Java Advanced Samenvatting

Generics

Generic Classes

- Dit zijn klassen, methodes, interfaces of contructors die een datatype gebruiken dat nog niet bepaald is.
- Tijdens het instantiëren van de klasse wordt er door de compiler pas concreet ingevuld welke datatype er gebruikt wordt.
- Dit voorkomt dat we steeds **Object** zullen moeten gebruiken en zullen moeten gaan Casten om fouten te voorkomen
- We vervangen dus eigenlijk het datatype **Object** door een Generic Datatype, bv **E**.
- We kunnen eender welke naam kiezen, maar in de praktijk worden vaak deze letters gebruikt:
    - T, U, V, etc – type
    - E – element
    - K – Key
    - N – Number
    - R – Result
    - etc.

Vroeger moesten we dit doen:

Duo
-first: Object -second: Object
+Duo(first : Object, second: Object) +getFirst() : Object +setFirst(first : Object) +getSecond() : Object +setSecond(second : Object)

- Met Generics kunnen we het gebruik van Object als overkoepelend object vervangen door een meer specifiek datatype dat we zelf gaan definieren.
- We weten nog niet precies wat er exact in zal zitten, maar we hebben wel een idee van de mogelijke datatypes die er in kunnen zitten.
- Het juiste datatype wordt zoals eerder gezegd bij het instantieren van de klasse effectief ingevuld door de compiler.

Vroeger moesten we dit doen:	***Nu kunnen we dit doen:***
public class Houder { *private Integer inhoud;* *public Integer geef() {* *return inhoud;* *}* *}* *public class Houder {* *private String inhoud;* *public String geef() {* *return inhoud;* *}* *}*	*public class Houder<T> {* *private T inhoud;* *public T geef() {* *return inhoud;* *}* *}* *Of er nu een Integer of een String wordt meegegeven aan de Constructor maakt niets uit. <T> is een Generic datatype dat eender welk Object in zich kan hebben.*

Voordien moesten in onze main methode dus ook telkens casten met de oude manier:

```
Houder houder1 = new Houder(1);
Houder houder2 = new Houder("Test");
Houder houder3 = new Houder(new Persoon("Jan"));
…
Integer inhoud1 = (Integer) houder1.geef();
String  inhoud2 = (String)  houder2.geef();
Persoon inhoud3 = (Persoon) houder3.geef();
```

Nu moeten we dit niet meer doen want het datatype is Generic. De .geef() methode geeft automatisch een Integer, String of Persoon object terug van het moment de Houder klasse geïnstantieerd wordt.

```
Houder<Integer> houder1 = new Houder<>(1);
Houder<String>  houder2 = new Houder<>("Test");
Houder<Student> houder3 = new Houder<>(new Student("Jan"));
…
Integer inhoud1 = houder1.geef();
String  inhoud2 = houder2.geef();
Persoon inhoud3 = houder3.geef();
```

Nog een voorbeeld:

```
public class CoinTosser {
private static Random rand = new Random();

public static <T> T toss(T item1, T item2) {
        return rand.nextBoolean() ? item1 : item2;
}
}
Student jan = …, piet = …
Persoon vrijwilliger = CoinTosser.toss(jan, piet);
```

Deze Generic methode kiest at random 1 van de twee T elementen die meegegeven wordt en zal deze returnen. Aangezien we bv 2 Integers, 2 Persoon objecten of 2 Strings kunnen meegeven, werkt deze methode voor elk datatype. T stelt dus ook eender welk type voor, welk type? Dat wordt op het moment dat de methode gebruikt wordt pas ingevuld door de compiler.

Het Generic Datatype beperken

Momenteel kunnen we eender welk Object in ons Generic datatype stoppen, maar in de realiteit willen we dit beperken tot enkele mogelijke datatypes. Als we bv willen dat er enkel een Integer, Long, Float, Double, Short of Byte wordt meegegeven, kunnen we bv definieren dat er enkel datatypes mogen meegegeven worden afgeleid worden van de abstracte Number klasse.

Ipv dat we bv <E> plaatsen in onze klasse bovenaan, kunnen we dan <E extends Number> plaatsen. Het type moet dus van de klasse **Number** zijn of het moet een **subklasse** van Number zijn. Op deze manier beperken we aantal datatypes dat **E** kan bevatten.

We doen om bv het volgende probleem te voorkomen:

```
NumberDuo<Number> nd = new NumberDuo<Integer>(7,5);
nd.setFirst(new Double(5.7);
```

NumberDuo is een klasse met een Generic datatype E, in de constructor kan je 2 objecten meegeven, samen vormen deze een duo. Er kunnen op het NumberDuo object vervolgens methodes uitgevoerd, zoals sum, setFirst, setSecond, etc.

Aangezien er eender welk Number object kan zitten, kan bv het eerste object vervangen worden via de methode setFirst() door een Double object. Er zit nu dan een Duo van Number objecten in, de eerste is een Double, de tweede een Integer. Dit geeft een error, want tijdens het aanmaken van het NumberDuo object hebben **Integer** gedefinieerd als het datatype.

Indien we willen voorkomen dat de objecten dus vervangen worden, kunnen de **wildcard** gebruiken.

```
NumberDuo<?> nd = new NumberDuo<Integer>(7,5);
```

Er zitten nog steeds 2 Integers in, maar omdat we **?** gebruikt hebben kunnen we de methodes setFirst en setSecond niet meer gebruiken aangezien het echte type ongedefinieerd is. De objecten eruit halen via de getFirst en getSecond methodes lukt wel, de zullen een **Number** object teruggeven.

Subklassen van Generics

We kunnen Generic klassen ook gebruiken om van te extenden:

```
public class IntegerDuo extends Duo<Integer> {
        public IntegerDuo(Integer first, Integer second){
                super(first, second);
        }
}
```

De nieuwe klasse is dus een subklasse van Duo dat enkel Integers verwacht.
We moeten dus in de constructor deze 2 integers doorgeven aan de constructor van Duo zelf.
Deze nieuwe klasse is wel geen Generic klasse zelf natuurlijk, ook al wordt deze extend van Duo.
Indien we in de subklasse zelf een nieuwe Generic datatype definieren, dan wordt deze wél een Generic klasse.

Dit werkt identiek aan klassen zelf, we kunnen een E datatype definieren bij het aanmaken van de Interface, dit kan bij gevolg gebruikt worden.

```
public interface Pair<E> {
        public E getLeft();
        public E getRight();
}
```

Elke klasse die deze interface implementeert zal bij gevolg deze 2 methodes moeten aanbieden, de methodes zelf geven een **E** object terug. Het E object kan eender welk Object zijn.

```
public class ShoePair implements pair<Shoe> {
        private Shoe left;
        private Shoe right;

        public ShoePair(Shoe left, Shoe right) {
                leftShoe = left;
                rightShoe = right;
        }

        public Shoe getLeft(){
                return leftShoe;
        }

        public Shoe getRight(){
                return rightShoe;
        }
}
```

De klasse ShoePair implemteert de interface Pair en geeft aan dat het E object van het datatype Shoe zal zijn. Bij gevolg moet deze klasse beide methodes uit de Pair Interface aanbieden. Deze geven allebei een Shoe object terug zoals aangegeven staat tijdens **implements**.

We kunnen ook zelf een Generic klasse schrijven die helemaal geen type meegeeft:

```
public class GeneralPair<E> implements Pair<E>{ …
```

De klasse geeft dus niet aan welk type er gebruikt wordt, pas wanneer de klasse gebruikt wordt (bv in de main) zal het juiste datatype ingevuld worden door de compiler:

```
GeneralPair<Integer> numberPair = new GeneralPair<>(2,5);
```

Als we nu Objecten willen opslaan die met elkaar vergeleken kunnen worden, dan moeten deze Objecten de Interface **Comparable** implementeren.

```
public class ComparableDuo<E Extends Comparable<E>>{
…
```

Het type dat meegegeven wordt moet dus **Comparable** implementeren, maar we weten zelf niet welke dat juist zijn.

Door dat we dit gedaan hebben kunnen we bij gevolg de Objecten vergelijken met de compareTo() methode. We zijn er immers zeker van dat deze Objecten deze methode zullen aanbieden!

```
public E getHighest(){
        return (first.compareTo(second)<0)?second:first;
}
```

Opmerking: Deze method geeft first OF second terug afhankelijk van de compareTo. Als deze kleiner is dan 0 geeft hij second terug, anders First.

We kunnen ook meerdere vereisten instellen tegelijkertijd d.m.v. het **&** teken:

```
public class NumberDuo<E extends Number & Comparable<E>> { …
```

In dit voorbeeld bepalen we bv dat Number zelf niet meer meegegeven worden, dit is een abstracte klasse en implementeert Comparable niet. Alle subklassen van Number voldoen wel aan de vereisten en kunnen wel worden meegegeven.

Generic Methoden

In de CoinTosser methodes van voordien hebben we eigenlijk al een Generic methode gemaakt:

```
public class CoinTosser {
private static Random rand = new Random();

public static <T> T toss(T item1, T item2) {
        return rand.nextBoolean() ? item1 : item2;
}
}
Student jan = …, piet = …
Persoon vrijwilliger = CoinTosser.toss(jan, piet);
```

De methode geeft dus at random een T object terug, welk object dat juist zal zijn wordt bepaalt door de argumenten die meegegeven worden aan de methode zelf. Als er dus twee Integer objecten worden meegegeven weet de compiler dat het return type ook een Integer moet zijn.

Als we tijdens het aanroepen van de methode zelf willen bepalen welk datatype er zal gereturned worden door de compiler moeten we dit expliciet vermelden:

```
Student jan = …, piet = …
Persoon vrijwilliger = CoinTosser.<Persoon>toss(jan, piet);
```

Er wordt een Student en een Persoon object meegegeven, maar we zeggen tegen de method CoinTosser dat er een Persoon object moet gereturned worden. (Student is een subklasse van Persoon, daarom werkt dit).

Als we op voorhand willen bepalen dat er eender welk type mag meegegeven worden gebruiken we weer **wildcards**.

```
public static void printDuo(Duo<?> d) {
        System.out.println(d.getFirst() + " "+ e.getSecond());
}
```

Wat we zullen meegeven is dus niet gedefinieerd, daarom zal de compiler dit als objecten van de Object klasse bekijken, dit is het enige waar de compiler zeker van is.

In dit voorbeeld werkt het afdrukken van de objecten voor eender welk type, maar als we nu bv de som willen maken ipv ze af te drukken, dan kunnen we bepalen dat er enkel subklassen van de Number klasse mogen meegegeven worden:

```
public static void PrintSum(Duo<? extends Number> d){
        number n1 = d.getFirst();
        number n2 = d.getSecond();
        System.out.println(n1 + " + " + n2 + " + " + n1.doubleValue()+n2.DoubleValue()));
}

Duo<Integer> id = new Duo<>(7,5);
DuoUtility.printSum(id);
```

Opmerking: **?** begrensd het type wat we kunnen meegeven, enkel Number objecten of subklassen er van. Het type dat teruggegeven wordt is dus ook telkens een Number Object, dat is het enige waar de compiler zeker van is.

De vorige methode begrensd naar boven toe, de subklassen van Number en Number zelf!
Als we nu naar onder willen begrensen, bv de Number klasse en alle superklassen van Number, dan kunnen we het keyword **super** gebruiken. Dit betekent dus ook dat er GEEN subklassen van Number gebruikt kunnen worden:

```
public static void method(Duo<? super Number> d) {
        …
}
```

Als we in een methode parameters willen meegeven van het type Duo en we willen zeker zijn beide Duo objecten van hetzelfde type zijn (bv 2 Duo's van Integers):

```
public static <T> void swapFirst(Duo<T> d1, Duo<T> d2){
        T temp = d1.getFirst();
        d1.setFirst(d2.getFirst());
        d2.setFirst(temp);
}
```

We verwisselen het eerste element uit beide Duo's, maar hiervoor moeten beiden Duo bv Integers in zich hebben, anders geeft dit een error. Door T te gebruiken weten we op voorhand dat d1 en d2 allebei hetzelfde type Duo zijn. Als we meerdere types willen toestaan:

```
public static <T1, T2> void PrintMixed(Duo<T1> d1, Duo<T2> d2){
        …
}
```

Indien we een array willen aanmaken van Duo objecten, mogen we niet expliciet definieren dat deze bv Integers in zich hebben.

```
Duo<Integer>[] duos;          //OK
duos = new Duo<Integer>[5];   //Compiler fout

Duo<?>[] duos;                //OK
duos = new Duo<?>[5]          //OK
```

Geneste en Anonieme Klassen

Geneste klassen (Inner Classes)

```
public class OuterClass {
        public class InnerClass {
                public void aMethod(){
                        …
                }
        }
}
```

We kunnen de InnerClass en de methode op 2 manieren aanroepen in onze code:

1. Door een object aan te maken van de InnerClass In de OuterClass zelf.

```
public class OuterClass {
        public class InnerClass {
                public void aMethod(){
                        …
                }
        }

        public void doSomething(){
                InnerClass inner = new InnerClass();
                inner.aMethod();
        }
}
```

2. Door in een andere klasse Eerst een OuterClass object aan te maken en vervolgens een InnerClass Object.

```
OuterClass outer = new OuterClass();
OuterClass.InnerClass inner = outer.new InnerClass();
```

Opmerkingen:

- InnerClass heeft toegang ALLE members van OuterClass, ook de privates.
- Als je de huidige instantie van de InnerClass wil aanspreken gebruik je **this**
- Als je de huidige instantie van de OuterClass wil aanspreken gebruik je **OuterClass.this**
- InnerClass kan ook de volgende componenten gebruiken:
    - final
    - abstract
    - public, private, protected
    - static

Lokale Geneste Klassen (Local Inner Classes)

- Je kan ook klassen maken binnen een method, dit noemen ze Local Inner Classes (denk aan Local variables in een methode).
- Je kan deze klasse natuurlijk enkel binnne de scope van je methode gebruiken.
- Enkel de **final** variabelen in de methode kunnen door de Local Inner Class gebruikt worden!
    - Dit is zo om te voorkomen dat we variabelen aanspreken die eventueel niet meer bestaan omdat de methode afgelopen is.

```
public class OuterClass{
    public void methodA() {
            final int CONST = 5;
            in var = 7;
            class LocalInnerClass {
                    System.out.println(CONST);          //OK
                    System.out.println(var);            //Error
            }
            LocalInnerClass inner  = new LocalInnerClass();          //OK
    }

    public void methodB{
            LocalInnerClass inner  = new LocalInnerClass();          //Error
    }
}
```

Anonieme Geneste Klassen (Anonymous Inner Classes)

- Deze klassen zijn eigenlijk identiek aan Lokale Geneste Klassen, maar deze hebben geen naam.
- Je maakt deze aan wanneer je een object aanmaakt
- Vorig jaar gebruikten we dit bv in Swing wanneer we een ActionListener wilden toevoegen aan een Button. We konden deze ActionListeners apart aanmaken en oproepen op de Button, ofwel ter plaatse een Anonieme instantie van de ActionListener aanmaken.

```
public class OuterClass {
    public void method(){
    SuperClass object1 = new SuberClass() {
            //Vervangen van de methoden die in SuperClass gebruikt worden
    }

    Interface object2 = new Interface() {
            //Implementatie van de methoden
    }
}
```

Static Geneste Klassen (Static Nested Classes)

Dit is identiek aan gewone Nested Classes op twee pagina's terug, maar men kan deze klassen enkel aanspreken op de OuterClass ipv op een object van de OuterClass.

```
public class OuterClass {
        public static class InnerClass{
        }
}

OuterClass.InnerClass inner = new OuterClass.InnerClass();
```

Enums werken ook op deze manier, dit zijn eigenlijk statische klassen, maar het woord Static wordt in hun geval weggelaten.

```
public class ColorEnums{
        public enum Color{
                …
        }
}

ColorEnums.Color color = ColorEnums.Color.RED;
```

RED is dus een waarde in de Enum Color

Lambda Expressions

- Lambda Expressions zijn simpelweg een kortere manier van schrijven, meerdere regels code kunnen vaak door 1 regel code vervangen worden, wat de leesbaarheid van de code zal verbeteren.
- Dit wordt gebruikt op plaatsen waar men bv een anonieme klasse/interface zou schrijven om deze ter plaatse te implementeren.

De klasse Text implementeert de Interface WordFilter, deze interface bevat maar 1 methode: isValid. De implementatie van deze methode in de klasse Text kan men telkens ter plaatse bepalen door een anonieme klasse te schrijven hiervoor. Men krijgt dan dit:

```
text.printFilteredWords(new WordFilter() {
        @Override
        public Boolean isValid(String s) {
                return s.length() > 4l
        }
}
```

Maar via een Lambda expression kan dit VEEL korter:

```
text.printFilteredWords((String s) -> s.length() > 4));
```

Er moet een WordFilter meegegeven worden, aangezien deze maar 1 methode heeft, weet de compiler vervolgens dat je stuk code de code in die methode voorstelt. Een Interface zoals WordFilter, die maar 1 methode bevat, noemen ze trouwens een **Functionele Interface**. In dit soort Interfaces is het aangeraden om trouwens de annotatie **@FunctionalInterface** bovenaan de Interface plaatsen, dit geeft een waarschuwing indien de Interface niet aan de voorwaarden zou voldoen.

Structuur van een Lamba Expression

(String s) -> s.contains("e");

- **String**
  - = Formele parameters
  - Lijst van de parameters in je methode, hier staan ronde haken rond.
  - Je hoeft dit niet expliciet mee te geven als de compiler kan afleiden welk type je parameter is.
  - Als je geen parameters meegeeft kan je gewoon **()** gebruiken.
  - De ronde haken kunnen we eventueel nog weglaten indien er maar **1** parameter is

  (String s) -> s.contains("e"); *OF* ***(s) -> s.contains("e");*** *OF* ***s -> s.contains("e");***

- **->** = Dit is de scheiding tussen de lijst van parameters en de inhoud van de methode
- **De inhoud (body)**
  - Hier staat de implementatie van de functionele methode.
  - Voor deze implementatie staat eigenlijk een impliciete return

    (String s) -> return s.contains("e");

  - In de methode waar je een Lambda gebruikt (bv in de main method) mag je parameternaam (bv. s) niet dezelfde zijn als reeds bestaande parameter.

    ```
    String s = "Hello";
    text.printFilteredWords(s -> s.contains("e"));        //ERROR
    ```

  - Ook mogen Lamba's enkel aan de variabelen van je omvattende methode indien deze CONSTANT zijn.

    ```
    final String c = "e";
    text.printFilteredWords(s -> s.contains(c));
    ```

  - **super** en **this** krijgen geen nieuw werkingsgebied binnen een Lambda, bij anonieme/geneste klassen is dit wel zo. Beiden worden dus bekeken in de context van de omgevende code.

Methode Referenties (Method References)

Method References worden gebruik in samenwerking met Lambda Expressions. Vaak bestaat een stuk code al dat je wil gebruiken, het volstaat dan om simpelweg een referentie te leggen naar dat stuk code.

Statische methoden

```
@FunctionalInterface
public Interface WordProcessor {
        public String process(String s);
}
```

In de klasse Text gebruiken we deze Interface:

```
public class Text {
        private String sentence;
        public Text(String sentence){
                this.sentence = sentence;
        }

        public void printProcessedWords(WordProcessor processor){
                for(String w : sentence.split(" ") {
                        System.out.println(processor.process(w));
                }
        }
}
```

De zin wordt gesplitst op spaties en elk woord wordt afgedrukt.
Als deze methode nu in je main method wil aanroepen:

```
text.printProcessedWords(s -> String.format("<<%s>>", s));
```

In plaatsd van de WordProcessor aan te maken tussen de haken die verwacht wordt, geven we meteen een Lamba expressive mee voor deze WordProcessor zijn functionele methode. Alle woorden worden afgedrukt en vervolgens omgeven voor << >>.

Uit gemak kunnen we nu deze Lamba Expression en anderen in een aparte Interface stoppen, bv TextUtil. We kunnen vervolgens kiezen uit een bepaald aantal methods die we willen toepassen op de woorden.

```
public interface TextUtil {
        public static String quote(String s) {
                return String.format("<<%s>>", s);
        }
}
```

Als we nu de code in de main method willen aanroepen kan dit zo:

text.printProcessedWords(s -> TextUtil.Quote(s));

Het hele stuk code tussen de ronde haken kunnen we nu vervangen door een **Methode Referentie**. We hoeven dan geen Lamba expressie meer te schrijven, we refereren gewoon naar een stuk code dat al klaar is.

text.printProcessedWords(TextUtil::quote);

De methode quote() in TextUtil heeft dezelde aantal parameters dan de methode process() in van de functionele interface. We hoeven geen Lamba meer te schrijven en we schrijven enkel nog de methode referentie.

Methoden van een gebonden object

Om dit soort methode referenties te gebruiken, moet er eerst een object aangemaakt worden waarin een methode zit die we nodig hebben.

TextPadder padder = new TextPadder();
text.printProcessedWords(padder::pad);

Zowel de pad() methode in TextPadder als de methode process() van de functionele interface verwachten een String als argument, we kunnen dit dus ook als een method reference schrijven.

Methoden van een ongebonden object

Dit soort methode referentie gebruik je als je een methode wil aanspreken van het object dat je meegeeft (bv (s)).

text.printProcessedWords(s -> s.toUpperCase());

Dit vervangen we dus door:

text.printProcessedWords(String::toUpperCase);

text is het object dat meegegeven wordt, maar we hangen hier niet aan vast, dit kan een ander object zijn. Daarom noemen we dit “ongebonden”.

Constructor referenties

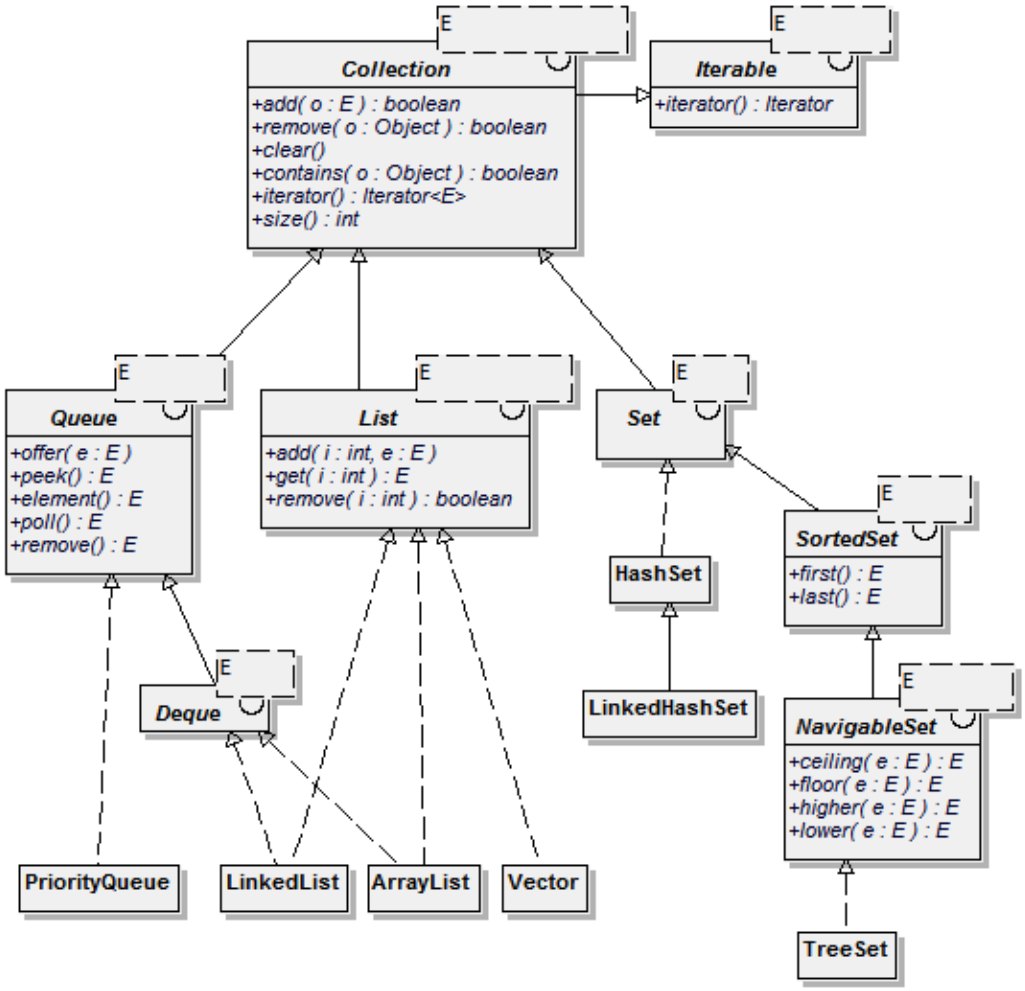
Met Constructor Referenties gaan we hetzelfde te werk, maar nu geven we eigenlijk een Object van een bepaalde klasse door aan de methode waar in we deze methode referentie gebruiken. We hebben bv een Interface NumberParser, hier in zit 1 methode genaamd Parse die een Long terug geeft. NumberParser wordt bv gebruikt als Object bij een methode genaamd PrintNumberValues, ipv via s -> new Long(s) te typen, vervangen we dit door een constructor reference.

text.printNumberValues(s -> new Long(s));

wordt dus

text.printNumberValues(Long::new);

- Vaak gebruiken we Arrays om objecten van eender welk type in een lijst op te slaan, maar deze hebben een groot nadeel: een vaste grootte.
- Als alternatief kunnen we het Collections Framework gebruiken, deze bestaat uit een aantal interfaces en implementaties om deze problemen op te lossen.
- Je hebt bv deze standaard mogelijkheden:
    - Objecten toevoegen
        - add()
    - Objecten verwijderen
        - remove()
    - Controleren of een Object in de verzameling zit
        - contains()
    - Een Object opvragen uit de verzameling
        - get()
    - Alle Objecten overlopen in de verzameling
        - forEach()
- De volgende figuur stelt het hele Collections Framework voor.



- Collections zijn allemaal van het Generic type **E**, tijdens het declareren en instantieren geven we dus mee welke type element deze zal bevatten
    - ArrayList<**Integer**> nummerkes = new ArrayList<**Integer**>();
    - We hoeven hierdoor ook niet te typecasten zoals bij een Array, we zijn op voorhand al zeker van het type element dat we van de Collection zullen verkrijgen.

- Sommige verzamelingen zijn **geordend**
    - Ieder element heeft een vaste plaats.
- Sommige geordende verzamelingen zijn ook **gesorteerd**
    - Gerangschikt volgens een bepaald algoritme
- Om alle elementen in een verzameling te overlopen moet we **itereren**.
    - Iedere Collection heeft een **Iterator** object, wat afkomstig is van de Interface **Iterator.**
    - De belangrijkste 2 methoden die je hier op kan uitvoeren zijn:
        - hasNext();
        - next();

```
Iterator<Integer> it = nummerkes.iterator();
while(it.hasNext()) {
        System.out.println(it.next());
}
```

    - Je kan dit ook met een ForEach lus doen, maar op de achtergrond wordt eigenlijk de **Iterator** gebruikt.

```
for(Integer i : nummerkes){
        System.out.println(i);
}
```

List

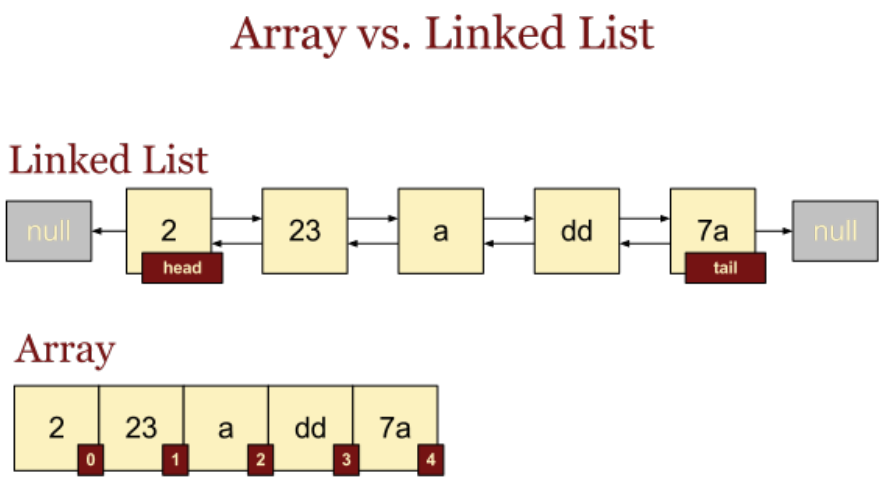
- Dit is een interface die overerft van **Collection** en dus alle methoden hier van in zicht heeft.
- Een List is **geordend** en er kunnen **duplicaten** voorkomen
- Ieder element heeft een vaste positive die wordt aangeduid met een **index**.
- Een List heeft de volgende methoden
    - add()
    - add(int index)
        - Toevoegen op een bepaalde plaats
    - remove(Object o)
    - clear()
    - contains()
    - iterator()
    - size()
    - set(int index)
        - Vervangen
    - get(int index)
    - remove(int index)

ArrayList

- Dit is de concrete implementatie van de List Interface.
- In de achtergrond wordt een array gebruikt waarvan de grootte dynamisch wordt aangepast.
- Het voordeel is dat je **snel** een willkeurige index-waarde kan **opvragen**.
    - Een ArrayList staat als 1 grote blok in het geheugen, daarom is het opvragen snel.
- Het **toevoegen** is **traag** omdat de grootte van de array steeds vergroot moet worden.
- Door **AutoBoxing** kan men ook primitieve datatypes toevoegen, men moet dus niet lijst.add(Integer(5)) typen, men kan dit korter doen door **lijst.add(5)**; te typen.

LinkedList

- Dit is een **dubbel gekoppelde lijst**, ieder element heeft dus een verwijziging naar het element ervoor en erna.
- Je kan deze lijst dus ook in 2 richtingen doorlopen
    - Van **head** naar **tail** of omgekeerd.
- **Toevoegen** gaat hierdoor veel **sneller**
    - De keten wordt geopened en het element wordt er tussen geschoven.
- Een element **opvragen** is **trager** aangezien men het element uit de keten moet halen en het element ervoor en erna aan elkaar moet koppelen



Zowel de ArrayList als de LinkedList kunnen trouwens in een object van de interface **List** gestopt worden, dit wordt **polymorfisme** genoemd.

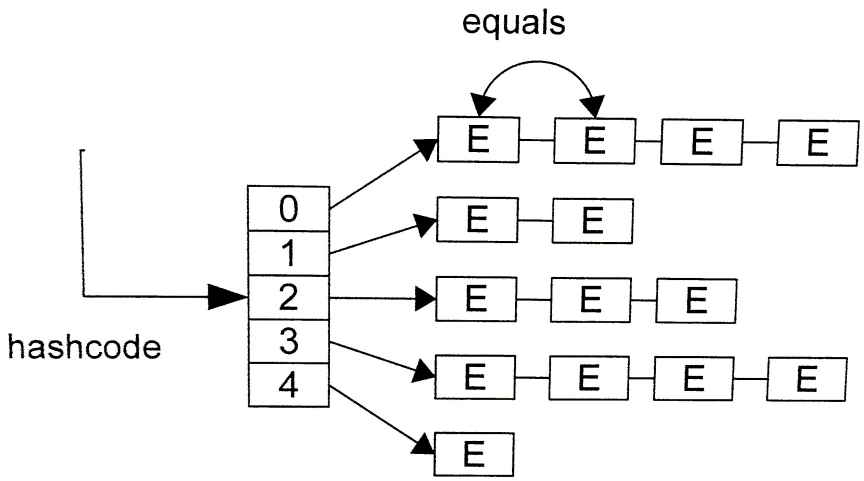
List<String> list = new LinkedList<>();

Set

- Dit is een verzameling van **unieke** objecten, er zijn dus ook **GEEN duplicaten**.
    - Dit wordt gecontroleerd aan de hand van de hashcode van ieder object
- De methoden **equals()** en **hashCode()** moeten dus ook geimplementeerd zijn in de objecten die men toevoegt, indien dit niet zo is wordt er gekeken naar de overervende klasse **Object**.
- Er wordt geen error gegeven indien je een duplicaat toevoegt, het object zal er simpelweg maar **1x** inzitten.

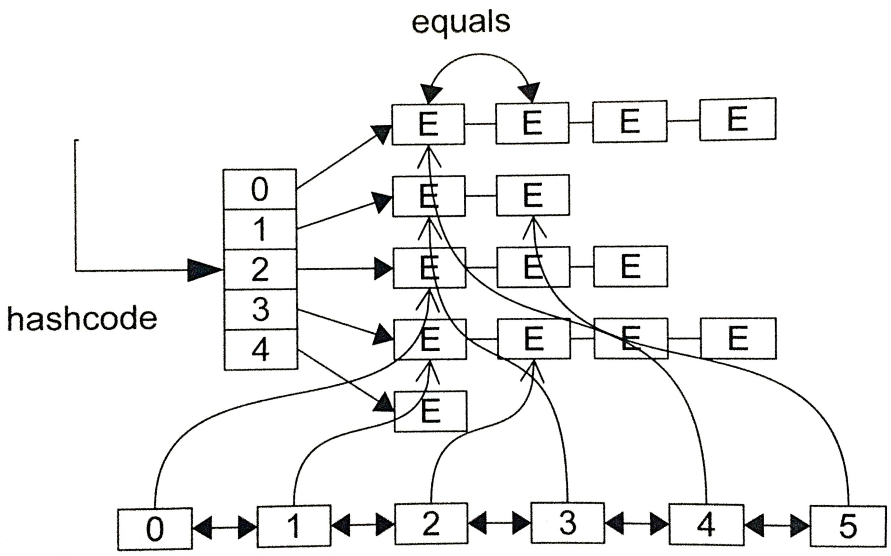
HashSet

- Dit is een **ongeordende** en **ongesorteerde** verzameling.
- Er zijn **GEEN duplicaten**.
- Alle elementen worden bijgehouden in een **hashtable**
- Hiervoor wordt de **hashcode** van ieder element gebruikt.
- De **hashcode** zorgt voor de **index** in deze lijst, achter iedere index zit een **gelinkte lijst** die effectief de elementen bevat
- Eerst wordt dus ook naar de **hashcode** gekeken en wordt de index bepaald, nadien wordt de methode **equals()** gebruikt of het element al in de achterliggende gelinkte lijst zit.
- Bij het opvragen is de volgorde **niet gegarandeerd**, de volgorde kan immers wijzigen bij het toevoegen of verwijderen van een element.



LinkedHashSet

- Dit is een **geordende** en **ongesorteerde** verzameling.
- Deze heeft dezelfde voordelen als een HashSet namelijk, **snel toevoegen, opvragen en verwijderen**
- Bijkomend voordeel is hier nog een dubbel **gekoppelde lijst** wordt voorzien om de **volgorde** bij te houden.



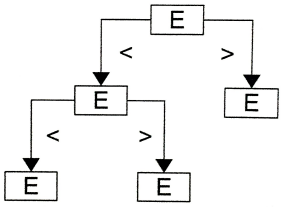
SortedSet & NavigableSet

- SortedSet is een interface die afgeleid is van Set, de elementen zijn **geordend** en **gesorteerd** volgens een bepaald algoritme.
- Je kan hiermee ook het eerste en laatste element opvragen
  - first()
  - last()

- NavigableSet is een interface afgeleid van SortedSet, deze heeft de extra mogelijkheid om een element op te vragen dat het dichtst bij een opgegeven element zit.
  - ceiling()
  - floor()
  - higher()
  - lower()

TreeSet

- Dit is de concrete implementatie van NavigableSet (en dus ook SortedSet aangezien deze er van afgeleid is). Deze zijn dus ook **geordend** en **gesorteerd**
- De elementen worden in een boomstructuur begehouden (Tree).
- Als een element wordt toegevoegd wordt de TreeSet **live** gesorteerd.
- Het **toevoegen en verwijderen** is **traag** aangezien het tijd vergt om de boomstructuur te doorlopen.



Queue

- Een **geordende** verzameling waar in de elementen één voor één kunnen opgevraagd en verwijderd worden zoals in een wachtrij.
- Een Queue heeft de volgende extra methoden:
    - offer()
        - Voegt een element toe
    - peek()
        - Geeft het eerstvolgende element, maar verwijdert het niet (geeft null als de queue leeg is)
    - element()
        - Geeft het eerstvolgende element, maar verwijdert het niet (geeft Exception als de queue leeg is)
    - poll()
        - Geeft het eerstvolgende element en verwijdert het (geeft null als de queue leeg is)
    - remove()
        - Geeft het eerstvolgende element en verwijdert het (geeft Exception als de queue leeg is)
- Een LinkedList is ook een implementatie van Queue
- Deze gebruikt het FIFO principe (First-In, First-Out)

PriorityQueue

- Dit is ook een Queue, maar **gesorteerd** volgens hun natuurlijk volgorde of volgens het algoritme van een Comparator.

Deque

- Dit is een Double Ended Queue
- Een gewone Queue kan je enkel aan het begin aanspreken, deze kan je langs beide zijden aanspreken
    - offerFirst()
    - peekFirst()
    - pollFirst()
    - offerLast()
    - peekLast()
    - pollLast()

Sorteren van verzamelingen

- Er is **Intrinsiek Sorteren**, de verzameling draagt zelf zorg voor de interne sortering.
- Er is ook **Eenmalig Sorteren**, dit is van toepassing op Lists en doe je met de methode **sort();**
    - De volgorde wordt wel niet behouden na het toevoegen/verwijderen van een element.

Intrinsiek sorteren

- Er zijn 2 manieren tot intrinsiek sorteren
    - Natuurlijke volgorde
        - De elementen die toegevoegd worden moeten de Comparable<T> interface implementeren.
        - Deze moeten de methode compareTo aanbieden en hier in wordt bepaalde op welke property het verschil tussen het huidige object **this** wordt vergeleken met het meegegeven object in de methode.
    - D.m.v een Comparator
        - De elementen die toegevoegd worden moeten via hun constructor een hulpobject (de comparator) ter beschikking stellen van de sorterende verzameling. Deze comparator implementeert de interface **Comparator** die slechts 1 methode in zich heeft:

          public int compare(T object1, T object2)

        - Een Comparator gebruiken doe je als het element geen natuurlijke volgorde heeft en dus Comparable niet implementeert.
        - Dit doe je dmv een nieuwe klasse te maken die Comparator implementeert en waar in de compare methode wordt gedefinieerd hoe de vergelijking moet gebeuren (volgens lengte, volume, leeftijd, etc)
        - Deze Comparator kan je vervolgens meegeven aan de constructor van bv een TreeSet

          SortedSet<Box> boxes = new TreeSet(new BoxLengthComparator());

        - De TreeSet verzameling zal de meegegeven Comparator gebruiken om de volgorde te bepalen van de elementen.

Streams/Consumers

- Tot nu toe gebruikten we telkens een For lus, forEach of een Iterator om door een verzameling te gaan.
- In Java 8 is het concept van Streams toegevoegd, dit vormt de verzameling om tot een **stroom** van **objecten**.
- Elk object uit de **stream** kan je behandelen door **ForEach**() op de stream uit te voeren.
- Aan de **ForEach** moet je wel een object meegegeven dat **Consumer** implementeert.
- De Stream zal vervolgens de objecten één voor één uit de verzameling halen en aan de **Consumer** geven, die er op zijn beurt iets mee doet.
- **Consumer** is een interface die **1 abstracte methode** heeft (=functionele interface)
- We kunnen aan de ForEach d.m.v. een Lambda Expression of Method Reference bepalen wat de Consumer met elk object uit de stream zal doen.

words.stream().forEach(System.out::println);

Filters/Predikaten

Filters

- Filters worden gebruikt op Streams, je kan bv alle woorden uit een verzameling halen die de letter “e” bevatten

words.stream().filter(s -> s.contains(“e”));

Predikaten

- De Lamba Expression die aan de filter wordt meegegeven wordt een **Predicate** genoemd
- Je kan deze ook apart aanmaken en dan pas meegeven aan de Filter

Predicate<String> predikaat = s -> s.length() > 3;
Predicate<String> predikaat2 = s -> s.contains(“e”);
words.stream().filter(predikaat).filter(predikaat2).forEach(System.out::println);
Alle woorden met een lengte > 3 en die de letter **e** bevatten worden afgedrukt.

Mappen

- Je kan ook Filters gebruiken die je een heel nieuw datatype terug geven, dit zijn methodes die de interface Stream aanbiedt.
- Het omzetten van het ene datatype naar een ander datatype noemen we **Mappen**.
- Dit kan je bv gebruiken als je alle lengtes (ints) van de woorden wil afdrukken

words.stream().mapToInt(String::length).forEach(System.out::println);
Alle lengtes van alle woorden worden afgedrukt

- Je kan ook je eigen definitie van een Map ingeven

words.stream().map(String::toUpperCase).forEach(System.out::println);
Alle woorden naar hoofdlettes omgezet en vervolgens 1 voor 1 afgedrukt

Reduceren

- Je kan ook streams gaan **reduceren**, dit wil zeggen dat je de hele stream gaat omzetten naar één iets (som, minimum, maximum, etc).

  words.stream().mapToInt(String::length).sum();
  De som van alle lengtes van alle woorden in de verzameling.

  words.stream.reduce(“”,(String t, Stringu) -> t+u);
  Alle woorden in 1 String plakken

- Als je dit soorten waarden in een variabele wil opslaan moet je hiervoor geen Integers, Doubles, etc gebruiken maar **OptionalInt, OptionalDouble**, etc

  OptionalInt max = words.stream().mapToInt(String::length).max();
  De lengte van het langste woord wordt in de variabele **max** gestopt

Collecteren

- Je kan ook een hele nieuwe verzameling maken van een stream, dit noemt men **collecteren.**

  List<String> shortWords =
  words.stream().filter(s -> s.length() > 3).collect(Collectors.ToList());
  shortWords.stream().forEach(System.out::println);
  Dit verzamelt alle woorden met een lengte < 3 en stopt ze in een nieuwe List van Strings. Vervolgens wordt deze nieuwe List afgedrukt.

Functionele Interface types / Opslaan van Method References

- Er zijn 4 soorten Functionele Interfaces (=interfaces met maar 1 methode)
- Deze kan je natuurlijk als datatype gebruiken.
    - Bv Consumer<boolean> conObject;
- In zo een object kan je vervolgens een methode stoppen, dit object kan je dan weer meegeven aan een andere klasse via de constructor.
- Die klasse kan vervolgens de methode uitvoeren waar naar verwezen wordt door de **accept()** methode uit te voeren op het object.
- Welk object type je kiest hangt van de methode af die je er wil in stoppen.
- **Consumer**
    - Methoden die parameters meekrijgen maar niets returnen.
- **Supplier**
    - Methoden die iets returnen maar niets meekrijgen
- **Function**
    - Methoden die iets mee krijgen en iets returnen
- **Predicate**
    - Deze krijgen een vergelijking binnen geven vervolgens een Boolean terug
    - Dit type gebruik je bv bij Filters, de Lambda die je aan een Filter meegeeft is eigenlijk een Predicate object
        - woordenlijst.stream().filter(***s -> s.length > 4***).forEach(System.out::println);
    - De Predicate geeft True terug als de lengte van het woord groter is dan 4.
    - De Filter houdt dus enkel de woorden over waarvan deze **True** is.
    - Enkel die woorden worden uiteindelijk ook afgedrukt.

Het vorige zal je 99% zeker op het examen moeten kunnen!
Kijk op BB naar de oefening ter voorbereiding op het examen:

- In de opdracht staat vermeld dat de Klant klasse niet zelf mag printen.
- Dit moet dus met een omweg gebeuren = method reference in een object stoppen en meegeven.
- In dit object wordt de methode printStatus(boolean status) gestopt via this::printStatus
- Klant krijgt dus een Consumer object binnen via de constructor (bv het object printStatus)
    - **Het is Consumer omdat printStatus enkel iets binnen krijgt en niets returned!**
- Zodra de klant wil printen doet deze **printStatus.accept(true)**
- Vervolgens wordt de methode naar waar het object verwijst aangesproken en wordt er **true** meegegeven als argument
- Deze drukt vervolgens een lachende smiley af, bij false drukt deze een verdrietige smiley af.

De Interface Map

- De interface Map en zijn subinterfaces vormen een aparte hiërarchie.
- Elk element in een Map is eigenlijk een paar van Objecten, een **key** en een **value**.

Map

- Dit is een gewone verzameling van keys en values, deze heeft de volgende methoden:
    - put(key, value) = key/value paar toevoegen
    - get(key) = key meegeven, value terug krijgen
    - remove(Object k) = verwijdert een key/value paar
    - keyset() = Geeft de verzameling terug in de vorm van een Set
    - forEach() = Door de Map gaan met een functionele methode

HashMap

- Dit is een **ongeordende** en **ongesorteerde** Map.
- Je mag **null** eenmaal meegeven als key, meermaals als value.
- **Key** objecten moet uniek zijn!

  Map<String, Integer> ingredients = new HashMap<>();
  ingredients.put("Potatoes", 5);
  ingredients.put("Carrots", 4);
  System.out.println(ingredients.get("Carrots"));
  Dit zal 4 afdrukken.
- Je kan ook de hele Map overlopen met een forEach()

  ingredients.forEach((k,v) -> System.out.println(k + ": " + v));
  Dit drukt alle key/value paren af

LinkedHashMap

- Dit is een **geordende** en **ongesorteerde** Map.
- Er wordt bijkomend een gelinkte lijst voorzien om de volgorde bij te houden.
- Dit werkt hetzelfde als bij de LinkedHashSet
- De volgorde blijft dus hetzelfde zoals ze de paren aanvankelijk zijn toegevoegd.

SortedMap & NavigableMap

- De SortedMap is een **geordende** en **gesorteerde** Map
- De **keys** moeten zoals bij de SortedSet de interface **Comparable** implementeren ofwel moet een **Comparator** aan de verzameling meegegeven worden.
- Je kan bij deze verzameling ook weer het eerste en laatste element opvragen
  - **firstKey();**
  - **lastKey();**
- NavigableMap is afgeleid van SortedMap, als extra functie kan je hier een **key** opvragen die het dichtst bij een opgegeven **key** zit.

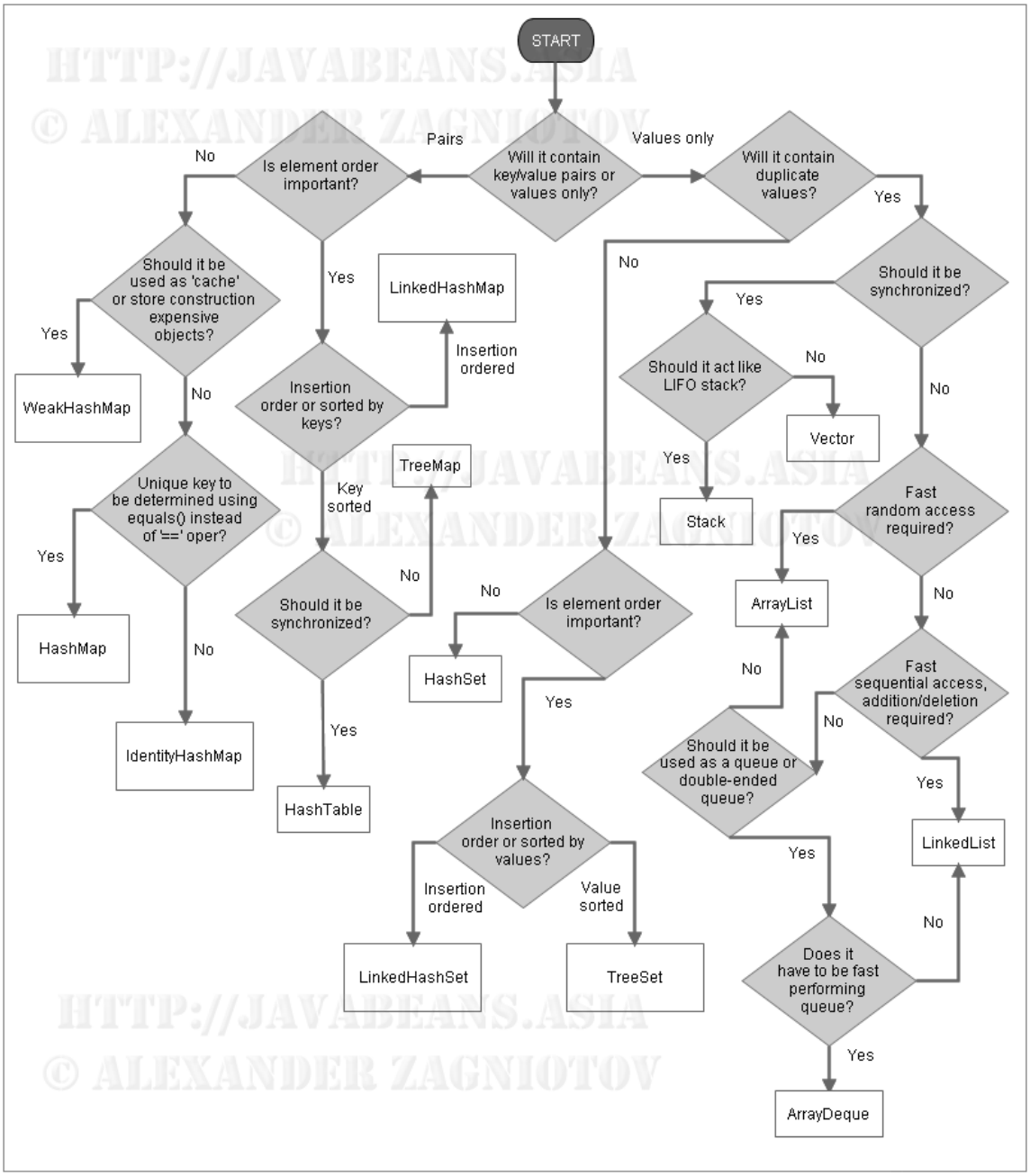
TreeMap

- Dit werkt hetzelfde als een TreeSet, maar dan met paren.
- Er is een boomstructuur en de **keys** worden **live** gesorteerd.

Eigenschappen van alle Lists, Sets, Queues en Maps

Verzameling	Uniek	Geordend	Gesorteerd	Random Access
ArrayList		X		X
LinkedList		X		
HashSet	X			
LinkedHashSet	X	X		
TreeSet	X	X	X	
PriorityQueue		X	X	
Deque		X		
HashMap	X			
LinkedHashMap	X	X		
TreeMap	X	X	X	

Flowchart Collections



Exceptions

- Een Exception is een gebeurtenis die optreedt tijdens de uitvoering een programma en die de normale loop van het programme verstoort.
- Voorbeeld

```
Scanner scan = new Scanner(System.in);
int num = Integer.ParseInt(scan.next());
int denom = Integer.ParseInt(scan.next());
int division = num / denom;
```

- Als je bv een letter ingeeft zal de volgende Exception optreden:

Exception in thread "main" java.lang.NumberFormatException: For input String "a"

Try/Catch

```
try {
        Scanner scan = new Scanner(System.in);
        int num = Integer.ParseInt(scan.next());
        int denom = Integer.ParseInt(scan.next());
        int division = num / denom;
}
catch(NumberFormatException ex) {
System.out.println("Invalid Number");
}
```

Meerdere Exceptions opvangen

```
try {
        …
}
catch(NumberFormatException ex) {
        …
}
catch (ArithmeticException ex){
        …
}
```

Meerdere Exceptions onder een gemeenschappelijk Exception klasse opvangen

```
try {
        …
}
catch(RuntimeException ex){
        …
}
```

RuntimeException is de superklasse van zowel NumberFormatException als ArithmeticException.

Alle exceptions opvangen

```
try {
        …
}
catch(Exception ex){
        …
}
```

Dit wordt wel niet aangeraden om je niet precies welke Exception zich heeft voorgedaan. Je kan dit wel te weten komen vias ex.getClass(), maar dit maakt de code onoverzichtelijk.

Finally

```
try {
        …
}
catch(){
        …
}
finally{
        …
}
```

Of er nu een Exception optreedt, een return statement in de try uitgevoerd wordt, of er helemaal geen Exception optreedt, de **finally** blok wordt altijd uitgevoerd. Dit kan handig zijn om bv bestanden of netwerkconnecties op te ruimen).

Exceptions genereren

- Exception **throw** je

```
throw new ExceptionClass();
```

- Dit kan je bv zelf toepassen in een klasse Rectangle

```
public void setHeight(int height) throws Exception {
    if (height < 0) {
            throw new Exception("Negative Height");
    } else {
            this.height = height;
    }
}
```

- We kunnen de ***throws Exception*** ook in de constructor van de klasse plaatsen

```
public Rectangle(int w, int h) throews Exception {
    super(x,y);
    setHeight(h);
    setWidth(w);
    count++;
}
```

- Er zijn verschillende soorten Exceptions
  - Checked Exceptions
    - De compiler controleert deze
    - Je moet deze opvangen of in de methode **throws MyExceptionClass** voorzien, anders kan de code kan niet compileren
  - Runtime Exceptions
    - Deze worden niet door de compiler gecontroleerd, deze moet je dus ook niet opvangen.
    - Meestal duiden deze op programmeerfouten (denk aan delen door 0 bij het inlezen van een nummer via het toetsenbord).

Een Exception klasse maken

```
public class NegativeSizeException extends RuntimeException {

        public NegativeSizeException(){
                super();
        }
        public NegativeSizeException(String message){
                super(message);
        }
}
```

Exceptions opvangen, inpakken en verder gooien

- Dit wordt vaak gebruikt om een Exception weg te schrijven in een log bestand en deze vervolgens weer verder te gooien.

```
try {
        …
}
catch(Exception ex){
        System.out.println(ex.Message());
        throw ex;
}
```

Root Cause

- Exceptions hebben een root cause die ingesteld kan worden via de constructor.

```
public NegativeSizeException(String message, Throwable cause) {
    super(message,cause);
}
```

- Unix system: 1 basismap, de **root** of **/**
- Windows systemen: schijfletters die ieder op zich een basismap voorstellen

De Interface Path

- Bestanden en mappen kan je in Java benaderen via de Interface **Paths**
- Objecten die deze Interface implementeren stellen een pad voor.
- Men kan objecten van deze Interface bekomen via de statisch methoden van de klasse **Paths.**

Windows:

Path path = Paths.get("C:\\data\\folder1\\file1.text");

Unix:

Path path = Paths.get("/data/folder1/file1.txt");

- Op Objecten van Path kan je de volgende methoden uitvoeren
  - compareTo
  - endsWith
  - startsWith
  - resolve
    - Nieuw pad meegeven, maakt een nieuwe pad aan op basis van dit en het pad in het object
  - relativize
    - Berekent het relatieve pad tov het meegegeven pad
  - toAbsolutePath
  - toRealPath
    - Bij bv gebruik van hoofdletters kan je deze methode gebruiken om het effectieve juiste pad te bekomen.
  - isAbsolute
  - toFile
  - iterator
  - getFileName
  - getRoot

FileSystem

- Deze objecten stellen een bestandssysteem voor.
- Je kan het huidige bestandssysteem opvragen door
  FileSystem fs = FileSystems.getDefault();
- Als je de **separator** wil ophalen van het bestandsssyteem kan je dit zo
  fs.getSeparator();
  \ = Windows
  / = Unix

Files

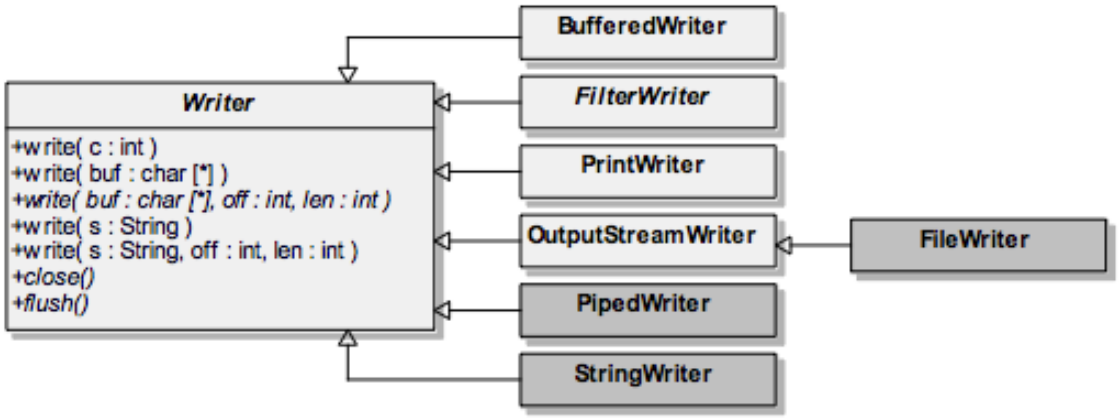
- Deze Utility klasse (statisch) dient om bestanden of mappen effectief te benaderen en er acties op uit te voeren.
    - copy
    - move
    - createDirectory
    - createDirectories
    - createFile
    - delete
    - readAttributes
    - setAttributes
    - readAllLines
    - write

Streams

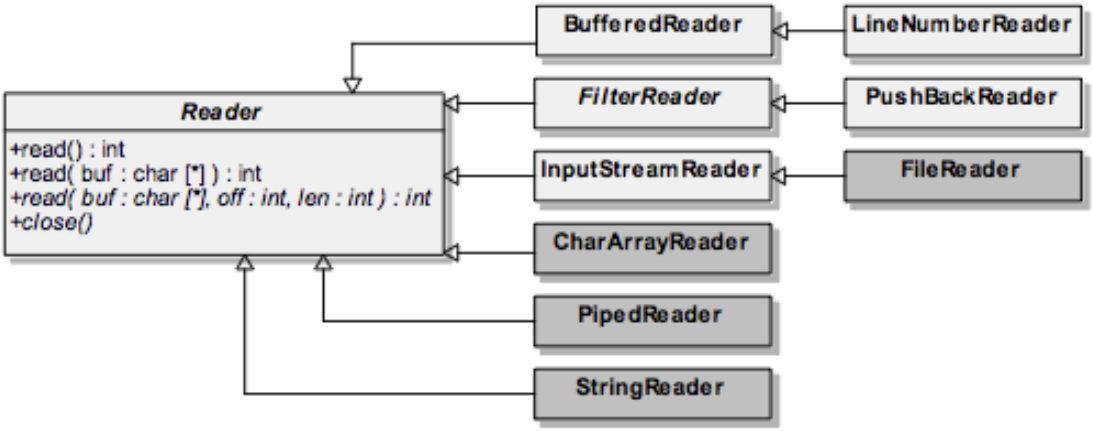
- Lezen en schrijven gebeurt via Streams, deze hebben altijd een bron en een bestemming.
- Als we bv System.out.println doen, dan is het scherm onze bestemming.
- Een stream is een stroom van gegevens, dit kan van een tekstbestand, het toetsenbord, geheugenbuffers of netwerkconnecties komen.
- Om te lezen/schrijven met een stream moeten 3 stappen ondernomen worden
    - De Stream openen
    - Lezen/Schrijven zolang er gegeven beschikbaar zijn.
    - De Stream sluiten

Stream-klassen

- Er zijn 2 grote soorten Streams
    - Character-Streams
        - 16-bits karakters, Unicode in Java
    - Byte-Streams
        - 8-bits gegevens, binair.
- Hierin worden nog eens 2 soorten onderscheiden
    - Data-Sink streams
        - Deze worden gebruikt om gegevens te sturen of te lezen van bestanden, strings, arrays, pipes.
    - Process-Streams
        - Deze dienen voor het encoderen/decoderen van gegevens.



Afbeelding 23: Character streams: Writer



Afbeelding 24: Character streams: Reader

Opmerking: lichtgrijze klassen zijn Processing Streams, donkergrijze klassen zijn Data Sink Streams.

- Er zijn 2 supperklassen, **Reader** en **Writer**.
- De subklassen voor ieder zijn
    - BufferedReader / BufferedWriter
        - De gegevens worden gebuffered, deze komen dus in grote blokken binnen of worden zo weggeschreven = Hogere snelheid.
        - LineNumberReader is een subklasse hiervan, hierdoor kan je gegevens in regels inlezen.
    - FilterReader / FilterWriter
        - Abstracte klassen om zelf Processing Streams te maken.
    - InputStreamReader / outputStreamWriter
        - De klassen link Character Streams aan ByteStreams
        - Deze verzorgen de omzetting tussen bytes en karakters
    - CharArrayReader
        - Deze kan gegevens uit een reeks lezen
    - PipedReader / PipedWriter
        - Gegevens van een **|** lezen of schrijven
    - StringReader
        - Gegevens van een String lezen of schrijven.

Tekst schrijven naar een bestand

```
FileWriter file = null;
try{
        file = new FileWriter("MyFile.txt");
        file.write("Hello");
        file.write("World");
}
catch (IOException ex) {
        System.out.println("Oops, something went wrong!");
}
finally{
        try{
                if (file != null){
                        file.close();
                }
        }
        catch(IOException ex){
                System.err.println("Error closing file!");
        }
}
```

FileWriter is een subklasse van OutputStreamWriter, die zijn op beurt een subklasse is van Writer. De methode **write()** die we gebruiken hoort bij de superklasse **Writer**.

De try/catch die we uitvoeren in bovenstaande code kan omslachtig zijn, ook het sluiten van het bestand moet op deze manier telkens in een **finally** blok gebeuren. Om dit te verkorteren kunnen we gebruik maken van een **try with resources**. De FileWriter wordft aangemaakt tussen ronde haken, deze bestaat dus ook alleen in de try blok. Na de try wordt deze automatisch afgesloten in de achtergrond. Dit werkt gelijkaardig aan **using()** in C#.

```
try (FileWriter file = new FileWriter("MyFile.txt")) {
        file.write("Hello");
        file.write("World");
}
catch(IOException ex) {
        System.out.println("Oops, something went wrong!");
}
```

Je kan ook meerdere resources meegeven aan de try

```
try (   FileWriter writer1 = new FileWriter("bestand1.txt");
        FileWriter writer2 = new FileWriter("bestand2.txt")) {
        …
}
```

In plaats van een String als pad mee te geven kan je ook een Path meegeven

```
Path path = Paths.get("File.txt");
FileWriter file = new FileWriter("path.toFile());
```

Tekst lezen van een bestand (karakter per karakter)

```
try(FileReader reader = new FileReader("MyFile.txt")) {
        int character;
        while((character = file.read()) != -1) {
                System.out.print((char) character);
        }
}
catch(IOException ex) {
        System.out.println(ex.getMessage());
}
```

Tekst lezen van een bestand (lijn per lijn)

```
try(    FileReader file = new FileReader("MyFile.txt");
        BufferedReader reader = new BufferedReader(file)) {
        String line = null;
        while((line = reader.readLine()) != null) {
                System.out.println(line);
        }
}
catch(IOException ex) {
        System.out.println("Oops, something went wrong");
}
```

Opmerking: In plaats van een FileReader te gebruiken die je vervolgens meegeeft aan de BufferedReader, kan je ook een Path object gebruiken

```
Path path = Paths.get("MyFile.txt");
try (BufferedReader reader = Files.newBufferedReader(path)) {
        …
}
```

Binaire gegevens schrijven naar een bestand

Maak hiervoor gebruik van een FileOutputStream en de methode write(), de rest is identiek aan een CharacterStream.

Binaire gegevens lezen van een bestand

Maak hiervoor gebruik van een FileInputStream en de methode read(), de rest is identiek aan een CharacterStream.

Encodering

- Als we niet expliciet opgeven welke encodering we willen gebruiken, dan wordt het standaard van besturingssysteem gebruikt, bv UTF-8.
- Maar je kan dit ook expliciet vermelden

```
FileOutputStream out = new FileOutputStream("MyEncodedFile.txt");
OutputStreamWriter writer = new OutputStreamWriter(out, "UTF-8");
```

Bij een BufferedWriter moet je dit doen sinds JDK-8

Files.newBufferedWriter(path, Charset.forName("UTF-8"));

Serializable

- Dit pas je toe op klassen waarvan je Objecten in hun geheel wil wegschrijven
- Deze klassen moet de Interface **Serializable** implementeren
- De objecten zelf kan je vervolgen wegschrijven met een **ObjectOutputStream**

```
try {   FileOutputStream file = new FileOutputStream("MyFile.ser");
        ObjectOutputStream out = new ObjectOutputStream(file);) {
        out.writeObject(text);
        out.writeObject(date);
}
catch(IOException ex) {
        System.out.println(ex.getMessage());
}
```

- Als een bepaalde klasse een superklasse is voor andere klassen en Serializable implementeert, zullen deze subklassen tevens Serizalizable zijn.
- Als een Object referenties heeft naar andere Objecten, dienen ook deze Objecten serializable te zijn.

Transients

- Alle variabelen worden standaard geserialiseerd, deze worden dus weggeschreven in het Object bestand
- Als men bepaalde eigenschappen niet wil laten wegschrijven, dan men dit definieren door er **transient** voor te plaatsen bij de declaratie.
- Men doet dit bv ook bij Klassen die een Object gebruiken waarvan die klasse een nieuwe Thread creeert. Dit soort klassen zijn niet serializable, door de property van die klasse **transient** te maken, kan men de andere klasse toch serializable maken.

transient int aantal;

Versienummering

- Alle serializable klassen moeten een uniek ID nummer krijgen, de compiler berekent dit nummer zelf.
- De volgende eigenschap wordt door de compiler toegevoegd

public static final long serialVersionUID = xxxxxx;

Properties

- Vaak wil je dat als je een programma hebt gemaakt, dat bepaalde eigenschappen worden opgeslagen.

```
try (FileOutputStream out = new FileOutputStream("Application.properties");) {
        Properties atts = new Properties();
        atts.setProperty("Attribute1", "Value1");
        atts.setProperty("Attribute2", "Value2");
        atts.store(out, "Application.properties");
}
catch(IOException ex){
        …
}
```

- De volgende keer als je het opent, worden deze dan terug ingeladen.

```
try(FileInputStream in = new FileInputStream("Application.properties");) {
        Properties atts = new Properties();
        atts.load(in);
        atts.list(System.out);
}
catch(IOException ex){
        …
}
```

System Resources

- Toegang ot bv het toetsenbord, afdrukken van gegevens op het scerm, opvragen van de systeemtijd, etc.
- Dit wordt geregeld door de JRE en wordt aangeboden via de klassen System en Runtime.
- System = platformonafhankelijk = geen objecten = statisch = final
- Runtime = platforafhankelijk

Gegevens inlezen via het toetsenbord

```
try (Scanner scan = new Scanner(System.in)) {
        System.out.println("Please enter your name and age:");
        String name = scan.next();
        int age = scan.nextInt();
        System.out.printf("Hello %s, your age %d years old.", name, age);
}
```

Gegevens afdrukken op het scherm

- Dit gebeurt via de standaard OutputStream **System.out**.

```
System.out.print("test");              //geen nieuwe lijn na het printen
System.out.println("test");            //nieuwe lijn na het printen
System.out.printf("%d plus %d is %d", 1,2,3);        //print 3 af
```

Errors afdrukken op het scherm

```
System.err.println("This is an error message");
```

De Klasse Console

- Om het inlezen van gegevens via de console te vergemakkelijken tov het inlezen met Streams, kan je de Console klasse gebruiken.
- Het object kan je opvragen via **System.console()**

```
Console console = System.console();
if(console == null) return;
String name = console.readLine("Name: ");
console.printf("Your name is %s\n", name);
char[] password = console.readPassword("Password: ");
console.printf("Your new password is %s\n", new String(password));
```

System Properties

- We kunnen via de System klasse ook een properties object opvragen:
  System.getProperties();
- Hier in zitten de oa deze eigenschappen opgeslagen:
  - file.separator (bv /)
  - java.class.path
  - java.class.version
  - java.hom
  - java.vendor
  - java.vendor.url
  - java.version
  - os.arch
  - os.name
  - os.version
  - path.separator (bv :)
  - user.dir
  - user.home
  - user.name
- Enkel één van deze eigenschappen kan via: ***System.getProperty();***
- Je kan ook eigenschappen aanpassen
  - ***System.setProperties();***
  - ***System.setProperty();***

Finalization & Garabage Collection

- Objecten waarvan geen referentie meer bestaat worden automatisch opgeruimd door de garbage collection
- Als er nog dingen moeten gebeuren na het opruimen kan je dit door een methode aan te maken die **finalize** heet

```
protected void finalize() throws Throwable {
    //cleanup code
    super.finalize();
}
```

Enkele andere methoden

```
arraycopy()          //gegevens van 1 array naar een andere kopieren
currentTimeMillis()  //Tijd in ms sinds 01/01/1970
nanoTime()           //Tijd in ns
exit()               //JRE afsluiten
loadLibrary()        //Externe module laden
```

Java via de CommandLine

- Om java files te compileren via de commandline moet je commandline/terminal de java map weten staan, in de map wordt immers **javac** gebruikt voor het compileren.
- Dit doe je door de java map aan de Path variabele van je systeem toe te voegen
- Configuratiescherm -> Systeem -> Geavanceerde Systeeminstelling -> Omgevingsvariabelen
- Voeg de volgende toe
    - JAVA_HOME
    - C:\Program Files\Java\jdk1.8.0_11
- Wijzig dan de Path variabele
    - Path
    - plaats ***%JAVA_HOME%\bin;*** aan het begin van de String

Compileren

- Om broncode te compileren gebruik je dus **javac** in je commandline
- Als je deze enkel zo intypt krijg je de help te zien
- Om deze te gebruiken moet je de naam van je broncode meegeven, bv:
    - ***javac –d bin src\eu\noelvaes\hello\HelloWorld.java***
    - Het pad stelt de structuur van de **package** voor waar de java file in zit
- Hij zal vervolgens het java bestand compileren naar een **.class** bestand in de **bin** map.
- Als je alle bestanden in **hello** wil compileren gebruik je ***.java**

Uitvoeren

- Om bestanden te runnen moet je het classpath meegeven, bv:
    - ***java –cp bin eu.noelvaes.hello.HelloWorld***
- De pakketnaam wordt automatisch omgezet in een pad zodat de JVM de klassebestanden kan vinden.

JAR bestanden

- Als je je programma wil afleveren aan iemand kan je er een **JAR** file van maken (Java Archive)
- Deze bundelen de klasse en andere bestanden in 1 bestand
- Voordelen
    - Beveiliging: Digitale handtekening
    - Snelheid: Het download van 1 bestand is sneller
    - Compressie: Bestanden in de JAR worden gecomprimeerd

Bestand aanmaken

jar cfe HelloWorld.jar –C bin
cf = create file
e = Manifest informatie toevoegen = Zeggen welke klasse de main() bevat

Inhoud van een jar bestand bekijken

jar tf HelloWorld.jar

Bestand uitpakken

jar xf HelloWorld.jar

Jar bestand *opnemen* in het classpath

jar –cp HelloWorld.jar eu.noelvaes.hello.HelloWorld

Jar bestand runnen

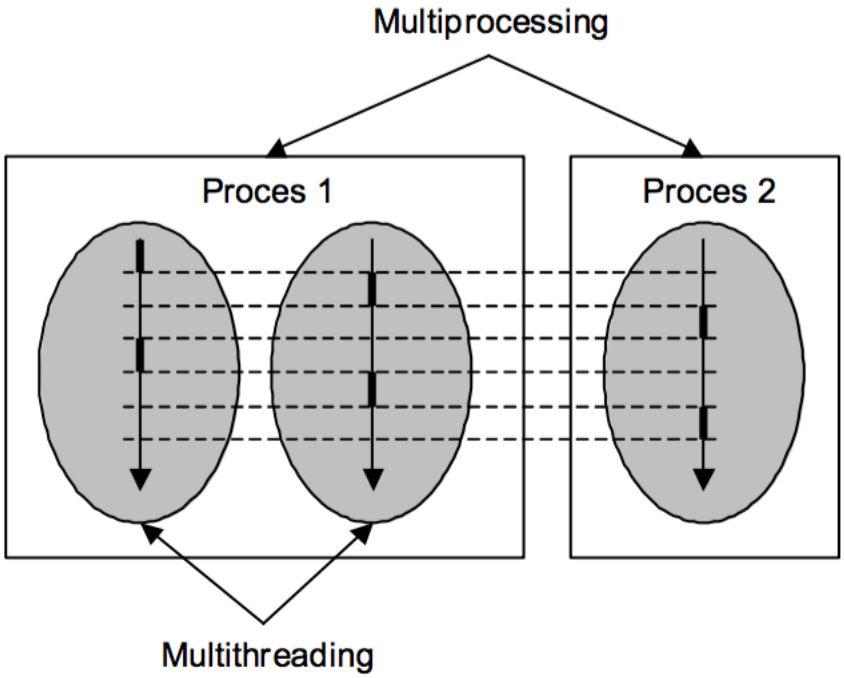
java –jar HelloWorld.jar

Resources uit een jar bestand lezen

```
try(InputStream in = getClass().getResourcesAsStream("Hello.jar")) {
        Properties props = new Properties();
        props.load(in);
        text = props.getProperty("text");
}
catch(IOException ex) {
e.printStackTrace();
}
```

Multithreading

- **Multiprocessing** = gelijktijdig uitvoeren van verschillende processen op één machine.
  - **Elk proces heeft zijn eigen stuk geheugenruimte!**
  - **Het Operating System is hier verantwoordelijk voor!**
- **Multithreading** = één proces beschikt over meerdere uitvoeringsaders (threads).
  - **Deze maken gebruik van hetzelfde stuk geheugenruimte!**
  - **De programmeur moet zelf nieuwe threads creëren!**
- Het wisselen tussen threads noemt men multitasking.



Afbeelding 39: Meerdere threads binnen één proces

Een nieuwe thread creëren

- Dit doe je via de klasse Thread door er een object van aan te maken
- Je kan de thread vervolgens starten met de methode **start()**

```
Thread thread = new Thread();
thread.start();
```

- Je kan ook de interface **Runnable** implementeren en de methode **run()** uitvoeren in Main.

```
public class CharacterPrinter implements Runnable {
        private char c;
        private int count;
        public CharacterPrinter(char c, int count) {
                this.c = c;
                this.count = count;
        }
        public void run() {
                for(int i = 0; I < count; i++) {
                        System.out.print(c);
                }
        }
}
```

- Om niet telkens zelf een nieuwe klasse te schrijven kan je ook threads maken via Lambda Expressions, dit bespaart heel wat regels code.
- De Thread klasse heeft 1 methode (**start()**) omdat Runnable maar 1 abstracte methode heeft (**run()**), daardoor is Runnable ook een functionele interface.
- Via een Lambda Expression kan je dus ter plekke een nieuwe thread aanmaken, wat veel makkelijker in gebruik is

  Thread thread1 = new Thread(() -> print("#",100));
  Thread thread2 = new Thread(() -> print("*", 100));
  thread1.start();
  thread2.start();

  Opmerking: de methode print is een aparte methode in dezelfde klasse die de meegegeven char x aantal keer afdrukt.

Levenscyclus van threads

- Een thread kan zich in verschillende toestanden bevinden
- Nadat men de methode start() heeft uitgevoerd gaat thread over in de toestand **RUNNABLE**
- Deze toestand heeft 2 sub toestanden, namelijk **READY** en **RUNNING**
  - **READY** = Threads die klaar staan om uitgevoerd te worden, de scheduler bepaalt welke thread wordt uitgevoerd.
  - **RUNNING** = De thread die uitgevoerd wordt, voor 1 CPU kan er maar 1 thread **RUNNING** zijn, de andere threads voor die CPU moeten wachten.
- Als alle taken zijn uitgevoerd in de thread zal deze zich in de toestand **TERMINATED** bevinden (nadat run() afgelopen is)
- De toestand van een thread kan je opvragen door de methode **getState()** er op uit te voeren.

Thread Prioriteit

- Threads in de toestand **READY** met een hogere prioriteit dan andere threads worden in de toestand **RUNNING** gebracht.
- Indien threads dezelde prioriteit hebben wordt er willekeurig eentje gekozen.
- Linux ondersteunt geen prioriteit voor threads.
- Je kan de prioriteit instellen via de methode **setPriority()**
  - **1 = minimum**
  - **5 = normal**
  - **10 = maximum**
- Threads die een andere thread gecreëerd worden (bv main()) krijgen dezelfde prioriteit als de hoofdthread mee.

Preëmptieve multitasking

- Dit komt simpelweg neer op het feit dat een actieve thread op ieder moment kan onderbroken worden als een nieuwe thread in de toestand **READY** komt en een hogere prioriteit heeft dan de huidige thread.
- De huidige thread gaat dan terug naar de toestand **READY**, de nieuwe thread komt in de toestand **RUNNING.**

Coöperatieve multitasking

- Threads met dezelfde prioriteit moeten wachten tot de huidige thread met dezelfde prioriteit klaar is.
- Het kan dus voorvallen dat sommige threads zeer lang moeten wachten eer ze aan de beurt zijn.
- Je kan threads laten samenwerken door regelmatig een thread even te laten rusten en een andere thread de kans te geven om zijn ding te doen. Dit doe je door de methode **yield()** uit te voeren op de thread.

  ```
  Thread thread1 = new Thread(() -> print("#",100));
  Thread thread2 = new Thread(() -> print("*", 100));
  thread1.start();
  thread2.start();

  public static void print(char c, int count) {
          for(int i = 0; i < count; i++) {
                  System.out.println(c);
                  Thread.yield();
          }
  }
  ```

  Na iedere iteratie geven we de andere thread de kans door de methode **yield()**.

  *##############**##########**############**###################**********

- Als je een programma maakt en je weet op voorhand niet hoeveel cores een machine ter beschikking heeft, kan je dit opvragen met:

  ```
  Runtime.getRuntime().availableProcessors();
  ```

Daemon Threads

- Onder normale omstandigheden kan de main() thread pas afgelopen zijn als alle subthreads hun taak vervuld hebben.
- Een **Daemon thread** is een thread die niet klaar hoeft te zijn om toch de volledige applicatie te kunnen sluiten.
- Dit doe je met de methode **setDaemon(true);**
- Als je alle thread onvoorwaardelijk wil beëindigd kan je dit met **System.exit();**

Wachttoestand

Slaaptoestand

- Een thread kan zichzelf in slap brengen met de methode **Thread.sleep();**
- Ze heeft altijd betrekking op de **huidige actieve thread**.
- Deze thread komt dan in de toestand **TIMED_WAITING** terecht
- De thread weer wakker maken doe je met de methode **interrupt()**
- Je moet deze methode wel een **try/catch** stoppen en de **InterruptedException** opvangen.
- Via **Thread.currentThread()** krijg je telkens de huidige thread terug.

Wachten op de beëindiging van een thread

- Als er meerdere threads actief zijn kan het nodig zijn dat een thread moet wachten op een andere thread.
- Een thread laten wachten op een andere thread kan je met de methode **join()**
- De huidige thread komt dan in de toestand **WAITING**
- Deze toestand wordt verlaten zodra de thread waarop men wacht klaar is of deze onderbroken wordt met de methode **interrupt()**

Synchronisatie

- In de vorige voorbeelden deedt elke thread zijn eigen ding, als meerdere threads dezelfde bronnen gebruiken of afhankelijk zijn van mekaar, moeten deze threads gesynchroniseerd worden.
- Als we bv een teller bijhouden via een aparte klasse Count (met methode increment en getCount) en 2 threads deze teller 1000 x laten verhogen, zal het uiteindelijke resultaat niet 20000 zijn.
    - We gebruiken trouwens ook join() om te wachten tot beide threads klaar zijn en dan drukken we het resultaat pas af
- De oorzaak is dat door **time slicing** een thread op eender welk moment kan onderbroken worden om een andere thread te activeren.
- De threads houden geen rekening met elkaars activiteiten.
    - Het kan dus dat de 1ste thread de tellerwaarde ophaalt en een andere thread op dat moment net hetzelfde doet
    - Beide threads verhogen de waarde en schrijven deze weg.
    - De teller is maar **1x** verhoofd en er is dus **1** verhoging verloren gegaan.
- Om te voorkomen dat meerdere threads tegelijkertijd een stuk code kunnen uitvoeren, kunnen we dit stuk code omringen door **synchronized(Object o) {…}**
- Aan synchronized moet een Object meegegeven worden, best maak je hier een apart object voor aan en noem je deze bv **key** of **lock**.
    - Je kan ook **this** gebruiken, op die manier gebruik je de huidige van de methode waar synchronized gebruikt.
- Het Object dient eigenlijk als een soort van sleutel, de thread die de sleute heeft kan het stuk code uitvoeren, een andere thread moet wachten tot hij de de sleutel krijgt van synchronized.
- Dit wachten is de zogenaamde Thread toestand **BLOCKED**

```
Object key = new Object();
public void increment() {
        synchronized(key) {
                count++;
        }
}
```

- Indien het stuk code een volledige methode is (zoals in dit geval) en het Object dat als monitor/key/lock dient, kunnen we nog korter aangeven:

```
public synchronized void increment(){
        count++;
}
```

- Als het stuk code een volledige statische methode is, is er geen impliciete **this** en wordt **this** bekeken als de instantie van de klasse.

Timer & TimerTask

- Deze klassen gebruik je om taken op een bepaald tijdstip uit te voeren of met een bepaald interval.
- De taak zelf is een object van de klasse **TimerTask** en geef je mee aan de constructor van **Timer**.
- TimerTask heeft een methode run die je zelf kan implementeren door bv een eigen klasse te schrijven die **extends TimerTask** toepast.
- Vervolgens schrijf je je code in de **public void run()** methode
- Via de **schedule()** methode kan je vervolgens het Timer object een interval meegeven

```
public class Timeout extends TimerTask {
        public void run() {
                System.out.println("De tijd is om!");
        }
}

public class Main {
        public static void main(String[] args) {
                Timeout task = new Timeout();
                Timer timer = new Timer(true);
                timer.schedule(task, 1000 * 10);
                System.out.println("Even geduld");
        }
}
```

Na 10 seconden wordt de taak uitgevoerd en verschijnt er **De tijd is om**!

Threads in grafische applicaties

- Swing applicaties gebruiken maar 1 thread om alles te handelen, de **event handling thread**.
- De grafische Swing componenenten zijn **niet thread-safe**, je maakt dus ook geen nieuwe threads aan in deze applicaties.
- De event handling thread is wel niet bedoeld langdurige activiteiten uit te voeren, maar enkel korte acties. Doe je dit wel, dan bevriest het scherm tot de thread terugkeert om weer grafische acties uit te voeren.
- Om toch langdurige activiteiten uit te voeren gebruikt Swing zijn eigen manier om enkel de event handling thread met grafische acties te belasten.
- We hebben bv een grafische applicatie die een image van een auto over het scherm laat bewegen, als we hier een Timer plaatsen om bv om x aantal milliseconden de positive van de auto opnieuw te tekenen, zal de grafische weergave gedurende de tijd van de timer bevriezen op het scherm.

```
Timer timer = new Timer(50, (p) -> moveCar());
```

- Om dit te voorkomen gebruiken we de klasse **SwingWorker** die het mogelijk maakt om berekeningen uit te voeren in een afzonderlijke thread zonder de event handling thread hiermee belast wordt.
- Deze klasse heeft enkele methoden
  - doInBackground();
  - get()
  - done()
  - execute()
- Je maakt van deze klasse een subklasse en implement **doInBackground()** waar de langdurige code in komt.
- Eventueel implementeer je ook de methode **done()**, deze gebruik als je op het einde van de berekening aanpassingen in de weergave te doen.

```
public class MyModel{
        public String longCalculation(){
                try{
                        Thread.sleep(5000);
                }
                catch(InterruptedException e){
                }
                return "Hello World";
        }
}

private class MyWorker extends SwingWorker<String,Object> {
        @Override
        protected String doInBackground() throws Exception {
                return model.longCalculation();
        }
        @Override
        protected void done(){
                try{
                        view.setResult(get());
                }
                catch(Exception e){
                        e.printStackTrace();
                }
        }
}
```

- Vervolgens ga je een object maken van MyWorker en kan je deze starten met de methode **execute()**
- De code in de methode **doInBackground()** wordt vervolgens uitgevoerd.
- Als deze klaar is wordt de **event handling thread** op de hoogte gebracht hier van en wordt de code in de methode **done()** uitgevoerd.

```
public void buttonClicked(){
        MyWorker worker = new MyWorker();
        worker.execute(0;
}
```

- Enkel pagina's terug staat de uitleg van **synchronized(Object o) { …}**
- Het gebruik hiervan voorkomt dat meerdere threads dezelfde bronnen gebruiken en dat er vervolgens foute berekening gemaakt worden
- Het **Concurrency Framework** maakt dit in principe nog makkelijker door een aantal Interfaces en Klassen aan te bieden.
    - Deze bevinden zich in **java.util.concurrent**
- De utility-klasse **Collections** biedt een aantal methoden aan die er voor zorgen dat je lijsten kan omvormen naar gesynchroniseerde versies daar van.
    - synchronizedCollection()
    - synchronizedList()
    - synchronizedNavigableMap()
    - synchronizedNavigableSet()
    - synchronizedSet()
    - synchronizedSortedMap()
    - synchronizedSortedSet()
- Deze versies zijn daardoor veilig in gebruik met meerdere threads en het is dan ook niet nodig om zelf **synchronized(Object o){ …}** toe te passen

- We hebben bv een klasse met een lijst van Integers, via een methode kunnen twee threads tegelijkertijd een item toevoegen aan de lijst.
- Zonder het gebruik van **synchronized** lopen deze threads elkaar constant voor de voeten en krijgen we soms Exceptions, is het aantal elementen in de lijst niet wat je verwacht, etc.
- We kunnen daarom een gesynchroniseerde versie van de lijst maken en deze simpelweg gebruiken

  List<Integer> originalList = new ArrayList<>();
  List<Integer> syncedList= Collections.synchronizedList(originalList);
  Thread thread1 = new Thread(() -> populate(syncedList, 100));
  Thread thread2 = new Thread(() -> populate(syncedList, 100));

  De twee threads gebruiken nu de gesynchroniseerde versie van de List en er treden daardoor geen problemen meer op.

- In java.util.concurrent zitten nog enkel gesynchroniseerde versies
    - ConcurrentHashMap
        - Gesynchroniseerde versie van HashMap
    - ConcurrentSkipListMap
        - Gesynchroniseerde versie van TreeMap
    - ConcurrentSkipListSet
        - Gesynchroniseerde versie van TreeSet
    - CopyOnWriteArrayList
        - Gesynchroniseerde versie van ArrayList
    - CopyOnWriteArraySet
        - Gesynchroniseerde versie van Set

Atomaire Objecten

- Als je terug denkt aan het voorbeeld waar we een teller willen verhogen, dan hebben we daar **synchronized** manueel gebruikt om er voor te zorgen dat de teller op de juiste manier werd verhoogd door beide threads.
- Deze teller was een Integer object.
- Er bestaan in het Concurrent Framework bestaan er speciale Wrapper klassen die **atomaire operaties** mogelijk maken, deze bevinden zich in het packet **java.util.concurrent.atomic**
    - AtomicBoolean
    - AtomicInteger
    - AtomicIntegerArray
    - AtomicLong
    - AtomicLongArray
    - AtomicReference
    - AtomicReferenceArray
- Als je deze Wrapper klassen gebruikt hoef je zelf geen **synchronized** meer toe passen op deze.

```
private AtomicInteger count = new AtomicInteger(0);
public void increment(){
        count.incrementAndGet();
}
public void decrement(){
        count.decrementAndGet();
}
public int getCount(){
        return count.get();
}
```

Callable/ExecutorService/Future

- Zelf een thread klasse maken en deze gebruiken hebben we enkele pagina's geleden gedaan.
- Het nadeel is wel dat je deze thread zelf moet starten en wachten op het resultaat.
- Om dit makkelijker te maken zijn er klassen en interfaces beschikbaar.
- Priemgetallen berekenen kan best lang kan duren en deze wil je dus ook uitvoeren in een aparte thread.
- Je maakt een klasse die **Callable<V>** implementeert, deze interface heeft 1 methode die **V Call()** heet.

```
    public class PrimCalculator implements Callable<List<Long>> {
        private long max;
        public PrimeCalculator(long max) {
            this.max = max;
        }
    }

    @Override
    public List<Long> call() throws Exception {
        List<Long> primes = new ArrayList<>();
        Boolean isPrime = true;
        for (Long number = 2; number <= max; number++) {

            for(int I = 0; I < primes.size() && isPrime && ((primes.get(i)primes.get(i)
        <= number); i++){
            isPrime = (number % primes.get(i)) != 0);
            }
            if(isPrime) {
                primes.add(number);
            }
        }
        return primes;
    }
}
```

- De klasse gaan we nu gebruiken in onze Main

```
PrimeCalculator pc = new PrimeCalculator(10000000);
ExecutorService es = Executors.newSingleThreadExecutor();
Future<List<Long>> future = es.submit(pc);

while (!future.isDone()) {
    System.out.println("Waiting");
}

List<Long> primes = future.get();

for (long p : primes){
    System.out.println(p);
}
es.shutdown();
```

- Taken die de interface **Callable** implementeren kunnen uitgevoerd worden door een object van **ExecutorService**
- Deze voer je vervolgens uit de met de methode **submit()**
- Het resultaat hiervan is een object van het type **Future** omdat het resultaat nog moet uitgevoerd worden door een andere thread.
- Je checken of de berekening klaar is met **isDone()** en als het klaar kan je het ophalen met **get()**.
- Op het einde moet je de ExecutorService afsluiten met shutdown();

Parallelisme

- In het hoofdstuk Collections hebben we gezien dat van lijsten een stream kan maken en hier op volgens kan consumeren, filteren, reduceren en collecteren.
- Je doet dus telkens een bewerking op elk element uit de stream, meestal met een Lambda Expression.
- De verwerking van die bewerking gebeurt door één thread die alle elementen één voor één behandelt.
- Maar het is ook mogelijk om dit door meerdere threads te laten doen en dan gebruik je een **parallel stream**

  Collection.ParallelStream();

  bv

  ```
  List<String> words = new ArrayList();
  words.parallelStream().forEach(System.out::println);
  ```

- Het gebruik hier van is niet noodzakelijk sneller want het aanmaken van een nieuwe thread en het opsplitsen van de bewerkingen zorgt voor extra werk.
- Bij hele grote lijsten kan dit wel voordelig zijn!