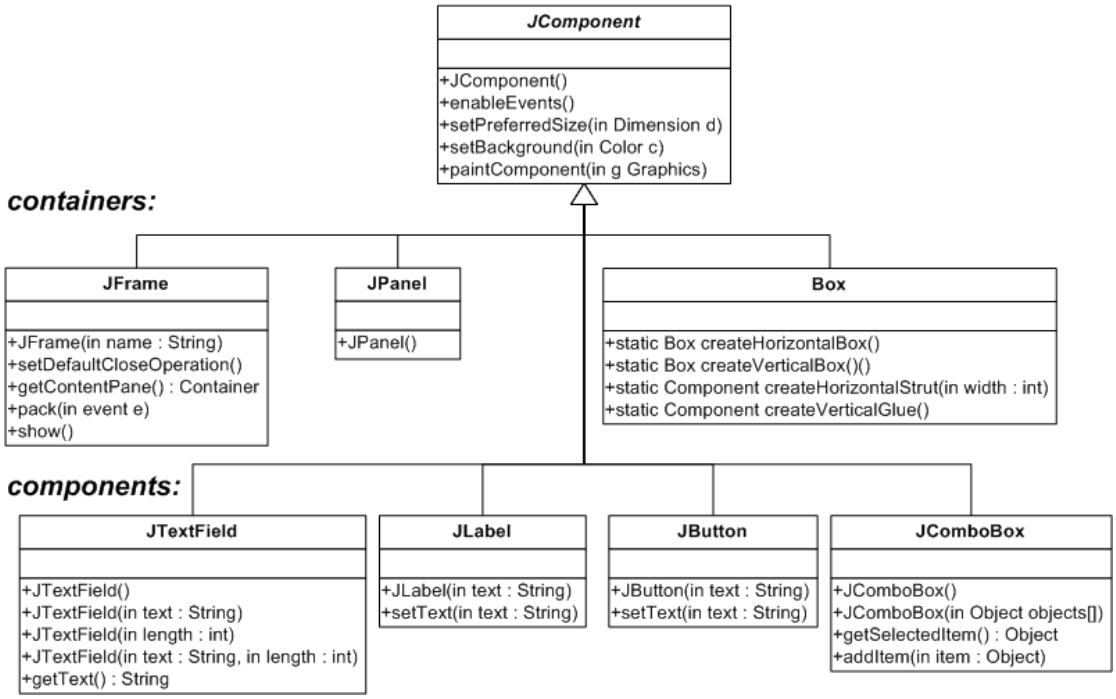
In Java zijn er uitgebreide libraries voorzien voor het maken van Graphical User Interfaces (GUI). Deze libraries zijn zeer groot, waardoor we maar een klein deel kunnen overlopen. In deze sectie bekijken we hoe je een eenvoudige GUI kunt maken met knoppen en invoervakjes, vormen kan tekenen en animaties kan maken. De javalibrary die wij gebruiken is Swing.

1.4.1. JFrames en JPanels

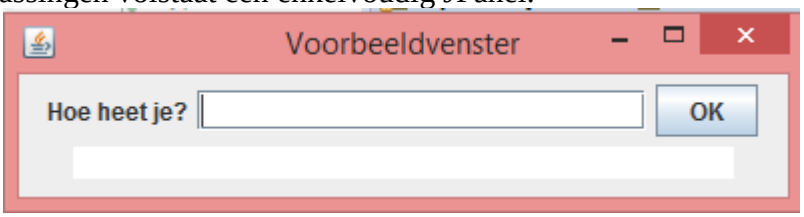
Twee belangrijke componenten van de "Swing"-library zijn JFrames en JPanels. Een JFrame is een klasse die gebruikt wordt als een venster: je kan het venster dimensioneren, een titel geven, knoppen toekennen, enz. JPanels daarentegen zijn "panelen" waarin elementen zoals knoppen, sliders, e.d. geplaatst kunnen worden. JPanels kunnen ook genest worden (zie onderstaand schema).



Hieronder een gesimplifieerd schema met de belangrijkste Swing-componenten. Enerzijds zijn er de containers (zoals JFrame en JPanel): componenten waarop je andere componenten plaatst. Anderzijds zijn er de basiscomponenten van de GUI: knoppen, labels, tekstveldjes, enzovoorts.



In vele toepassingen volstaat een enkelvoudig JPanel.

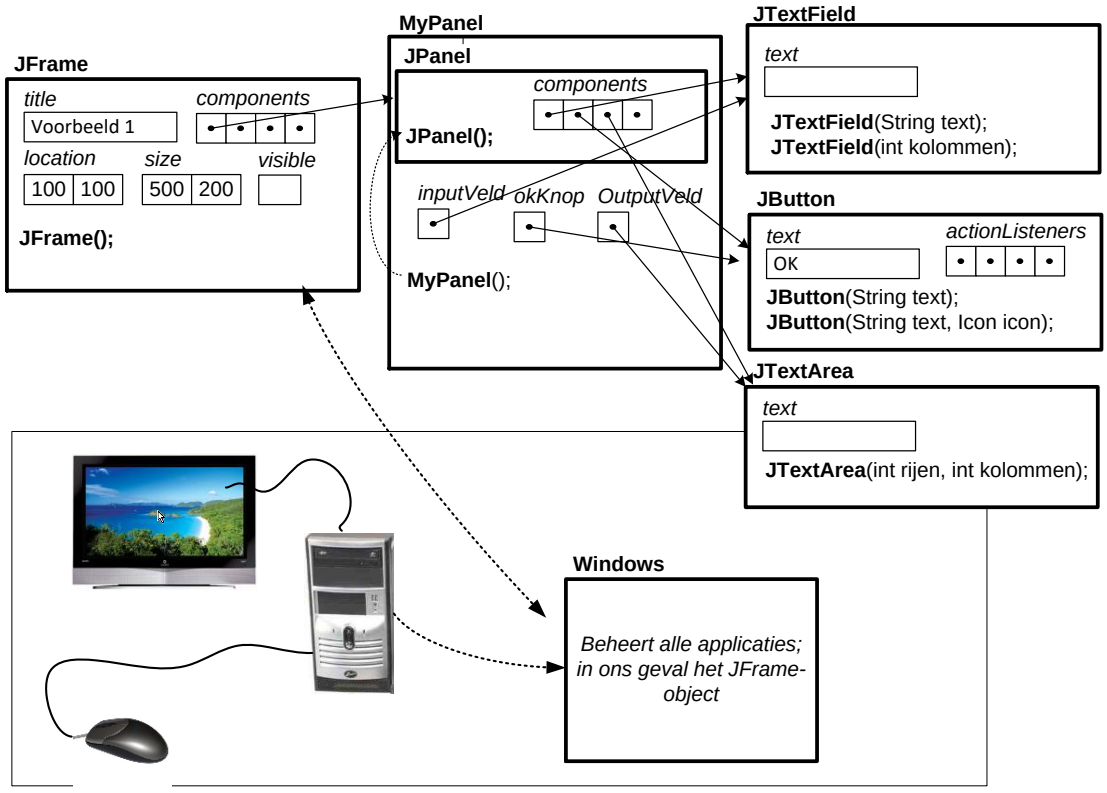


We hebben een enkelvoudig JPanel met de volgende elementen: een tekstlabel (JLabel), invoerveldje (JTextField), een knop (JButton) en een uitvoerveld (JTextArea).

```
import javax.swing.*;

public class MyPanel1 extends JPanel {

  // GUI-componenten
  private JTextField inputVeld;
  private JButton okKnop;
  private JTextArea outputVeld;

  public MyPanel1()
  {
    inputVeld = new JTextField(20);  // groote veld (# symbolen)
    okKnop = new JButton("OK");  // inhoud knopje
    outputVeld = new JTextArea(1,30); // dimensies veld (rijen, kolommen)

    // toevoegen van veldjes aan de GUI
    this.add(new JLabel("Hoe heet je?")); // Jlabel: tekst op je GUI
    this.add(inputVeld);
    this.add(okKnop);
    this.add(outputVeld);
  }
```

```
  public static void main(String[] args)
  {
    JFrame frame = new JFrame();
    frame.setSize(400, 100);  // dimensies van het venster
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // zorgt dat het
programma afsluit bij het klikken op de 'X'-knop
    frame.setTitle("Voorbeeldvenster");  // titel van het venster
    frame.setLocation(100, 100); // positie van de linkerbovenhoek van het
venster op het bureaublad

    JPanel hoofdpaneel = new MyPanel1();  // maak MyPanel1-object aan
    frame.add(hoofdpaneel);  // stop het in de Frame

    frame.setVisible(true);  // het venster wordt geactiveerd
  }
}
```

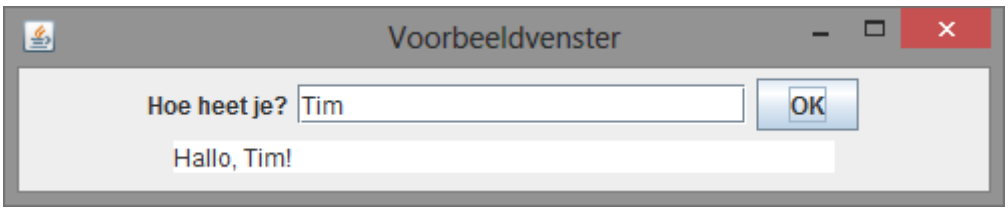
In de constructor voegen we de componenten toe aan onze JPanel-subklasse. In de main maken we een JFrame aan waarin de JPanel komt. Dit resulteert in de volgende objecten:



De GUI bestaat dus uit 5 componenten die hiërarchisch gekoppeld zijn: de 3 basiscomponenten in de JPanel die dan weer in de JFrame zit (het *components*-attribuut). Onze eigen klasse MyPanel is een subklasse van JPanel en erft dus alles over. De 3 basiscomponenten worden al bijgehouden in de *components*-lijst maar omdat we niet aan die lijst kunnen, houden we elke component ook bij in onze eigen MyPanel. Het schema toont ook de constructors. Vanuit MyPanel() wordt de superconstructor JPanel() opgeroepen.

1.4.2. Events

We willen dat er een actie wordt uitgevoerd wanneer er op de OK-knop gedrukt wordt: de opgegeven naam wordt dan gebruikt in het bericht van JTextArea. Het resulterende venster wordt dan:



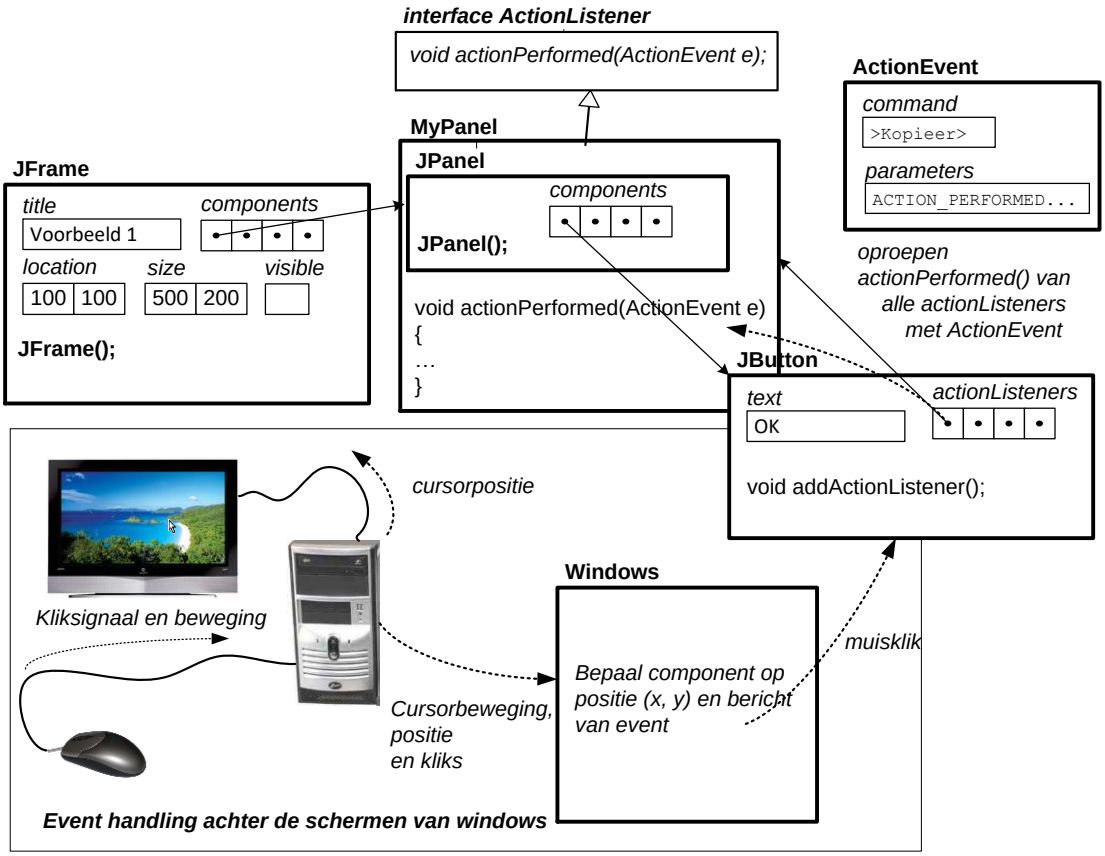
Om dit te bekomen maken we gebruik van het interface "`ActionListener`". Deze is als volgt gedefinieerd:

```
interface ActionListener {

     public void actionPerformed(ActionEvent e);
}
```

We maken `MyPanel2` een subklasse van ActionListener via **`implements`**. Dit verplicht de implementatie van de methode "`actionPerformed(ActionEvent)`", welke zal opgeroepen worden wanneer een element met een ActionListener wordt geactiveerd. Daarom passen we "`addActionListener(this)`" toe op de Jbutton.

```
public class MyPanel2 extends JPanel implements ActionListener {

       // GUI-componenten
       private JTextField inputVeld;
       private JButton okKnop;
       private JTextArea outputVeld;

       public MyPanel2()
       {
              …

              okKnop.addActionListener(this);  // associatie ActionListener

              ...
       }

       // Dit wordt uitgevoerd wanneer er op de knop gedrukt wordt
       public void actionPerformed(ActionEvent e) {
              outputVeld.setText("Hallo, " +  inputVeld.getText() + "!");

       }

       public static void main(String[] args)
       {
              ...
       }
}
```

Implementatie van interface `ActionListener` door `MyPanel` verplicht ons om de `actionPerformed`-methode te implementeren. Vervolgens voegen we `MyPanel` toe aan de `JButton` als actionListener (wordt bijgehouden in een lijst van actionListeners).



Als er nu met de muis geklikt wordt, bepaalt Windows op welk component dit gebeurde. Dit component (hier: de `JButton`) zal vervolgens de `actionPerformed` oproepen van alle toegevoegde actionListeners. In ons geval wordt dus onze eigen `actionPerformed`-code uitgevoerd. Een actionListener 'luistert' naar de acties van een GUI-component. Op een gelijkaardige manier verbindt je acties aan toetsen (`KeyListener`) of muisbewegingen (`MouseMotionListener`).

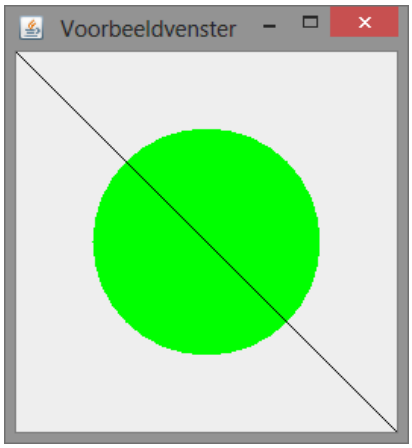
1.4.3. Graphics

Alle componenten in de Swing-library (JFrames, JPanels, enz.) hebben een methode "`paintComponent(Graphics g)`" waaraan een Graphics-object wordt meegegeven. Dit Graphics-object bevat een referentie naar het scherm dat gebruikt wordt om ernaar te tekenen. Deze methode kan overschreven worden, zodat we zelfgekozen dingen naar het scherm kunnen tekenen. Een voorbeeld:

```
public class MyPanel3 extends JPanel {

	public void paintComponent(Graphics g) {
		g.setColor(Color.GREEN);
		g.fillOval(50,50,150,150);
		g.setColor(Color.BLACK);
		g.drawLine(0,0,250,250);
	}
```

```
        public static void main(String[] args) {
                ...
        }
}
```

Met "`setColor`" stel je de kleur in waarin objecten getekend worden. "`drawLine`" tekent een lijn gegeven de pixelcoordinaten (x_1, y_1) en (x_2, y_2). "`fillOval`" tekent een gevulde ovaal gegeven een middelpunt, breedte en hoogte. Dit geeft het volgende resultaat:

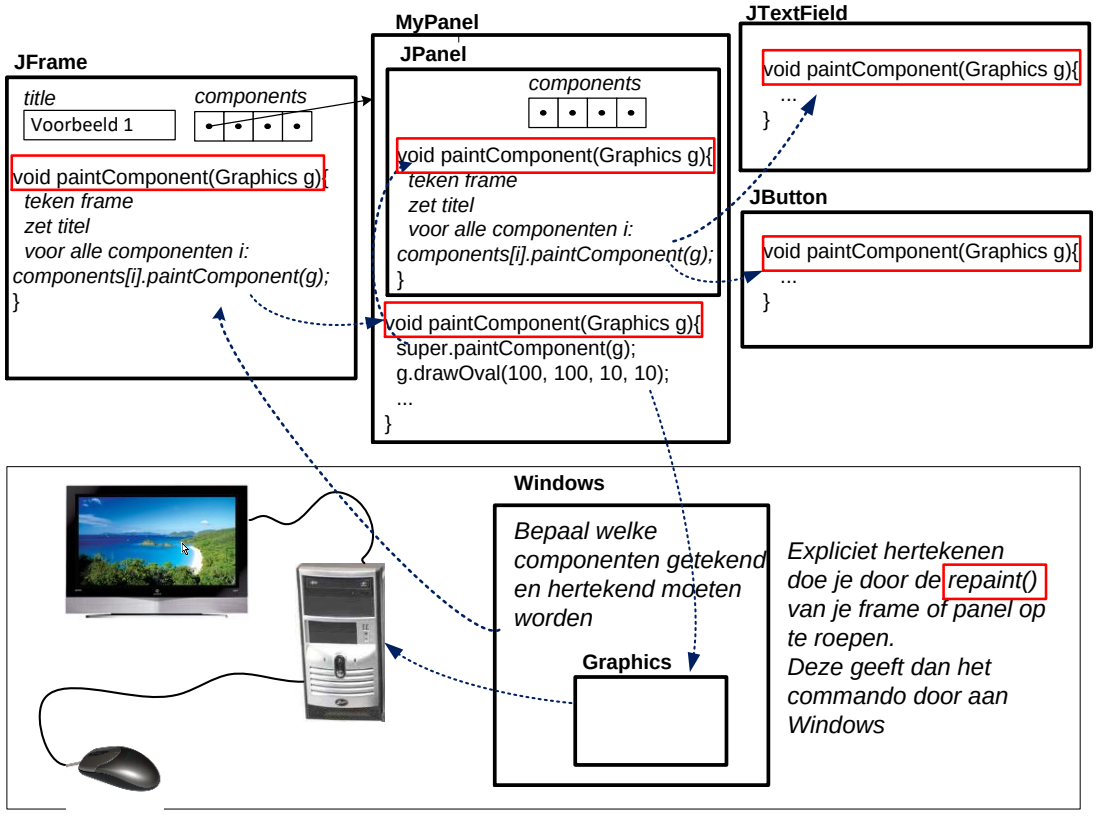


Er bestaan vele methodes voor het tekenen naar het scherm. Hieronder een selectie:

Method Summary	
`abstract void`	**`clearRect`**`(int x, int y, int width, int height)` Clears the specified rectangle by filling it with the background color of the current drawing surface.
`void`	**`draw3DRect`**`(int x, int y, int width, int height, boolean raised)` Draws a 3-D highlighted outline of the specified rectangle.
`abstract void`	**`drawArc`**`(int x, int y, int width, int height, int startAngle, int arcAngle)` Draws the outline of a circular or elliptical arc covering the specified rectangle.
`void`	**`drawChars`**`(char[] data, int offset, int length, int x, int y)` Draws the text given by the specified character array, using this graphics context's current font and color.
`abstract boolean`	**`drawImage`**`(`**`Image`**` img, int x, int y, `**`ImageObserver`**` observer)` Draws as much of the specified image as is currently available.
`abstract void`	**`drawLine`**`(int x1, int y1, int x2, int y2)` Draws a line, using the current color, between the points `(x1, y1)` and `(x2, y2)` in this graphics context's coordinate system.
`abstract void`	**`drawOval`**`(int x, int y, int width, int height)` Draws the outline of an oval.
`abstract void`	**`drawPolygon`**`(int[] xPoints, int[] yPoints, int nPoints)` Draws a closed polygon defined by arrays of *x* and *y* coordinates.

abstract void	**drawPolyline**(int[] xPoints, int[] yPoints, int nPoints) Draws a sequence of connected lines defined by arrays of *x* and *y* coordinates.
void	**drawRect**(int x, int y, int width, int height) Draws the outline of the specified rectangle.
abstract void	**drawRoundRect**(int x, int y, int width, int height, int arcWidth, int arcHeight) Draws an outlined round-cornered rectangle using this graphics context's current color.
abstract void	**drawString**(**String** str, int x, int y) Draws the text given by the specified string, using this graphics context's current font and color.
void	**fill3DRect**(int x, int y, int width, int height, boolean raised) Paints a 3-D highlighted rectangle filled with the current color.
abstract void	**fillArc**(int x, int y, int width, int height, int startAngle, int arcAngle) Fills a circular or elliptical arc covering the specified rectangle.
abstract void	**fillOval**(int x, int y, int width, int height) Fills an oval bounded by the specified rectangle with the current color.
abstract void	**fillPolygon**(int[] xPoints, int[] yPoints, int nPoints) Fills a closed polygon defined by arrays of *x* and *y* coordinates.
abstract void	**fillRect**(int x, int y, int width, int height) Fills the specified rectangle.
abstract void	**fillRoundRect**(int x, int y, int width, int height, int arcWidth, int arcHeight) Fills the specified rounded corner rectangle with the current color.
abstract **Color**	**getColor**() Gets this graphics context's current color.
abstract **Font**	**getFont**() Gets the current font.
abstract void	**setColor**(**Color** c) Sets this graphics context's current color to the specified color.
abstract void	**setFont**(**Font** font) Sets this graphics context's font to the specified font.
abstract void	**translate**(int x, int y) Translates the origin of the graphics context to the point (*x*, *y*) in the current coordinate system.

Het volgend schema geeft weer wat er gebeurt:



Windows bepaalt dat de GUI getekend moet worden: bij creatie of bij oproep van `repaint()` bijvoorbeeld. Windows roept dan de `paintComponent` van de `JFrame` op. Deze zal op zijn beurt de `paintComponent` van al zijn componenten oproepen. `JPanel` is ook een container en zal zichtzelf tekenen en ook de `paintComponent` van al zijn componenten oproepen. Door het overschrijven van de `paintComponent` van `JPanel` kunnen we onze eigen tekeningen maken. We moeten wel nog de originele `paintComponent` oproepen. Dit doen we met `super.paintComponent()`.

1.4.4. Animaties en Timers

Animaties worden gerealiseerd door het beeld meermaals per seconde te updaten. Elke "frame" moeten de posities van alle getekende objecten geüpdatet worden. Om dit periodisch te doen kunnen we gebruik maken van Timers.

Een Timer krijgt een "TimerTask" toebedeeld, via de methode "`schedule(TimerTask task, long delay, long period)`". De waardes `delay` en `period` drukken respectievelijk de initiële vertraging en de periode van de uitvoer van de `TimerTask` uitgedrukt in milliseconden. Met "`cancel()`" wordt de Timer stopgezet.

Een "TimerTask" is een abstracte klasse (zie 1.6.4. voor meer info), waarvan de abstracte methode "run()" overschreven moet worden. Hierin wordt de code geschreven die elke stap uitgevoerd moet worden.

In het onderstaande voorbeeld maken we een animatie van een draaiende rode schijf. De code van het paneel is:

```
public class MyPanel4 extends JPanel {

	private final int bigradius = 300, smallradius = 100;
	private double t = 0;	// drukt de tijd uit

	MyPanel4() {
		Timer timer = new Timer();
		timer.schedule(new MyTimerTask(this), 0, 20);
	}

	public void updateme() {
		t += 0.05;	// we incrementeren t met een zekere delta_t
		repaint();	// hiermee laten we het beeld hertekenen
	}

	public void paintComponent(Graphics g) {
		super.paintComponent(g);
		g.setColor(Color.RED);
		g.fillOval(bigradius+(int)(bigradius*Math.cos(t)), bigradius +
(int)(bigradius*Math.sin(t)), smallradius, smallradius);
	}

	public static void main(String[] args)
	{
		...
	}
```

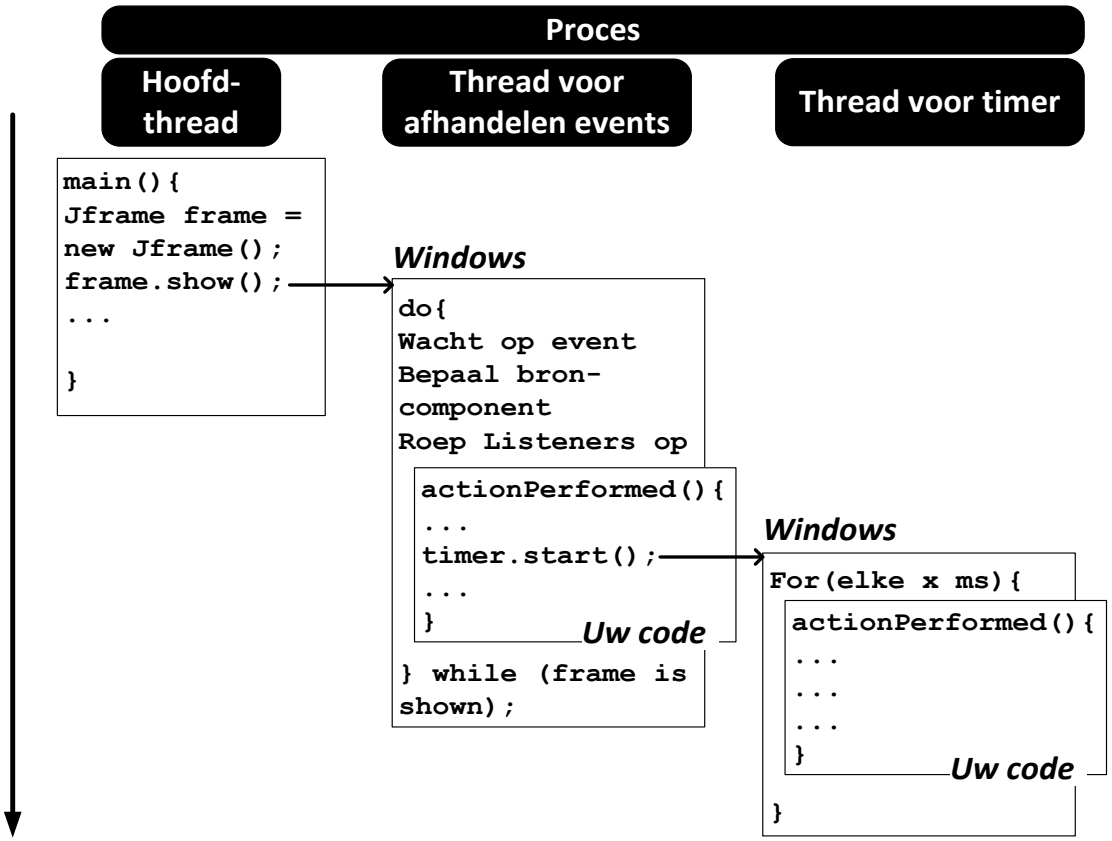
De "`repaint()`" methode zal ervoor zorgen dat de "`paintComponent`" met het juiste Graphics-object opgeroepen wordt. De code van de extensie van de `TimerTask` is:

```
public class MyTimerTask extends TimerTask {

	private MyPanel4 panel;

	MyTimerTask(MyPanel4 panel) {
		this.panel = panel;
	}

	@Override
	public void run() {
		panel.updateme();
	}
}
```

1.4.6. Threads

Een programma voert een sequentie van instructies stap voor stap uit: ze begint met de eerste instructie van de main en stopt nadat de laatste instructie is uitgevoerd. Bij een functie of methode worden al die instructies ook sequentieel uitgevoerd. Het 1-voor-1 uitvoeren van een reeks instructies gebeurt door een *thread* (engels voor draad). Normaal heeft je programma 1 thread.
Eén programma kan echter bestaan uit meerdere *threads*. Elke thread voert een sequentie van instructies uit. Een sequentie kan je zien als een 'draad', vandaar de benaming. Vele moderne toepassingen zijn zelf opgebouwd uit een groot aantal onafhankelijke threads die simultaan uitgevoerd worden. Het tekstverwerkingsprogramma *Word* bvb. gebruikt verschillende threads om teksten en figuren op het scherm te tonen, zodanig dat wanneer men snel door een tekst wenst te lopen men niet hoeft te wachten op het tekenen van alle figuren. Een andere thread zal tegelijkertijd je taalfouten opsporen (de *spellingchecker*).

Een programma met een GUI zal een tweede thread opstarten, en als je een timer gebruikt, dan wordt deze uitgevoerd door een aparte thread. Je programma bestaat dan uit 3 threads:



De windows-threads wordt opgestart na de *show()* en zorgt voor het afhandelen van de events door het oproepen van de listeners. Als hij bezig is met het afhandelen van 1 event zal er geen ander event afgehandeld worden! De timer-thread zal om de x milliseconden de code van de timer uitvoeren.

1.5 Exception handling

Naarmate de complexiteit van een programma toeneemt, kan er steeds meer mislopen in je programma. Veel van die dingen zijn moeilijk te anticiperen en doorheen je code goed af te handelen. Een paar voorbeelden van zo'n problemen zijn:

- Missend of corrupt bestand
- Verkeerde gebruikersinvoer
- Geheugen is op
- Verbinding wordt verbroken
- Falen van een library of een component
- …

Al die problemen doorheen je code constant checken met if-blokken is onhandig of doorgaans zelfs onmogelijk. Anderzijds is het voor vele toepassingen zeker geen optie om het programma gewoon te laten crashen en herstarten. Daarom bestaat er in Java "exception handling". Een fout genereert een exception via de throw-instructie. Het programma zal stoppen en crashen *behalve* als de code beschermt wordt d.m.v. een try-catch(-finally) blok:

```
try {
   //Beschermde code
} catch(ExceptionType1 ex1) {
   //Catch blok
} catch(ExceptionType2 ex2) {
   //Catch blok
} catch(ExceptionType3 ex3) {
   //Catch blok
} finally {
      // het "finally"-blok wordt altijd
      // uitgevoerd, ook na een return statement.
      // Dit blok is optioneel.
}
```

De code die je wil beschermen steek je in het try-blok. Zodra er zich in het try-blok een "exception" produceert, zal het programma uit het try-blok springen en gaan naar het eerstvolgende catch-blok met een overeenkomstig exception-type. Is geen enkele catch-blok van toepassing, zal het programma alsnog crashen. Tenslotte wordt het optionele finally-blok altijd uitgevoerd (typisch wordt dit gebruikt om bestanden/verbindingen proper af te sluiten of om iets in een logfile weg te schrijven).

In het onderstaande voorbeeld wordt een invoervakje in een GUI aangemaakt, waarin de gebruiker een strict positief geheel getal moet invullen. Er kunnen zich twee soorten fouten voordoen:

1. De invoer is geen integer.
2. De invoer is een integer die kleiner of gelijk is aan 0.

De methode "`Integer.parseInt(String)`" zal een "`NumberFormatException`" gooien wanneer de invoer niet kan omgezet worden in een String. Alle resterende expressies

in try-blok worden overgeslagen en de pop-up box met de foutmelding in het eerste catch-blok wordt uitgevoerd. De "`ArithmeticException`" zal typisch aangemaakt worden wanneer er zich een onmogelijke wiskundige operatie voordoet (bv. deling door nul). Maar we kunnen ook zelf exceptions triggeren: dit kan m.b.v. het keyword **`throw`**, zoals aangegeven in het voorbeeld onderaan. Dit gebruiken we om een "`ArithmeticException`" aan te maken wanneer de invoer kleiner dan 0 is. Merk op dat je alle klassen in principe kan "throwen".

Er bestaan honderden soorten exceptions, maar deze worden niet verder in de cursus behandeld. Je vindt hier uitgebreid informatie over online. Meestal zal de compiler een waarschuwing geven (of je zelfs verplichten) dat je een bepaald try-catch blok moet voorzien voor bepaalde expressies die een exception kunnen genereren. Bijvoorbeeld als je iets van file wilt lezen ben je verplicht een try-catch toe te voegen om op te vangen dat de file bvb niet bestaat of dat het lezen onderbroken wordt.

```
private JTextField getalVeld;
…

public void actionPerformed(ActionEvent e) {
       try
       {
          int getal = Integer.parseInt(getalVeld.getText());
          if (getal <= 0)
              throw new ArithmeticException();
          maakGetalSlinger(getal);
       }
       catch(NumberFormatException ex) {
          JOptionPane.showMessageDialog(null, "Input moet een getal zijn",
                           "Foute ingave", JOptionPane.ERROR_MESSAGE);
       }
       catch(ArithmeticException ex) {
          JOptionPane.showMessageDialog(null, "Getal moet groter 0 dan
                     zijn",  "Foute ingave", JOptionPane.ERROR_MESSAGE);
       }
}
```

1.6 Geavanceerde concepten: Static, abstracte klassen, debugger en jar-files.

1.6.1 Het gebruik van static

Het gebruik van static kan uit de hand lopen als dit verkeerd gebeurt. Deze paragraaf probeert hieraan te verhelpen. Een object heeft typisch zijn eigen waarden voor elk attribuut, elke persoon heeft een andere naam. Dit is niet voor een *static* attribuut want die heeft maar 1 waarde over alle objecten heen. In feite hangt die aan de klasse vast. Ook zo met de statische methodes. Je kan het zien als een klasse-attribuut. Je spreekt het aan met de klasse: *Klasse.x* of *Klasse.methode()*. Da's het handige van static: je hebt geen referentie naar een object nodig hebt. Referenties naar objecten hebben we in 1.9 besproken.

De sinus methode van de Math-klasse kan je overal oproepen met Math.sin(x), je hebt geen object nodig. Als de sinus-methode niet statisch was moest je een Math-object aanmaken. In het volgende hoofdstuk gaan we *aantalIteraties* als static attribuut aanmaken en gebruiken.

Door een slecht gebruik van *static* zal je in de knoei geraken met je programma. Vanuit een object kan je static dingen oproepen, maar niet omgekeerd: **vanuit static kan je geen objectattributen of methodes oproepen van een gewoon object!!**
De compiler (Eclipse) zal dit niet toelaten en zal een fout geven: "Cannot make a static reference to the non-static method…". Wat de student dan meestal doet is het attribuut of de methode ook static maken, waardoor op het einde alles static wordt… Da's natuurlijk niet de bedoeling. Merk op dat je wel een niet-statisch ding kan oproepen vanuit een statisch als je statische referenties bijhoudt naar het object. Maar dat is meestal niet de juiste oplossing.

Dit probleem treedt niet als je static logisch gebruikt, waar het voor bedoeld is. Wanneer is het logisch om static te gebruiken?

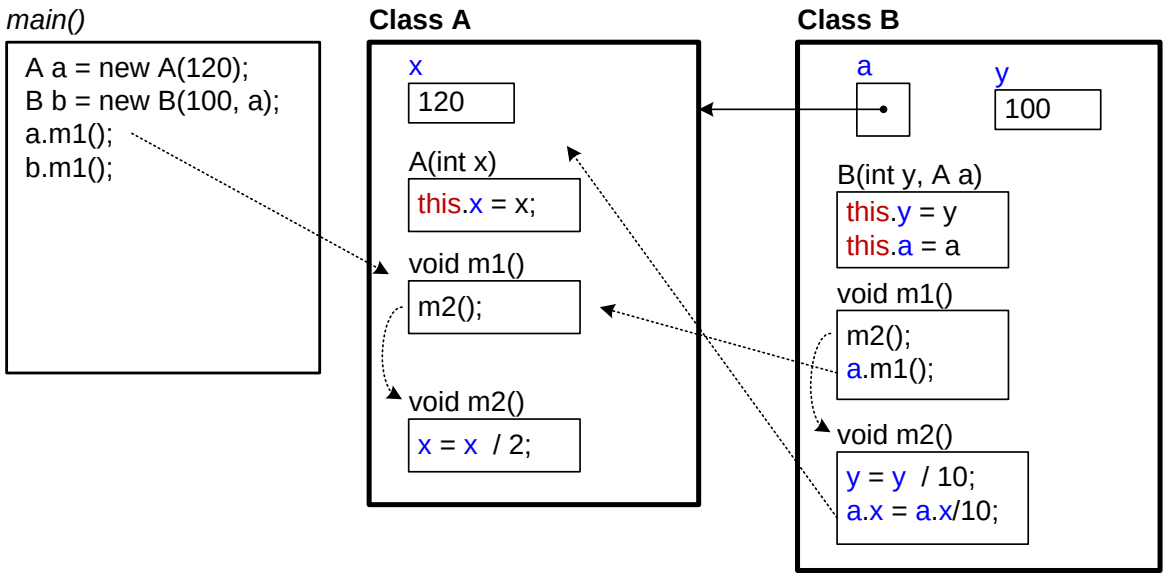
- Algemene functies, die niets aan een object veranderen. Voorbeeld: berekenen gemiddelde, hoek ofzo.
- Algemene, globale variabelen, met 1 waarde die duidelijk algemeen moet zijn over alle objecten heen. Bijvoorbeeld grootte van het scherm, algemene parameters, …
- Als je functie een 2e waarde moet teruggeven geeft static een oplossing (al is die niet al te proper), zie verder in de cursus voor voorbeelden.

Conclusie: let op als je non-static dingen naar static begint te veranderen om uw programma werkend te krijgen!

1.6.2. Oproepen van methodes vanuit objecten

Voor een beginnende programmeur lijkt object-georiënteerde code snel hopeloos complex te worden en worden veel fouten gemaakt. Je moet in feite een mentaal model maken van alle objecten en hun relaties om de code te begrijpen. Deze paragraaf probeert hiertoe bij te dragen. (Als je net begint aan deze cursus kan je deze en de volgende paragrafen best overslaan, grijp er later naar terug).

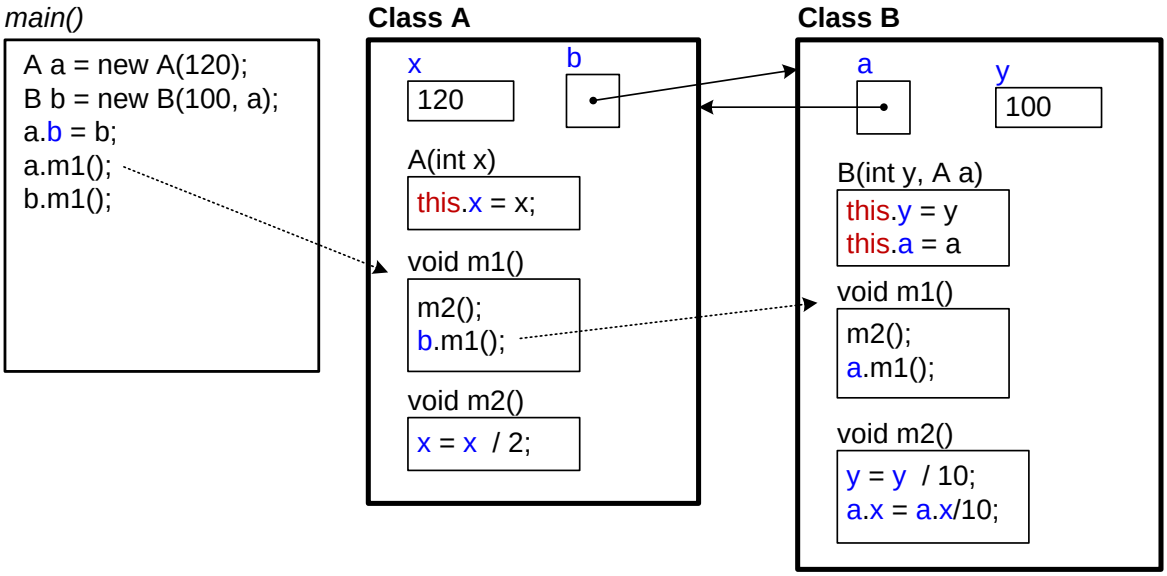
Bestudeer het volgende voorbeeld (**package voorbeelden**):



In de *main()* maken we een object *a* van klasse *A* en een object *b* van klasse *B*. Merk op dat ik de *main()* buiten de klasses teken. We kunnen de *main()* beter beschouwen als iets externs, al moeten we hem in een klasse definiëren. Als we *a.m1()* oproepen, wordt als het ware de controle doorgegeven aan object *a*. We zitten dan in object *a* en kunnen bijvoorbeeld een eigen methode oproepen – hier: *m2()* – of een eigen attribuut veranderen – hier: *x*. Maar in *m1()* kunnen we niets doen met object *b*, want we hebben geen referentie naar *b*!
Vanuit de *main()* kunnen we wel een methode van *b* oproepen, want we hebben een referentie naar het object. We roepen *m1()* van *b* op. Eénmaal in *b* kunnen we eigen methodes oproepen, maar ook methodes van *a*, want nu hebben we wèl een referentie naar *a*!

Dat is de belangrijke les hier: *om objecten te manipuleren moet je er een referentie naar creëren*. Eigen methodes kan je steeds oproepen, methodes van andere objecten niet.
Uit ervaring weet ik dat je eerste grote object-georiënteerde programma nogal snel onoverzichtelijk kan worden. Hierdoor weet je uiteindelijk niet meer waar wat te zetten, en hoe je met objectreferenties om moet gaan, enzovoorts. Het is heel belangrijk om dit 'spelen met objecten' onder controle te krijgen.

Je kunt vervolgens in een situatie terecht komen waarin je wèl vanuit *a* iets met *b* wilt doen. Hoe moet je dit aanpakken? Zoals net gezegd, moet je een referentie naar *b* gaan bijhouden in *a*, anders zie je dat object niet. Hierdoor krijg je een soort kip en ei probleem, je kunt *b* niet meegeven met de constructor van *a*, omdat *b* op dat moment nog niet is aangemaakt. En *b* mag je ook niet eerder aanmaken, want *b* moet *a* hebben. De enige oplossing is om *b* aan *a* mee te geven na het aanmaken van *b*, met een extra instructie.

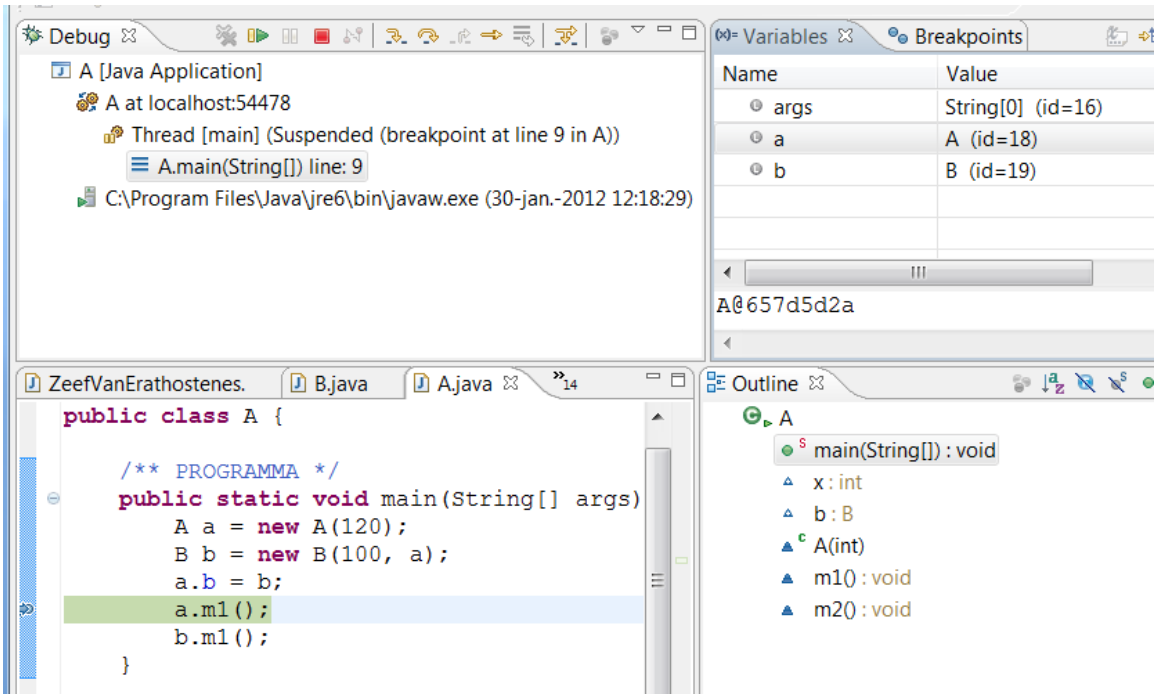


Nu kan *a* wel een methode van *b* oproepen.

1.6.3. Call stack, debugger en geheugengebruik

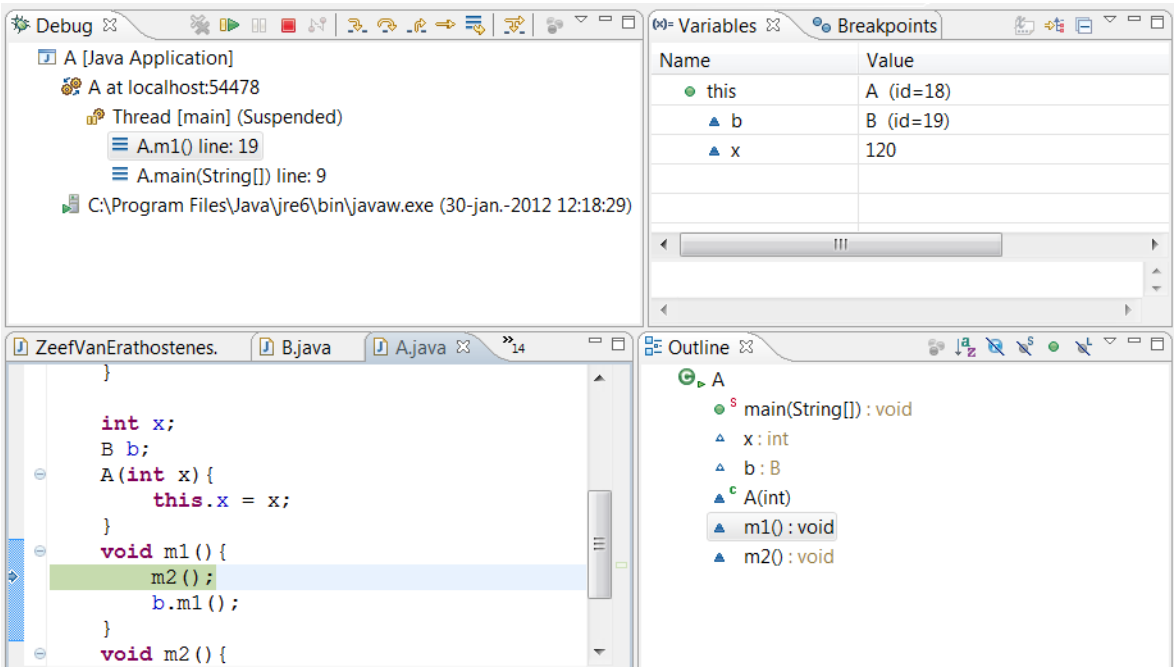
Nu gaan we na hoeveel *geheugen* het programma gebruikt op elk moment van zijn executie, welke *objecten* er aanwezig zijn en welke *methoden* er 'actief' zijn. Als ingenieur is het belangrijk om hier inzicht in te hebben.

Om dit uit te leggen runnen we het programma van 1.1.5 stap-voor-stap met de debugger ( ).

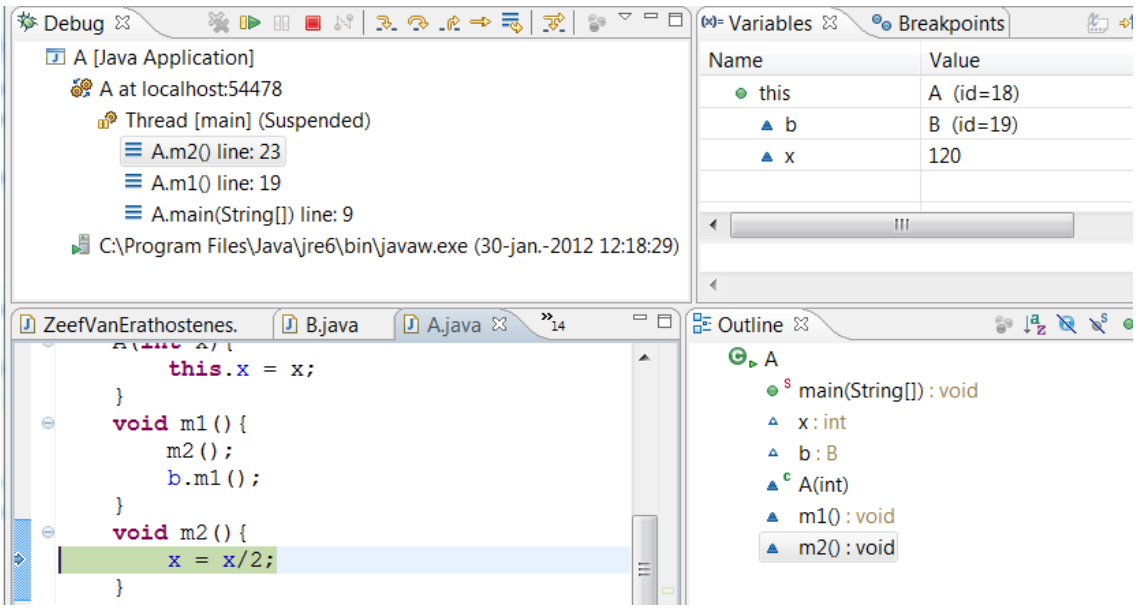


Linksonder zie je de *main()*. Ik heb de debugger laten stoppen op de vierde instructie. Linksboven zie je dat de main van A de actieve methode is. Rechtsboven zie je alle variabelen die je op dat moment `ziet'. Rechtsonder zie je de definitie van de klasse A.

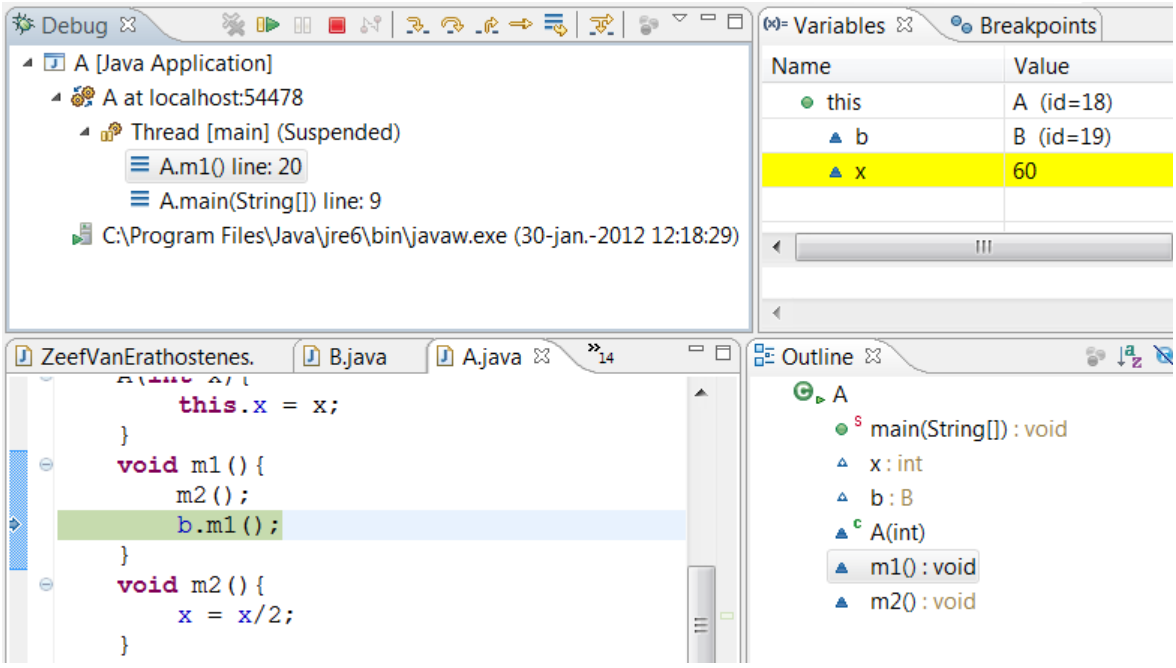
Ik laat het programma vervolgen met Step Into (of F5). Dan krijgen we het volgende:



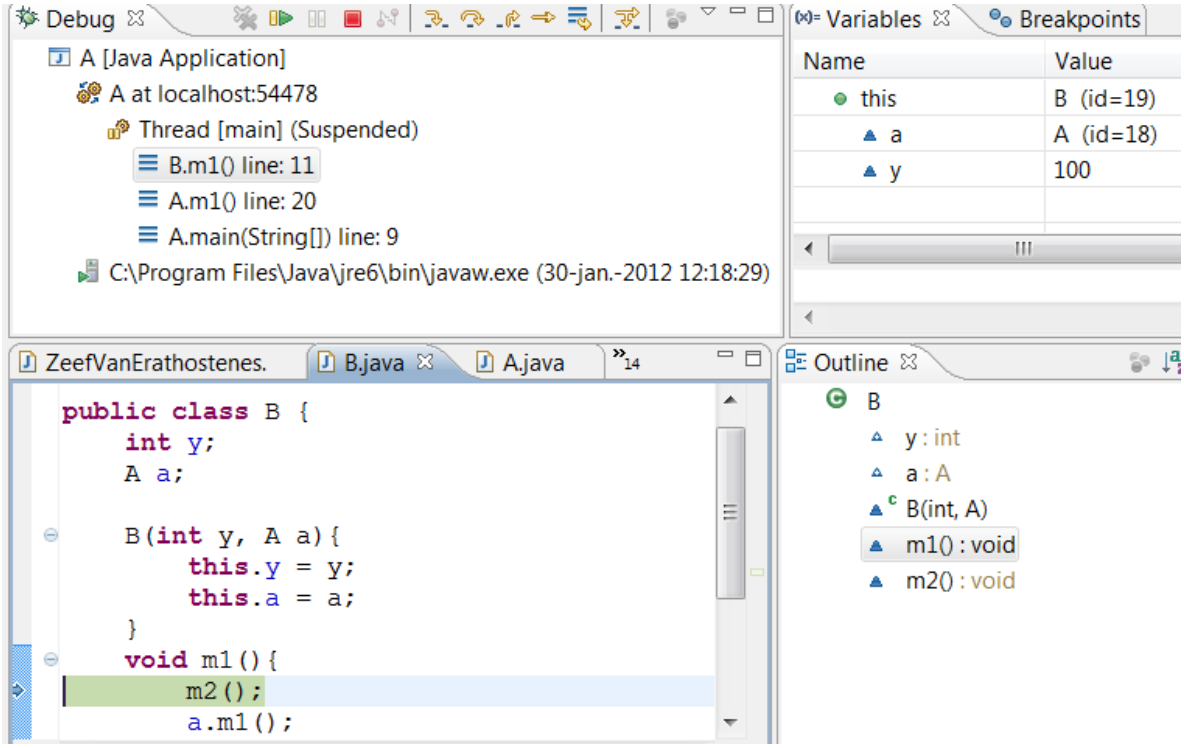
Linksboven zien we nu dat A.m1() de actieve methode is. Rechtsboven zien we de actieve variabelen: het object *a* met zijn twee attributen, object *b* en *x*. We vervolgen en springen in m2().



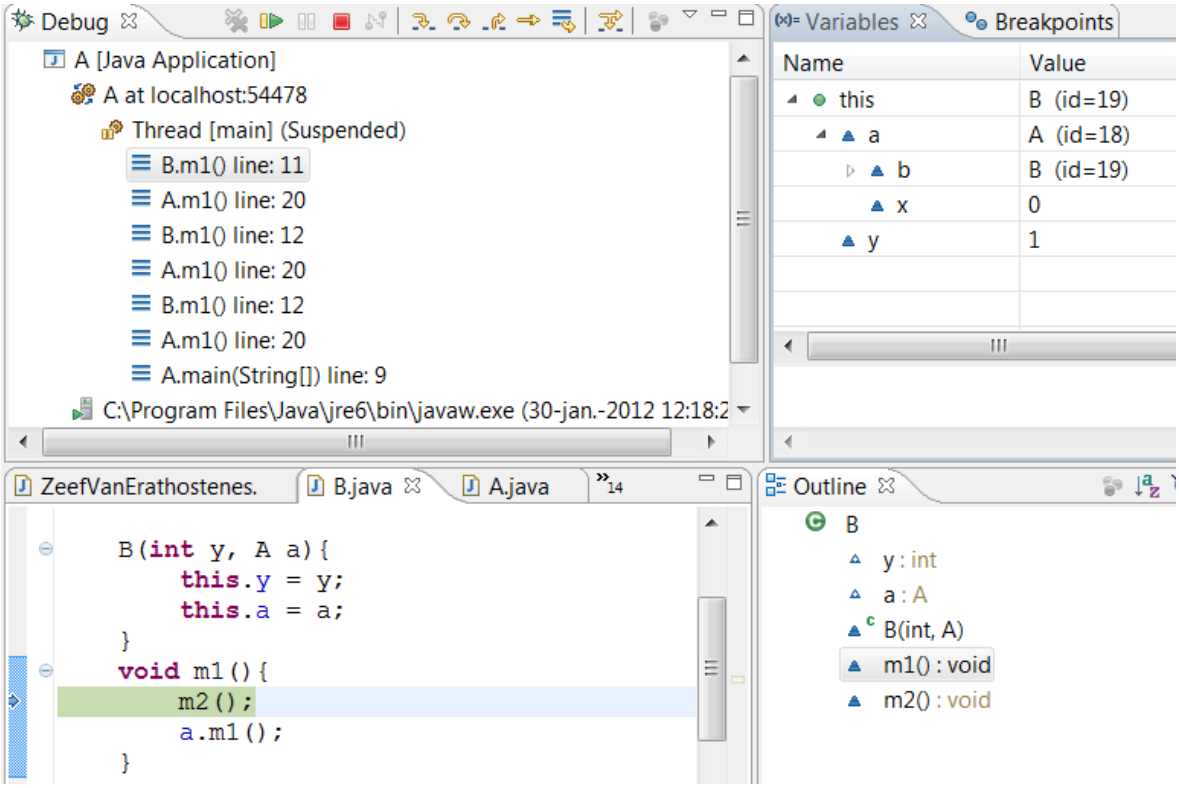
We blijven in object *a*. *m2()* is nu de uit te voeren methode, deze werd opgeroepen vanuit *m1()* die dan weer vanuit de *main()* werd opgeroepen. Dit zien we linksboven en dit heet de *call stack*: de keten van actieve methodes.
Na het uitvoeren van *m2()* is *x* 60 geworden (Eclipse toont dat *x* veranderd is):



Merk op dat er nu nog slechts twee actieve methoden zijn: de *main()* en *m1()*. Vervolgens wordt *m1()* van *b* uitgevoerd.



Nu zien we *B.m1()* op de *call stack*. Rechtsboven zien we dat we in object *b* zitten. Als we de computer het programma laten vervolgen, zullen van vanuit *m1()* van *b* methode *m1()* van *a* oproepen. En vanuit die methode weer *m1()* van *b*. De computer gaat op en neer tussen beide objecten en methoden. De *call stack* groeit aan:



Al switchen we steeds tussen dezelfde twee methodes, je mag niet vergeten dat we ooit nog terug moeten naar de methodes van waar we komen! Dit zal hier niet gebeuren, er is geen stopconditie. De methodes blijven elkaar oproepen. Tot in het oneindige... Maar als je bedenkt dat de *call stack* moet worden opgeslagen en dat elke methode-oproep een klein deel van he geheugen vraagt, dan kun je al raden dat dit maar kan duren tot het geheugen 'op' is.
Merk ook op, rechtsboven, dat *a.x* 0 is geworden en *b.y* 1. Je ziet dat object *a* bekend is binnen *b* en dat *b* op zijn beurt ook bekend is binnen *a*. Elk object krijgt binnen Java een nummer. Daaraan kun je zien dat het actieve object *b* dezelfde is als het attribuut *b* binnen *a* (id 19).

1.6.4. Abstracte klasses

Abstracte klasses bevinden zich tussen de concrete klassen (voorbeelden hierboven) en interfaces in: ze bevatten een mengeling van concrete dingen (attributen en concrete methodes) en abstracte dingen (*abstracte methodes*). Je schrijft *abstract* voor de klasse en voor elke abstracte methode waarbij deze methode geen implementatie heeft maar eindigt met een ';'.

Een klasse kan overerven van zowel een abstracte klasse (slechts 1) als van interfaces (meerdere). De klasse wordt pas concreet als *alle* abstracte methodes ingevuld zijn. Enkel van een concrete klasse mag je objecten maken, ook wel het *instantiëren* van de klasse genoemd.

Nog enkele voorbeelden om de verschillende mogelijkheden voor een constructor uit te leggen. Neem de volgende abstract klasse:

```
abstract class KlasseA{
	int x;
	abstract void f();
}
```

De klasse heeft geen expliciete constructor, wat impliceert dat de *default constructor* van toepassing is. Deze vraagt geen parameters en heeft een lege implementatie. De volgende code toont de default constructor expliciet. De code is dus compleet identiek aan de vorige code.

```
abstract class KlasseA{
	int x;
	KlasseA(){

	}
	abstract void f();
}
```

Bemerk dat *x* geïnitialiseerd zal worden met 0, ondanks dat dit niet expliciet in de constructor staat. Alle attributen krijgen per default waarde 0 (of *null* als het attribuut op een object slaat).
De volgende subklasse van *KlasseA* heeft een expliciete constructor, maar geen expliciete oproep van een superconstructor. Dit hoeft ook niet, aangezien het om de default constructor gaat. In feite staat er `super();` maar deze hoef je niet te schrijven.

```
abstract class KlasseB extends KlasseA{
	int y;
	KlasseB(int y){
		this.y=y;
	}
}
```

De volgende subklasse van *KlasseB* moet wel expliciet een superconstructor oproepen, omdat voor *KlasseB* de default constructor niet geldig is. Zodra je expliciet een constructor definieert, vervalt de default constructor namelijk.

```
class KlasseC extends KlasseB{
	int z;
	KlasseC(int y, int z){
		super(y);
		this.z=z;
	}
	void f(){
	}
}
```

Tot slot definiëren we een subklasse van *KlasseB* die een eigen default constructor definieert. Omdat *KlasseB* geen default constructor heeft, moet elke subklasse `super(...);` expliciet oproepen met een integer-parameter. Zelfs als *KlasseD* zelf geen parameter heeft in zijn constructor. *KlasseD* moet expliciet de default constructor definiëren waarin de `super(...);` komt te staan:

```
class KlasseD extends KlasseB{
	KlasseD(){
		super(10);
	}
	void f(){
	}
}
```

Ter samenvatting: je gebruikt abstracte klassen typisch wanneer je meerdere dochterklassen hebt die van een moederklasse overerven, maar het niet zinvol/mogelijk is om instanties te maken van een moederklasse.

Voorbeeld: een moederklasse “Animal” met dochterklassen “Dog” en “Cat”. De methode “make_sound()” in Animal is abstract omdat deze gespecifieerd moet worden door de dochterklassen. Verder is de expressie “new Animal()” niet toegelaten (en niet zinvol): elke instantie van een Animal moet een specifiek dier zijn. Zo weet je dat als je een List<Animal> aanmaakt, dat alle elementen zeker “Dog” of “Cat” zijn.

1.6.5. jar-files en runnable jars

Tijdens het programmeren zal Eclipse je Java-files omzetten in zogenaamde class-files. Deze code is nagekeken op fouten en omgezet (=compileren) naar een intermediaire taal (nog geen machinetaal). Persoon.java wordt gecompileerd naar Persoon.class. Het zijn deze class-files die bij het uitvoeren van je programma door de Java Virtual Machine omgezet worden naar machinetaal. Zie hoofdstuk 6 van Deel III voor meer uitleg hierover.
Class-files kan je bundelen tot jar-files. Een jar-file is niets anders dan een zip-file bestaande uit class-files. Je kan die zelf aanmaken met Winzip en dan de .zip in .jar. Of je gebruikt Eclipse.
Nog interessanter zijn de runnable jar-files. Hiermee maak je een programma dat op eender welke computer kan uitgevoerd worden mits die de Java Virtual Machine heeft geïnstalleerd. Een runnable jar voer je uit door te dubbelklikken, net als bij een normaal programma op je computer.
Aanmaken van runnable jars in Eclipse is heel eenvoudig:

- ***Menu ‘File’: Export => Java => Runnable JAR file***
- ***Je kiest vervolgens de klasse met de main() die uitgevoerd moet worden***
- ***Eclipse zorgt ervoor dat alle nodige class-files mee in de jar komen***
- Zorg dat beeldjes en audiofiles ook in de jar komen door ze bij je packages te zetten

Bekijk ook de documentatie op de parallel website voor verdere informatie.

1.7 Plan van aanpak voor klasse-identificatie

Hier een aantal stappen en tips om vanuit een probleemstelling de klassen, attributen en methodes van je applicatie te definiëren.

1. Identificeer de **objecten** van je applicatie. Dit zijn meestal de *onderwerpen* van de zinnen uit de opdracht of uit de beschrijving van de functionaliteit. Elk type object komt overeen met een **klasse**.
   - Als een bepaald object enkel door een naam gekenmerkt wordt, hoeft het niet een aparte klasse te worden, maar gebruik je o.a. een enumeratie (enum). Stel bijvoorbeeld dat je de vakken die een 'student' (= klasse) kan volgen enkel met hun naam benoemt (vb.: Fysica, Informatica, …). Deze vakken stel je dan voor met een String of met een enumeratie. Als je echter ook van elk vak de titularis, het aantal studiepunten, etc. gaat bijhouden, maak je wel een klasse 'Vak' aan (die dan de gepaste attributen bevat).
2. Identificeer vervolgens de **attributen** van elke klasse (de methoden komen later aan bod): deze vind je vaak ook rechtstreeks terug in de tekst, meestal als lijdend voorwerp (vb.: elke 'student' (klasse) heeft een 'rolnummer' (attribuut)).
   - Onderzoek de **relaties** tussen de objecten. Vb.: elke 'student' kan meerdere 'vakken' volgen: één attribuut van de student-klasse wordt aldus een lijst van objecten van de vak-klasse. Dit attribuut is dus een verwijzing naar 1 of meerdere andere objecten. Op deze manier zijn de objecten van de student-klasse gelinkt met de objecten van de vak-klasse.
3. Creëer eventueel een **hiërarchie** van klassen voor objecten die overlappen, waarbij de gemeenschappelijke attributen in de hoofdklasse gestoken worden (en worden doorgegeven via overerving).
   - Let op: voor aanverwante objecten worden enkel meerdere klassen voorzien indien deze objecten verschillen in attributen (een andere lijst van eigenschappen) of in methodes (een andere lijst van methodes of aan andere invulling van de methodes, vb.: de objecten worden anders getekend en hebben dus een andere 'paintComponent()' methode). Indien objecten enkel verschillen in de waarde voor een bepaalde eigenschap (vb.: de waarde van het attribuut 'kleur' of het attribuut 'grootte'), dan worden er in het algemeen geen verschillende klassen gemaakt maar simpelweg objecten van één en dezelfde klasse met verschillende attribuutwaardes.
4. Voor elke klasse definieer je de **constructor(s)**. Deze hebben als argumenten de eigenschappen die je minimaal wilt weten bij aanmaak van een object, alsook kennen ze de eigenschappen toe die je typisch kent bij het aanmaken van de objecten (vb.: het initialiseren van een array/arraylist of het toekennen van een default waarde). Voorzie meerdere constructors indien er zich meerdere gevallen kunnen voordoen.
5. Denk na over de operaties die bij elk object passen en voeg deze toe als een **methode** aan de klasse. Ofwel verandert de methode iets aan het object (wijziging van de waarde van een attribuut, toevoeging van een waarde aan een array-attribuut, etc.), ofwel of geeft de methode informatie terug over het object (vb.: de waarde van een berekening gebaseerd op de attributen van het object).
6. Handig is het aanmaken van de **toString()** methode. Zo wordt je object netjes geprint als je het meegeeft aan de System.out.println() functie. Dit is onder andere heel nuttig bij het debuggen van je programma.
7. In je **hoofdprogramma** (de main() functie) maak je objecten (of op zijn minst één object) aan en roep je hiervan de methodes op waardoor het programma doet wat het moet doen.
   - Old-school (cfr. je Python programma's) is om in de main() functie ook allerlei berekeningen uit te voeren op je objecten. Mooi OO programmeren doe je echter door al deze berekeningen in de methodes van de objecten zelf te steken. Je specifieert in de main() enkel *wat* er moet gebeuren door de methodes op te roepen. *Hoe* dit exact moet gebeuren staat in de methodes zelf gedefinieerd.

1.8 Oefeningen

De volgende oefeningen zullen we oplossen tijdens de hoorcolleges. Ze zullen je helpen te leren denken als een object-georiënteerde programmeur.

```
/**
   Wat is de output van het volgend programma?
 */
public class ObjectOefening1 {

	/** PROGRAMMA */
	public static void main(String[] args) {

		KlasseA a1 = new KlasseA(10);
		KlasseA a2 = new KlasseA(2, 8);

		System.out.println("a1 = "+a1);
		System.out.println("a2 = "+a2);

		KlasseB b1 = new KlasseB();
		System.out.println("b1 = "+b1);

		a1.f(3);
		a2.f(3);

		System.out.println("a1 = "+a1);
		System.out.println("a2 = "+a2);

		b1.g(a2);
		System.out.println("b1 = "+b1);
	}
}
class KlasseA {
	int x, y;
	KlasseA(int x){
		this.x=x;
		this.y=0;
	}
	KlasseA(int x, int y){
		this.x=x;
		this.y=y;
	}
	void f(int factor){
		x = factor * x;
		y = y / factor;
	}
	public String toString(){
		return "A [x="+x+";y="+y+"]";
	}
}
class KlasseB {
	int z = 5; // default waarde

	void g(KlasseA a){
		z = z * a.x + a.y;
	}
	public String toString(){
		return "B [z="+z+"]";
	}
}
```

```
package testoefeningen;
/**
 * Gegeven de volgende klassen: Trapezium, Rechthoek, Vierkant
 *
 1. Maak een object aan van elke klasse. Kies je eigen waarden voor de
parameters.
 2. Duidt aan welke constructors worden opgeroepen en welke
superconstructors.
 3. Geef voor elk object de waarde van de attributen.
 4. Bereken de oppervlakte van de trapezium en print deze af.
 5. Voeg een methode toe aan Rechthoek die de oppervlakte berekent.
 6. Maak van de rechthoek een subklasse van het trapezium. Dan zijn de
attributen 'breedte' en 'lengte' in feite niet meer nodig, die mag je dus
schrappen. De oppervlaktemethode ook.
 7. Vierkant erft alle attributen over, maar heeft er in feite slechts 1
nodig. Schrap de 'extends Rechthoek', voeg het nodige attribuut toe en de
methode voor oppervlaktebrekening.
 */
public class KlasseOefening1 {
	/** PROGRAMMA */
	public static void main(String[] args) {









	}
}
class Trapezium {
	double lengte1, lengte2, breedte;
	Trapezium(double lengte1, double lengte2, double breedte){
		this.lengte1 = lengte1;
		this.lengte2= lengte2;
		this.breedte = breedte;
	}
	double oppervlakte(){ return breedte * (lengte1 + lengte2)/2; }
}
class Rechthoek {
	double breedte, lengte;
	Rechthoek(double lengte, double breedte){

		this.breedte = breedte;
		this.lengte = lengte;
	}


}
class Vierkant extends Rechthoek {

	Vierkant(double zijde){
		super(zijde, zijde);

	}


}
```

```
1. public class KlasseOefening2 {
2.     /** PROGRAMMA */
3.     public static void main(String[] args) {
4.           Kl1 o1 = new Kl1(5);
5.           Kl2 o2 = new Kl2(7);
6.           Kl3 o3 = new Kl3();
7.           o1.f();
8.           o2.f();
9.           o3.f();
10.    }
11. }
12. // welke klassen hebben de default constructor? ........
13. class Kl1 {
14.    int a;
15.    Kl1(){ // op welke lijn(en) wordt dit opgeroepen? ........
16.          this.a=5;
17.    }
18.    Kl1(int a){ // op welke lijn(en) wordt dit opgeroepen? ........
19.          this.a=a;
20.    }
21.    void f(){  // op welke lijn(en) wordt dit opgeroepen? ........
22.    }
23. }
24. class Kl2 extends Kl1 {
25.    Kl2(int x){ // op welke lijn(en) wordt dit opgeroepen? ........
26.          super(x);
27.    }
28.    void f(){  // op welke lijn(en) wordt dit opgeroepen? ........
29.    }
30. }
31. class Kl3 extends Kl1 {
32.    int b;
33. }
```

InterfaceOefening

```
	// a. Welke concrete klasses zijn correct (zijn echt concreet)?
	// b. Hoe maak ik ze concreet?
	// c. Welke methoden zou ik mogen weglaten opdat het toch nog
concrete klassen blijven?

	interface I1 {
		void f();
		void g(int x);
	}
	interface I2 {
		int h();
		int k(int a);
	}

	class Cl1 implements I1{
		public void f(){
			...
		}
		public void g(){
			...
		}
		public int h(){
			...
		}
	}
	class Cl2 extends Cl1 implements I2 {
		public void f(){
			...
		}
		public void g(int x){
			...
		}
	}
```

Index