Java Programming - Deel 2

# Overerving

///DRAFTeen vorm van code-herbruik, uitbreiding van een bestaande klasse

2 klassen met dezelfde attributen => 1 superklasse met 2 subklassen waarbij de superklasse de gemeenschappelijke attributen bevat

Square (subklasse) extens Rectangle (superklasse) (klassenhiërarchie), subklasse erft over van/is een soort van superklasse.

een subklasse erft eigenschappen en methoden over van superklasse, kan zelf eigenschappen en methoden toevoegen en kan deze overriden( (één superklasse per klasse/één keer extends), die superklasse kan wel subklasse zijn van een andere superklasse enz

overriden = eigenschap met dezelfde naam en datatype, je krijgt bijgevolg de overschreven eigenschapswaarde

DON’T do this met eigenschappen, wel met methoden (bad practice)

//TOEGANGSNIVEAU DIAGRAM

alle public en protected methoden van de superklasse kunnen we gebruiken vanuit de subklasse, een subklasse is een uitbreiding van de superklasse (alle niet private eigenschappen en methoden van de superklasse zijn ook beschikbaar in de subklasse).

override (overschrijven/verbergen) is wanneer we in een subklasse een andere implementatie geven van een bestaande methode in de superklasse (dezelfde signatuur = naam + parameters + return type)

!elke methode uit een superklasse is standaard overschrijfbaar in een subklasse

public class Square extends Rectangle {  
 private int side;

//met deze annotatie in de subklasse om de compiler te laten nakijken of deze methode wel degelijk een methode uit zijn superklasse overschrijft  
 @Override  
 public int getArea(){  
 return side \* 2;  
 }  
}

om aan te duiden dat de methode uit de superklasse gebruikt moet worden

public void setSide(int side){  
 super.setWidth(side);  
 super.setHeight(side);  
}

Constructors

Indien geen geschreven => default

//default contructor van normale klasse, in dit geval superklasse  
public Rectangle(){  
}

//default constructor van subklasse  
public Square(){  
 super();  
}

Indien een subklasse gebruikt maakt van een constructor van de superklasse, moet deze met de juiste parameters opgeroepen worden in de subklasse

public Square(int side){  
 super(this, this); //altijd eerste statement  
 this.side = side;  
}

Final klassen

Kunnen geen subklassen hebben met als nut: beveiliging, ontwerp, snelheid

public final class GeenSubKlassen{}

Abstracte klassen

HIervan kunnen geen objecten gemaakt worden, dienen als superklasse van andere klassen4

public abstract class GeenObjecten{}

public abstract void abstracteMethode(); // geen implementatie!

deze bevatten (eventueel) abstracte methoden dewelke verplicht geïmplementeerd dienen te worden door de niet abstracte subklassen. Een subklasse roept altijd de constructor van de superklasse op omdat die een uitbreiding is van de superklasse!

OBJECT De klasse object is de superklasse van alle klassen, alle klassen erven over van de klasse Object. Geen attributen, enkel methoden. enkele vb.:

String toString()

boolean equals(Object o) --> vergelijkt 2 objecten (standaard zijn ze gelijk als ze naar hetzelfde object in het geheugen refereren. Als we deze overriden moeten deze samen met de methode hashcode vervangen.

int hashcode()

We bepalen zelf de eigenschappen wanneer 2 objecten gelijk zijn, als deze gelijk zijn moeten ze ook dezelfde hashcodeteruggeven:

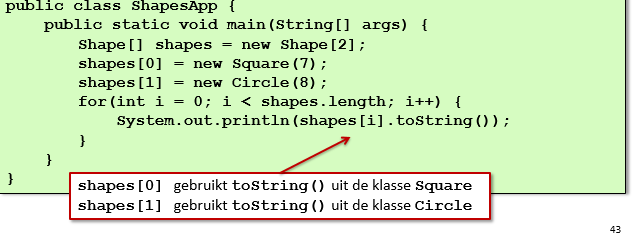
public int hashCode(){  
return x ^ y ^ height ^ width;}

Class getClass()

instanceof (kijkt na of een object een instantie is van de klasse, superklasse of interface

POLYMORFISME

Meerdere vormen/verschijningsvormen. In code betekent dit dat objecten van klasse die afgeleid is van een superklasse beschouwd en behandeld kunnen worden als objecten van die superklasse.

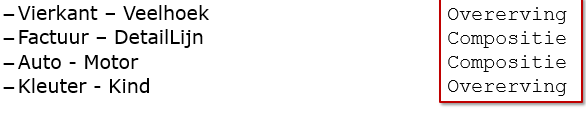


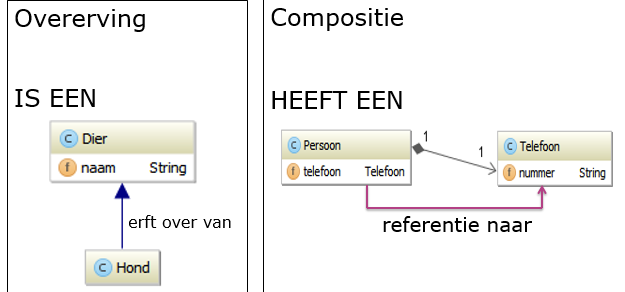
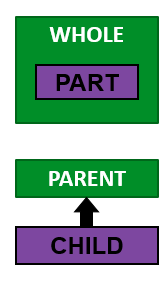
Late binding = uit welke klasse de methode gebruikt wordt, wordt at runtime bepaald. Polymorfisme geldt niet voor klasse (static) methoden of eigenschappen.

ASSOCIATIES tussen objecten

COMPOSITIE: een object bevat als onderdeel een ander object relatie: ... heeft een ...$

OVERERVING: een object is een afgeleide van een ander object relatie: ... is een ... (instanceof om te testen);





superklassen en subklassen: elke subklasse kan zelf ook een superklasse worden, een subklasse is specifieker dan zijn superklasse --> specialisatie

elke klasse kan slecht één directe superklasse hebben -> single inheritance

klassenhiërarchie begint in Java bij de klasse Object

 bladljfiazhiofjaifhoaizh

*Abstract class*

Single inheritance : een subclass kan enkel overerven van 1 abstract class

*Interface*

Multiple inheritance : een subclass kan meerder interfaces implementeren

# Enums, Objecten en Datum

## Variabelen

Een **instantievariabele** hoort bij een object en is uniek voor elk object, het heeft het object nodig om gebruikt te kunnen worden (objectnaam.variabelenaam).

public class Point {  
 private int x; //instantievariabele  
 ...  
 x = 5;  
 this.x = 7; //this keyword enkel nodig bij naamconflict  
 ...  
}

**** beginwaarde geven bij declaratie of in de constructor.

Een **klasse variabele**, aangegeven door static, hoort bij de klasse en is gemeenschappelijk voor elk object van die klasse (Klassenaam.variabelenaam).

public static final String *COLOR* = “red”;

Point point = new Point();  
Point.COLOR // ”red”  
~~point.COLOR~~ //  verwarrend en conceptueel fout!

vb.: om het aantal gecreëerde objecten van een klasse bij te houden:

private static int counter = 0;

public Point(int x, int y){  
 ...  
 counter++; //verhoogt de klasse variabele counter  
}

## Methoden

Een **instantiemethode** is een methode die gebruik kan maken van de eigenschappen van een concreet object.

Point point = new Point(); //point is een concreet object  
point.setPosition(9, 11); //setPosition is een instantiemethode

Een **klasse methode** is een methode die bij de klasse hoort en enkel gebruik kan maken van klassevariabelen! De **main methode** is een klasse methode en kan bijgevolg ook enkel gebruik maken van klasse variabelen.

import static graphics.Point.\*;

//klassenaam niet meer nodig gezien static import  
System.*out*.println(COLOR);   
System.*out*.println(getCount());

## Enum

Elke waarde heeft een volgnummer (ordial), de opsommingswaarden zijn altijd in hoofdletters. Een enum is eveneens een klasse en kan dus ook eigenschappen, constructors en methoden bevatten.

**** De constructor van een enum is private omdat er van buitenaf geen opsommingswaarden mogen toegevoegd worden. Let op met setters!

public enum Color {  
 BLACK(0x000000), WHITE(0xFFFFFF), RED(0xFF0000), GREEN(0x00FF00),  
 BLUE(0x0000FF), YELLOW(0XFFFF00);

//elke opsommingswaarde roept de constructor op: waarde(parameter)

private int rgb;

private Color(int rgb) { //constructor om rgb waarde mee te geven  
 this.rgb = rgb;  
 }

...  
}

public class Demo {  
 public static void main(String[] args){  
 Color voorgrond = Color.BLUE;  
 Color achtergrond = Color.YELLOW;  
 ...  
 }

private static void printColor(Color color) {  
 System.*out*.println(color.name()); //waarde van de color  
 System.*out*.println(color.ordinal());//index van de color  
 }  
}

|  |  |
| --- | --- |
| name() | naam van de waarde |
| ordinal() | index van de waarde |
| values() | alle mogelijke waarden van de Enum |
| toString() | naam van de waarde, idem als naam() |

## Datum

De belangrijkste methoden van de package (API) java.time

#### LocalDate

LocalDate huidigeDatum = LocalDate.now();

LocalDate datum = LocalDate.*of*(2017, 11, 27); //2017-11-27

Boolean isSchrikkelJaar = datum.isLeapYear();

* alleen datum, geen tijd
* geen publieke constructor

#### LocalTime

LocalTime nu = LocalTime.now(); //22:59:23.865

LocalTime tijd = LocalTime.*of*(22, 10, 25, 3592); //22:10:25.000003592

tijd.toSecondOfDay(); //79825

#### LocalDateTime

LocalDateTime vandaag = LocalDateTime.now(); //2017-11-25T16:48:34.686

LocalDateTime datum = LocalDateTime.of(2016, 11, 8, 23, 30, 12);  
//2016-11-8T23:30:12

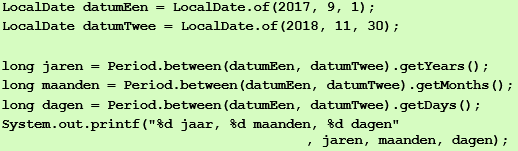
#### Datums vergelijken

|  |  |
| --- | --- |
| LocalDate.isBefore() | LocalDateTime.isBefore() |
| LocalDate.isAfter() | LocalDateTime.isAfter() |
|  |  |
| LocalTime.isBefore() |  |
| LocalTime.isAfter() |  |

#### Tijdsverschillen bepalen

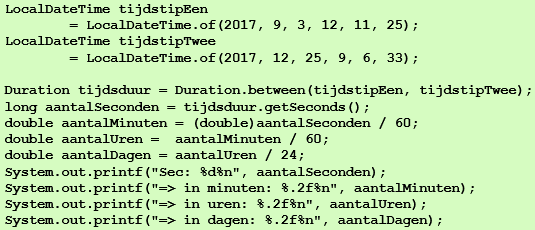
|  |  |
| --- | --- |
| Period.between() | verschil tussen LocalDates |
| Duration.between() | verschil tussen twee tijden (LocalTime & LocalDateTime) |
| ChronoUnit | verschil tussen twee tijdsintervallen |

#### Period.between



1 jaar, 2 maanden en 29 dagen

#### Duration.between



Sec: 9752108  
=> in minuten: 162535,13  
=> in uren: 2708,92  
=> in dagen: 112,87

#### ChronoUnit

ChronoUnit.{Tijdsindeling}.between({startTijd},{eindTijd});

long jaren = ChronoUnit.YEARS.between(tijdEen, tijdTwee);

#### DateTimeFormatter

DateTimeFormatter d1 = DateFormatter.ofLocalizedDate(FormatStyle.FULL);  
//vrijdag 3 november 2017

DateTimeFormatter d2 = DateFormatter.ofLocalizedDate(FormatStyle.MEDIUM);  
//3-nov-2017

DateTimeFormatter d3 = DateFormatter.ofLocalizedTime(FormatStyle.MEDIUM);  
//13:23:29

DateTimeFormatter d = DateFormatter.ofLocalizedDateTime(FormatStyle.SHORT);  
//3/11/17 13:15

<https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>

# Interfaces

Een **interface** is een verzameling van methoden en/of constanten, maar deze kan ook leeg zijn. Deze kan overerven van meer dan één interface, in dat geval bestaat de interface uit methoden en/of constanten van de interface zelf EN die van de super-interfaces.

* om klassen te verplichten bepaalde methoden te implementeren
* een vorm van meervoudige overerving mogelijk te maken
* polymorfisme mogelijk maken zonder overerving, totaal verschillende objecten een identiek gedrag geven

public interface Naam extends SuperEen, SuperTwee, ...  
//enkel public of (package-private), nooit static (wel static methodes)!

vb.: de methode herschaal moet uitgewerkt worden in de klassen die de interface implementeren.

public interface Herschaalbaar {  
 void herschaal(int percentage); //automatisch public abstract  
  
 int *DUBBEL* = 200; //automatisch public static final  
 int *HALF* = 50; //vanuit andere klasse: Herschaalbaar.HALF  
}

### Implementatie



public class N

**** Een klasse kan meer dan één interface implementeren, maar slechts 1 superklasse hebben. Alle constanten van de interface(s) worden door de klasse als static waarde overgeërfd.

### Data type

Een interface kan ook een referentietype zijn waardoor polymorfisme mogelijk wordt. Elk object dat een bepaalde interface implementeert is van dit type.

Herschaalbaar herschaalbaar = new Vierkant(2.0);  
//Herschaalbaar is het referentietype -> referentie naar Vierkant object

**** In dit geval zijn alleen de methoden van het referentietype beschikbaar!

## Inner Classes

Een geneste klasse binnen een andere klasse, nuttig wanneer je objecten hebt die maar voor één object nuttig zijn.

* gewone geneste klassen
* lokale geneste klassen
* anonieme geneste klassen

### Gewone geneste klasse

|  |  |
| --- | --- |
| this | refereert naar de geneste klasse zelf |
| {OuterClass}.this | refereert naar het gerelateerde OuterClass object |

public class OuterClass {  
 ...  
 class InnerClass {  
 ...  
 }  
 ...  
 InnerClass inner = new InnerClass(); //binnen de nestende klasse  
}

OuterClass outer = new OuterClass(); //buiten de nestende klasse  
OuterClass.InnerClass innerOut = outer.new InnerClass();

**** Je kan enkel een instantie van een inner class maken uit een instantie van een outer class buiten de nestende klasse.

### Anonieme geneste klasse

Een naamloze lokale geneste klasse. Een subklasse of een klasse die een bepaalde interface implementeert, deze anonieme innerklasse moet de methoden van de superklasse overrriden of de abstracte methoden van interface implementeren.

**** Kan extra methoden bevatten, maar deze zijn ontoegankelijk. Altijd een default constructor.

public class MijnAndroid {  
 Android android = new Android() {  
 //anonieme inner class Android  
 @Override  
 public void print(){  
 System.*out*.println(“Hello van subklasse van Android”);  
 }  
 };  
}

Toegankelijk via:

MijnAndroid mijnAndroid = new MijnAndroid();  
mijnAndroid.android.print(); //Hello van subklasse van Android

### Static geneste klasse

Net zoals static variabelen of methoden horen ze rechtstreeks toe aan de nestende klassen.

public class OuterClass {  
 public static class InnerClass {  
 ...  
 }  
}

OuterClass.InnerClass nested = new OuterClass.InnerClass();

Een enum kan evenzeer genest worden, dit is equivalent als een static nested class.

public class Kleuren {  
 public enum Kleur {  
 *ROOD*, *BLAUW*, *GROEN*  
 }  
}

...  
  
Kleuren.Kleur[] lijst = Kleuren.Kleur.values();  
for (Kleuren.Kleur kleur : lijst) {  
 System.*out*.println(kleur);  
}

# Exception handling

Een **exception** is een uitzonderlijke gebeurtenis die zich voordoet tijdens de uitvoering van een programma en die het normale verloop verstoort. Voordelen van exception handling:

* een exception is zelf een **object** met nuttige informatie
* de code dewelke de fouthandeling behandelt is **gegroepeerd** **en** **gescheiden**
* exception **borrelt** **omhoog** doorheen de call-stack
* exceptions zijn **hiërarchisch** opgebouwd waardoor je de best passende exception kan kiezen

#### Stacktrace

Exception in thread "main" java.util.InputMismatchException  
 at java.util.Scanner.throwFor(*Scanner.java:864*)  
 at java.util.Scanner.next(*Scanner.java:1485*)  
 at java.util.Scanner.nextInt(*Scanner.java:2117*)  
 java.util.Scanner.nextInt(*Scanner.java:2076*)  
 at DemoException.main(*DemoException.java:9*) //zelfgeschreven klasse

Process finished with exit code 1

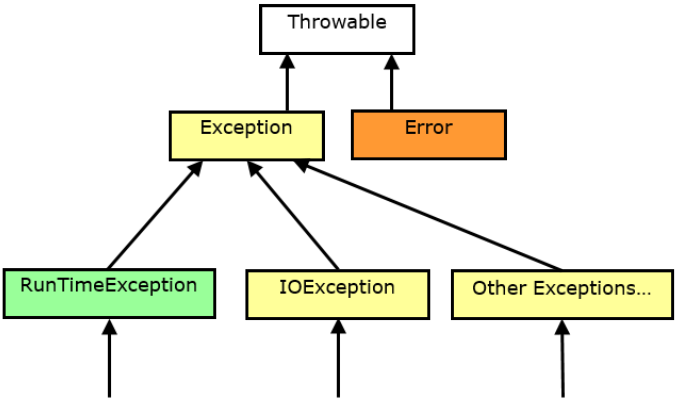
Een stacktrace dien je van onderaan te lezen, de exacte methode waarin de Exception zich heeft voorgedaan staat eerst vermeld.

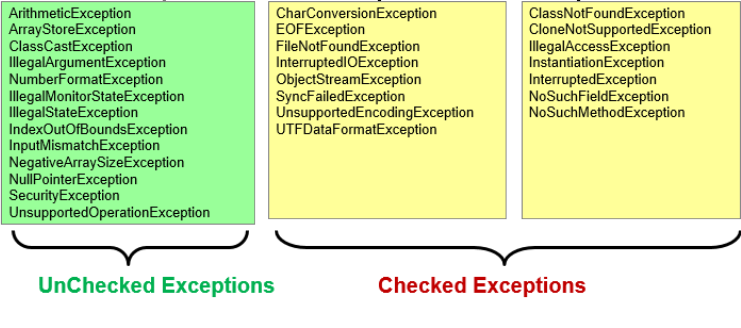
**** JavaDoc via ***<SHIFT> + F1*** of informatie over de methode via ***<CTRL> + Q***.

## Exceptions klassenhiërarchie

**Unchecked** **exceptions** zijn van het type **RunTimeException** en mogen opgevangen worden, deze worden niet door de compiler gecontroleerd. Deze mogen eigenlijk niet voorkomen (fouten).

**Checked** **exceptions** zijn verplicht op opgevangen te worden, ofwel door de methode zelf of door een oproepende methode uit de call-stack. De code kan niet gecompileerd worden indien deze niet opgevangen worden.





## Tekstbestanden lezen

De klasse Path stelt een file of directory in het onderliggend filesysteem voor. Path heeft geen publieke constructor, er kan dus geen object gecreëerd worden met new.

Path myFile = Paths.*get*(“**resources**\\**myfile.txt**”);

De klasse Files bevat static methodes om met een Path object iets te doen.

Path myFile = Paths.*get*(“**myfile.txt**”);  
if (!Files.*exists*(myFile)){  
 try {  
 Files.createFile(myFile);  
 } catch (IOException ioe) {  
 //afhandeling  
 }   
}

|  |  |
| --- | --- |
|  | De eerste regel van een tekstbestand inlezen. |
|  | Een tekstbestand blijven uitlezen zolang er regels tekst in voorkomen.  **** Indien men Scanner gebruikt in combinatie met bestanden is het belangrijk de Scanner af te sluiten na gebruik. Anders kan dit een negatieve invloed hebben op de performantie. |

#### Efficiënter via “try with resources”

|  |  |
| --- | --- |
|  | Java zal automatisch zorgen dat de Scanner afgesloten wordt (geen close methode en geen finally block meer nodig). |

# Collections

## Wrapper klassen

Alle **wrapper klassen** voor getallen bevatten de constanten MIN\_VALUE en MAX\_VALUE. Verder heeft elke wrapper klasse een aantal valueOf methoden die telkens een object van de overeenstemmende klasse teruggeven.

|  |  |
| --- | --- |
| CONSTANTEN |  |
| Integer.MIN\_VALUE | -214783648 |
| Integer.MAX\_VALUE | 214783648 |
| Double.NaN | Not a Number (geen getal) |
| Double.NEGATIVE\_INFINITY | - oneindig |
| STATIC METHODS |  |
| Integer.parseInt() | String naar int |
| Integer.parseDouble() | String naar double |
| Double.isFinite() | is het een eindige waarde? |

Het omzetten van een primitief type naar object kan op 2 manieren:

#### Auto(un)boxing

private static void autoBoxing(){  
 Integer intObject = 3; //3 is primitief, wordt omgezet naar wrapper  
 int intPrimitive = new Integer(3); //object naar primitief  
}

#### Casting

private static void autoBoxing(){  
 int cijfer = 5;  
 System.*out*.println(((Integer)cijfer).hashCode());  
}

**** Boxing kost tijd en dus snelheid! Wrapper klasses zijn overbodig als je geen specifieke methodes oproept.

 Autoboxing en conversie tegelijk lukt niet!

## Collections framework

Is een framework voor het werken met verzamelingen van objecten dat bestaat uit interfaces en implementaties. Arrays zijn statisch en hebben vaste lengte, Collections niet.

#### Collection

|  |
| --- |
| * superinterface voor andere interfaces (List, Set, ...) |
| * is een verzameling van willekeurige objecten van eenzelfde generiek type |
| * beschrijft op abstracte wijze het gedrag van een collection |

#### List

Geordende, sequentiële verzameling dewelke dubbels mag bevatten.

#### Set

Niet-geordende verzameling van unieke objecten (geen dubbels!).

### ArrayList

De klasse ArrayList is een concrete implementatie van de List interface, is een dynamisch aanpasbare reeks (ArrayList beheert de achterliggende array).

|  |  |
| --- | --- |
| **+** | ieder element heeft een bepaalde positie, benaderbaar via index |
| **-** | minder geschikt voor veelvoudig toevoegen en verwijderen van elementen |

List<String> myList = new ArrayList<>();

 Geen primitieve types toegelaten in een ArrayList.

##### Loopen over een List

for(int i = 0; i < myList.size(); i++){  
 System.*out*.println(myList.get(i));  
} //shortcut: itli

for(String elementOfList : myList){  
 System.*out*.println(elementOfList);  
} //shortcut: iter

for(Iterator<String> it = myList.iterator(); it.hasNext();){  
 System.*out*.println(it.next());  
} //shortcut: itco

##### Array omzetten naar ArrayList

String[] myArray = {“Alfa”, “Bravo”, “Charlie”, “Delta”};

List<String> myList = new ArrayList<>();  
for (String name : myArray){  
 myList.add(name);  
}

List<String> myList = Arrays.*asList*(myArray); **** READ ONLY !!!

##### ArrayList omzetten naar Array

Dit kan met de static methode toArray van List.

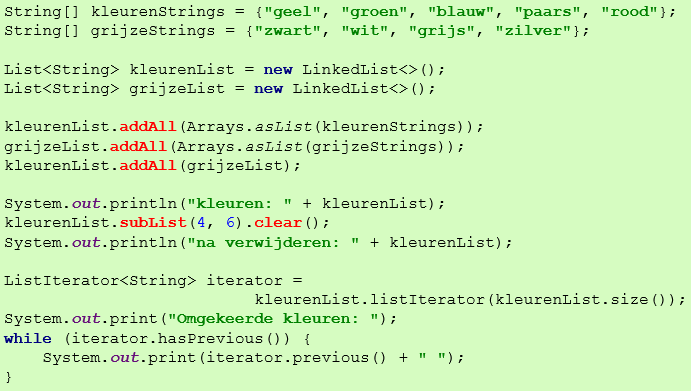
String[] myArray = myList.toArray(new String[0]);  
// parameter is lege Array omdat caste niet lukt!

### LinkedList

De klasse ArrayList is een concrete implementatie van de List interface, is een dubbel gelinkte lijst. Elk element heeft een verwijziging naar zijn voorganger/opvolger.



|  |  |
| --- | --- |
| **+** | zeer geschikt voor toevoegen/invoegen/verwijderen |
| **-** | minder geschikt voor willekeurige toegang |





|  |  |
| --- | --- |
| ArrayList VS LinkedList | |
| snel bij positioneren (get en set) | traag bij get en set |
| traag bij add (int index, Object o) en remove | snel bij alle add en remove methoden |

## Collections klasse

Is niet hetzelfde als de interface Collection! Collections is een toolbox met allerlei static methoden. Collections.sort sorteert de elementen van een List, de volgorde wordt bepaald door de compareTo methode (de enige methode in de Comparable interface).

### Sorteren

|  |  |
| --- | --- |
| List<String> woorden | **** Collections.sort(woorden); |
| List<Integer> getallen | **** Collections.sort(getallen); |
| List<Double> temperaturen | **** Collections.sort(temperaturen); |
|  |  |
| List<Student> student |  Student niet Comparable! |

We dienen de klasse Student eerst zelf Comparable te maken.

public class Student implements Comparable<Student>{  
 private String naam;  
 private String nummer;  
  
 public Student(String naam, String nummer){  
 this.naam = naam;  
 this.nummer = nummer;  
 }  
 //... sorteren op de naam van de student  
 public int compareTo(Student andereStudent){  
 return this.naam.compareTo(andereStudent.naam);  
 }

public static void main(String[] args){  
 List<Student> studenten = new ArrayList<>();  
 studenten.add(new student(“Kevin”, 20));  
 //...  
 Collections.*sort*(studenten);  
 //...  
}

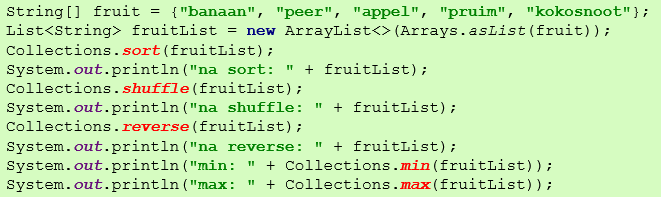
Indien we op meerder criteria dienen te sorteren moeten we gebruik maken van een externe Comparator klasse met daarin een compare methode.

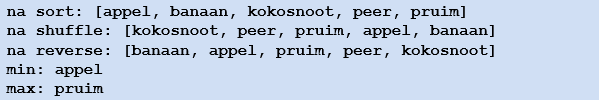
public class NummerComparator implements Comparator<Student>{  
 @Override  
 public int compare(Student een, Student twee){  
 return een.getNummer() – twee.getNummer();  
 }  
}

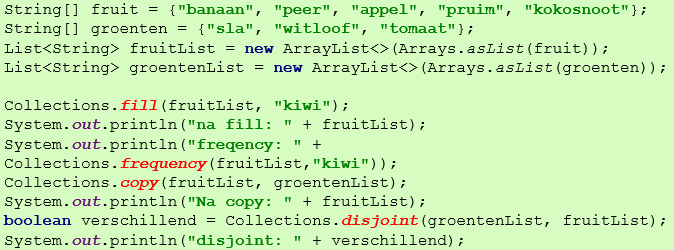
public static void main(String[] args){  
 List<Student> klasgroep = new ArrayList<>();  
 klasgroep.add(new Student(“Kevin”, 20);  
 //...  
 Collections.sort(klasgroep); //compareTo methode van Student  
 Collections.sort(klasgroep, new NummerComparator());  
 //...  
}

**** sinds JDK8 -> klasgroep.sort(new NummerComparator());

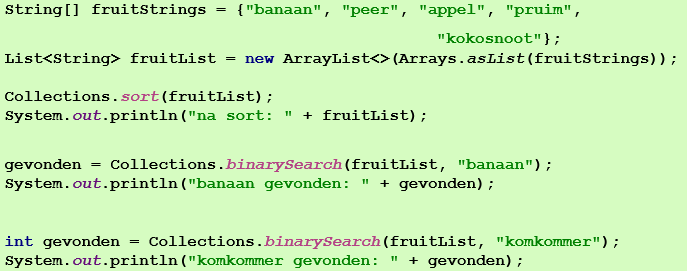
### Methodes van Collections







|  |  |
| --- | --- |
|  | disjoint -> returns true als de 2 opgegeven collections geen gemeenschappelijke elementen hebben. |



|  |  |
| --- | --- |
|  | **** eerst de lijst sorteren alvorens binarySearch toe te passen! |

**** Indien gevonden: index plaats, anders –insertionpoint -1