HO
GENT

Collections

Table of Contents

1. Doelstellingen

- Kan de juiste generieke datastructuur uit het collections framework kiezen
- Kan de juiste generieke datastructuur uit het collections framework correct gebruiken
- Kan iterators gebruiken om een collectie te doorlopen
- Kan meerdere methodes uit de Collections klasse toepassen om collecties te verwerken

2. Inleiding

In dit hoofdstuk gaan we een blik werpen op het Java Collections Framework. Bij software ontwikkeling komen collections, losweg vertaald 'groepen van objecten', 'verzamelingen', veelvuldig voor. Stel je bijvoorbeeld software voor een simpel kaartspel voor. De stapel kaarten is een collection van kaart-objecten, de verschillende spelers vormen een collection van Speler-objecten, en de kaarten in die bij elke speler horen zijn ook collections van Kaart-objecten. Als je denkt aan een webshop kan je ook verschillende collections voorstellen. De webshop biedt *producten* aan, beheert zijn *klanten*, *bestellingen*, ...

Collections en Collections Framework

Een **collection** is een datastructuur, i.e. een object, die een verzameling van objecten als 1 geheel representeert. De objecten in de verzameling noemen we de **elementen**.

Een **collections framework** biedt een waaier van interfaces, abstracte en concrete klassen aan die toelaten op een flexibele manier verzamelingen van objecten te beheren en te manipuleren.

Door gebruik te maken van wat een collections framework aanreikt hoef je niet alle details van de interne representatie van collections te kennen of zelf te implementeren, en hoef je ook niet zelf algoritmes te implementeren om bijvoorbeeld specifieke elementen in een verzameling te vinden, of je verzameling te sorteren.

Het is een uitdaging voor de ontwikkelaar om vat te krijgen op dit framework. Verschillende collections hebben immers verschillende eigenschappen. Wil je een verzameling waarin je dubbels kan opslaan? Wil je een verzameling waarin de elementen gesorteerd zitten? Is het belangrijk dat je in je verzameling vliegensvlug elementen kunt opzoeken, of heeft het snel toevoegen en verwijderen van elementen een hoge prioriteit voor je? Dergelijke eigenschappen kunnen mede bepalen met welke collection uit het framework je aan de slag gaat. Met kennis van het collections framework zal je in staat zijn op een gefundeerde manier de meest geschikte collection te kiezen voor een taak en eveneens op een gepaste manier gebruik te maken van alle functionaliteit die het framework aanbiedt om performante software op een elegante en flexibele manier te ontwikkelen.

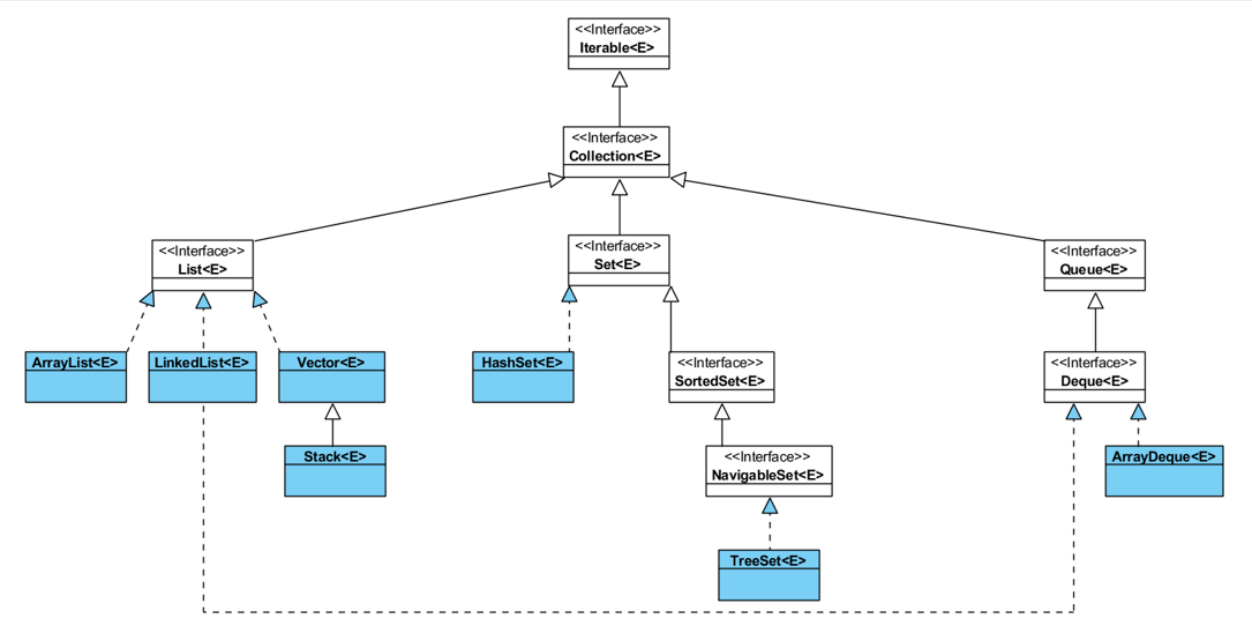
In dit hoofdstuk gaan we slechts een deel van het omvangrijke Java Collections Framework toelichten. Naarmate je je verder specialiseert in software ontwikkeling zal je ongetwijfeld nog meer over dit framework leren. De zaken die je over het Java Collections Framework leert zullen ook een goede basis vormen om collection frameworks in andere programmeertalen te leren

begrijpen. Iemand die vertrouwd is met het Java Collections Framework zal bijvoorbeeld snel vat krijgen op het C# collections framework, en omgekeerd…

- het collections framework is onderdeel deel van de ***java.util package***
- het collections framework is ***generiek*** opgezet
  - de interfaces en klassen hebben een geparametriseerd type en bevatten dus herbruikbare code
- elementen in een collection zijn ***referenties naar objecten***
  - dikwijls zeggen we kortweg dat de collection objecten bevat

3. Overzicht

Klassen en interfaces uit het Java Collections Framework



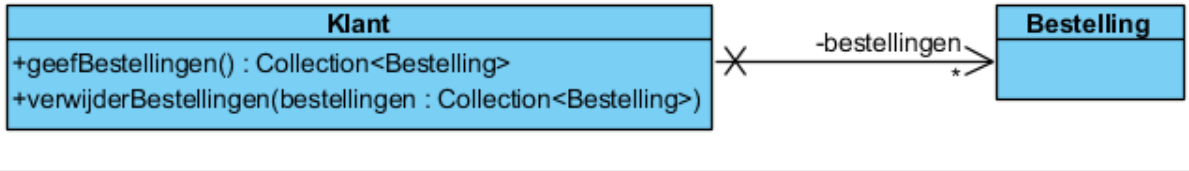
*deze afbeelding is beperkt tot **klassen en interfaces** die we in dit hoofdstuk gaan behandelen,*
*de **concrete collection klassen** staan in het blauw aangegeven*

4. Interface Collection<E>

De interface **Collection<E>** vormt de basis, de root van het collection framework. Alle andere interfaces en concrete klassen die we in dit hoofdstuk bespreken stammen af van deze interface. Er zijn trouwens geen directe concrete afstammelingen van deze interface, er zijn enkel andere interfaces, sub-interfaces, die deze interface 'extend-en' (=uitbreiden).

Het universele karakter van deze interface maakt dit een prima kandidaat om als type te fungeren bij het doorgeven van collections aan methodes, i.e. als type voor parameters, of bij het retourneren van collections als het resultaat van een methode, i.e. als returntype van een methode. Op deze manier kan een klasse op een vrij **algemene manier functionaliteiten aanbieden terwijl de**

implementatiedetails van de collection verborgen blijven voor de client code (dit zijn de andere klassen die van de aangeboden functionaliteit gebruik maken). De concrete implementatie van de collection zit op deze manier ingekapseld en kan desgewenst gewijzigd worden zonder dat de client code hiervoor hoeft aangepast te worden.



- de klasse Klant retourneert alle bestellingen van een klant als een Collection<Bestelling>, de concrete collection klasse die de bestellingen beheert, blijft verborgen voor de client code en kan desgewenst gewijzigd worden zonder dat dit invloed heeft op de cliënt code

- aan de klasse Klant kan gevraagd worden om een aantal bestellingen te verwijderen, de cliënt heeft de vrijheid om te kiezen hoe deze verzameling concreet zal aangeleverd worden

- merk op hoe de generische parameter <E> in Collection<**E**> hier een invulling kreeg met het type **Bestelling**

De interface Collection<E> bevat enkele belangrijke 'algemene' methodes. De op deze manier aangeboden functionaliteit laat bijvoorbeeld toe om het aantal elementen die in een verzameling zitten op te vragen, na te gaan of de verzameling al dan niet leeg is, of ze een bepaald element al dan niet bevat, en laten eveneens toe om een element toe te voegen en te verwijderen...

Collection<E> methodes

```
int size()
boolean isEmpty()
boolean contains(Object o)
boolean add(E element)
boolean remove(Object element)
Iterator<E> iterator()
```

Zoals hierboven reeds werd aangehaald werken collections met objecten. Indien we gegevens van een primitief datatype in collections willen bijhouden zullen we moeten gebruik maken van de **primitieve wrapper klassen** om dergelijke collections te declareren. Via de **auto-boxing en auto-unboxing** features van Java zal je daar verder weinig extra voor moeten doen.

De **contains**-methode, die retourneert of een element al dan niet aanwezig is in een collection maakt gebruik van de **equals** methode om op gelijkheid te testen. Dit is dus een belangrijk gegeven als we met het collection framework aan de slag gaan. We moeten zorgen dat deze, en dan uiteraard ook de hashcode-methode, een goede invulling heeft gekregen die de correcte gelijkheid van elementen uitdrukt. We hebben deze methodes uitvoerig besproken in het hoofdstuk 'Overerving deel 2'.

Er zijn ook enkele methodes voor bulk operaties aanwezig in Collection<E>. Deze methodes werken

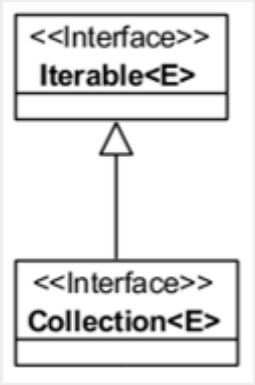
met volledige collections in plaats van afzonderlijke elementen. De duidelijke namen van de methodes laten je reeds raden wat onderstaande methodes doen maar vergeet niet de Java API te raadplegen om in meer detail te begrijpen wat de methodes doen vooraleer je ermee aan de slag gaat.

Collection<E> 'bulk' methodes

```
boolean containsAll(Collection<?> c)
boolean addAll(Collection<? extends E> c)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
```

*- Collection<**?**> duidt op een collection van eender welk type.*
*- Collection<**? extends E**> duidt op een collection van een type dat een subtype van E is.*

5. Interface Iterable<E>



Elke collection erft van de interface Iterable<E>. Concreet betekent dit dat we een mechanisme hebben om over de elementen van eender welke collection te itereren.

- elke collection kan overlopen worden met een **enhanced for-loop** of een **forEach()**
  - *tijdens dergelijke iteratie mag de collection **niet gewijzigd worden***
- elke collection kan ook overlopen worden met behulp van een **iterator**
  - *tijdens dergelijke iteratie kunnen eventueel elementen uit de collection verwijderd worden*

Een iterator is een object. Door methodes aan te roepen op een iterator kan je een collectie overlopen. Via de volgende methode uit Collection<E> kan je dit object opvragen:

```
Iterator<E> iterator()
```

Via de onderstaande methodes uit de interface Iterator<E> kan je nu doorheen de collection navigeren en eventueel elementen uit de collection verwijderen. Daar een iterator *een concrete implementatie* is van de interface Iterator<E> heb je ook de flexibilteit om zelf een iterator te implementeren.

```
boolean hasNext() // retourneert true als de iteratie nog elementen bevat
E next()          // retourneert volgende element van de iteratie
void remove()     // verwijdert het laatste element dat werd geretourneerd
                  // door deze iterator uit de onderliggende collection
```

Merk op dat door de **eenvoudige en compacte syntax van de enhanced for-loop** je waarschijnlijk niet expliciet een iterator zal gebruiken om gewoon een collection te overlopen. De enhanced for-loop zal dan je voorkeur genieten.

De **forEach-methode itereert impliciet** over de collection en voert een actie uit op elk element van de collection gedurende deze iteratie. De actie die moet worden uitgevoerd op elk element moet de functionele interface Consumer<T> implementeren en kan dus op een eenvoudige wijze doorgegeven worden als een **lambda expressie** of een **method reference**. De functionele interface Consumer<T> bevat volgende abstracte methode:

```
void accept(T t) // abstracte methode in Interface Consumer<T>
```

Enkele voorbeelden:

```
    public void iterateWithEnhancedFor(Collection<String> colors) {
        for (String color : colors) {
            System.out.printf("%s ", color);
        }
    }

    public void iterateWithForEach(Collection<String> colors) {
        colors.forEach(c -> System.out.printf("%s ", c));
    }

    public void iterateWithIterator(Collection<String> colors) {
        Iterator<String> iterator = colors.iterator();
        while (iterator.hasNext()) {
            System.out.printf("%s ", iterator.next());
        }
    }


    public void useIteratorAndRemoveElements(Collection<String> colors) {
        Iterator<String> iterator = colors.iterator();
        while (iterator.hasNext()) {
            String nextColor = iterator.next();
            if (nextColor.length() > 4)
                iterator.remove();
        }
        System.out.printf("The collection contains %d colors...", colors.size());
    }

    public void useSomeBasicCollectionOperators(Collection<String> colors) {
        System.out.printf("This collection is%s empty%n", colors.isEmpty() ? "" : "
not");
        String myColor = "red";
        System.out.printf("Colors does %scontain %s%n", colors.contains(myColor) ? ""
: "not ", myColor);
        colors.remove(myColor);
        System.out.printf("Colors does %scontain %s%n", colors.contains(myColor) ? ""
: "not ", myColor);
        myColor = "yellow";
        System.out.printf("Colors does %scontain %s%n", colors.contains(myColor) ? ""
: "not ", myColor);
        colors.add(myColor);
        System.out.printf("Colors does %scontain %s%n", colors.contains(myColor) ? ""
: "not ", myColor);
    }
```

Bekijk hieronder de uitvoer van een aanroep naar deze methodes.
Het argument ***colors is een collection die de strings "red", "green" & "blue" bevat...***

```
    iterateWithEnhancedFor(colors);
    // red green blue

    iterateWithForEach(colors);
    // red green blue

    iterateWithIterator(colors);
    // red green blue

    useIteratorAndRemoveElements(colors);
    // The collection contains 2 colors...

    useSomeBasicCollectionOperators(colors);
    // This collection is not empty
    // Colors does contain red
    // Colors does not contain red
    // Colors does not contain yellow
    // Colors does contain yellow
```

In volgend voorbeeld wordt het verschil van twee collections genomen.

```
    private void removeColorsWithIterator(Collection<String> collection1, Collection
<String> collection2) {
        Iterator<String> iterator = collection1.iterator();
        while (iterator.hasNext())
            if (collection2.contains(iterator.next()))
                iterator.remove();
    }

    private void removeColorsWithCollectionMethodRemoveAll(Collection<String>
collection1,
            Collection<String> collection2) {
        collection1.removeAll(collection2);
    }
```

We dalen nu verder af in het collections framework. Er zijn **drie belangrijke interfaces** die erven van Collection<E>. Elk van deze drie heeft zijn **specifieke eigenschappen** en ze vormen dan ook de basis voor vrij verschillende soorten verzamelingen.

List<E> - Set<E> - Queue<E>

6. Interface List<E>

Een List is een **geordende collection** (ook wel **sequence** genoemd). Elk element in een List heeft een **exacte positie** in de verzameling. Deze positie noemen we ook wel de **index** en ze is **zero-based** (i.e. start van 0). Een gebruiker van deze interface heeft dus precieze controle over de plaats waar een element in de verzameling geplaatst wordt, of moet opgehaald worden. In een List zijn **dubbels toegelaten**.

Hieronder vind je enkele methodes, die bij uitbreiding van de Collection<E> interface, te vinden zijn in List<E>. Merk op hoe bij elke methode gebruik wordt gemaakt van de positie van een element, de **index**.

enkele List<E> methodes

```
E get(int index)
int indexOf(Object o)
int lastIndexOf(Object o)
E remove(int index)
E set(int index, E element)
```

Deze interface bevat ook enkele handige methodes om de List<E> te sorteren en sub-lists te nemen. Via de toArray-methode is het mogelijk een List om te zetten naar een array. De mogelijkheid om een collection om te zetten naar een array vinden we ook in andere interfaces terug en zullen we verderop nog bespreken.

```
    List<E> subList(int fromIndex, int toIndex)
    void sort(Comparator<? super E> c)
    <T> T[] toArray(T[] a)
```

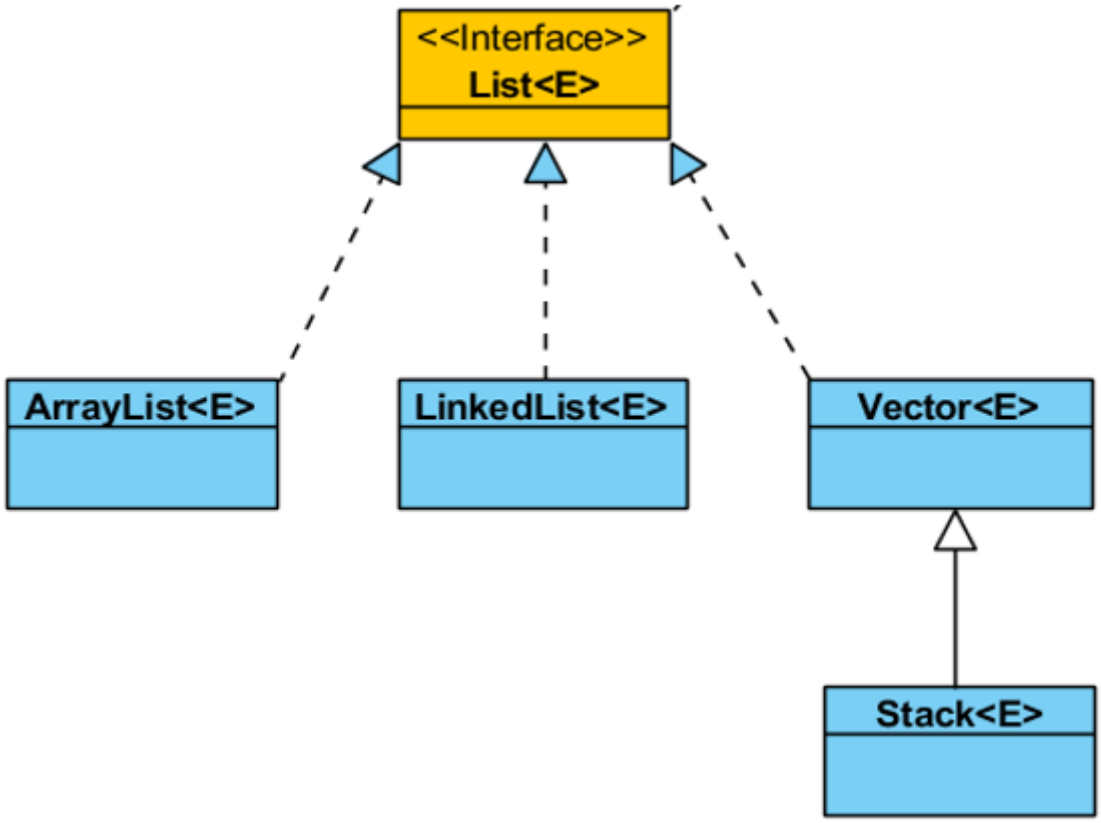
Bij een List<E> kan je gebruik maken van de **ListIterator** om over de elementen van de List te itereren. Deze ListIterator is krachtiger dan de Iterator waarvan Collection<E> een sub-interface is. Je kan de ListIterator opvragen via één van de volgende methodes.

```
    ListIterator<E> listIterator()
    ListIterator<E> listIterator(int index)   // iterator die start op de aangegeven
index
```

Zoals je merkt aan de methodes hieronder kan een ListIterator over de List itereren in **twee richtingen** en laat ze toe om tijdens het itereren ook **elementen toe te voegen, te verwijderen en/of te wijzigen**.

```
    void add(E e)
    boolean hasNext() & boolean hasPrevious()
    E next() & E previous()
    int nextIndex() & int previousIndex()
    void remove()
    void set(E e)
```

We zijn nog geen concrete klassen uit het collection framework tegengekomen. Daar komt nu verandering in. Hieronder zie je in het blauw enkele **concrete klassen** die List<E> implementeren.



Programmeer steeds naar een interface!

Nu we de overstap maken naar concrete klassen voor onze verzamelingen is het belangrijk om dit principe steeds in gedachten te houden. Er is bijna nooit een reden om als type van je verzameling een concrete klasse te verkiezen boven een interface...

```
ArrayList<E> mijnVerzameling = new ArrayList<E>()  // DO NOT!
```

```
List<E> mijnVerzameling = new ArrayList<E>()      // DO!
```

Door gebruik te maken van de interface List<E> als type voor je verzameling ben je vrij één van de concrete klassen te instantiëren voor die verzameling. Je behoudt op deze manier de vrijheid om de gekozen concrete implementatie te wijzigen zonder dat dit een grote impact heeft op je code. Je kan bovendien op deze manier beter gebruik maken van methodes uit de Java API die vaker een interface type zoals List<E> zullen retourneren dan een concreet type zoals ArrayList<E>. Zo kan je bijvoorbeeld het resultaat van de methode subList, een methode uit de klasse ArrayList, niet toekennen aan een verzameling van het type ArrayList, maar wel aan een verzameling van het type List...

```
    List<E> subList(int fromIndex, int toIndex)
```

Naast een parameterloze constructor, zoals hierboven werd gebruikt, kan je ook gebruik maken van een constructor waar je een collection kan aan doorgeven.

```
ArrayList(Collection<? extends E> c)
```

Deze constructor maakt een nieuwe ArrayList en vult ze op met alle elementen uit de aangereikte collection. Dergelijke constructor noemt men een **conversion constructor** en is beschikbaar in alle concrete klassen die we in dit hoofdstuk behandelen. Dit is belangrijk want via deze constructor kan je dus **gelijk welk type collection uit het framework omvormen tot een ander type**.

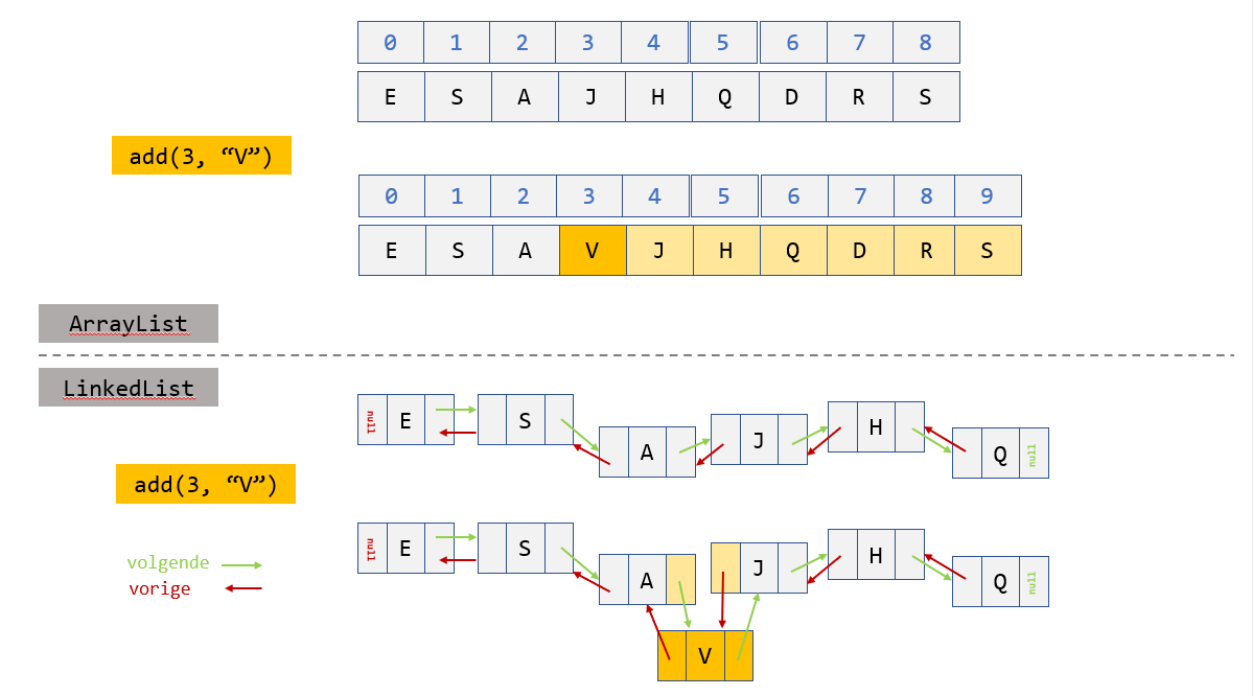
6.1. Klasse ArrayList<E>

Het is nuttig om enige kennis te hebben over de concrete implementatie van een collection. De ArrayList<E> maakt onderliggend gebruik van een **resizeable array implementatie**. De elementen worden dus opgeslagen in een array, die bij creatie van een nieuwe lijst een initiële capaciteit heeft. Daar een array een fixed size structuur is en een lijst kan groeien moet er echter soms op zoek gegaan worden naar een groter stuk geheugen waar een grotere array kan gehuisvest worden. Dit gebeurt wanneer er tijdens het toevoegen van een element aan de lijst geen genoeg plaats meer blijkt te zijn in de onderliggende array. Dit neemt wat tijd in beslag en weegt negatief op de performantie van de ArrayList implementatie. Hoewel de initiële capaciteit en het groeien van de array volledig automatisch gebeurt voorziet deze klasse wel in methodes en een constructor die toelaten dit te manipuleren. Doordacht gebruik van deze methodes kan de performantie ten goede komen. De methodes vind je hieronder maar we zullen er verder niet mee aan de slag gaan in dit hoofdstuk. De *int*-parameter bij de methodes stelt het aantal elementen voor die de onderliggende array moet kunnen huisvesten.

```
ArrayList(int initialCapacity)
void trimToSize()
void ensureCapacity(int minCapacity)
```

6.2. Klasse LinkedList<E>

Bij een LinkedList<E> worden de elementen in de lijst aaneengeregen via referenties. Concreet heeft elk element in een **doubly-linked list** implementatie een referentie naar zijn voorganger, en naar zijn opvolger. Wanneer er gebruik wordt gemaakt van een index om een bepaald element uit de lijst op te halen dan moet de lijst vanaf het eerste element doorlopen worden, en moet telkens de referentie naar het volgende element gevolgd worden, tot het element op de aangegeven index wordt bereikt. Dit betekent dan ook dat dergelijke operatie trager verloopt dan bij een ArrayList. Anderzijds moeten bij deze implementatie nooit elementen verschoven worden als we ergens in de lijst een element willen tussenvoegen. Referenties naar voorganger en opvolger kunnen eenvoudig en snel aangepast worden. Bij een ArrayList daarentegen betekent dergelijke operatie dat er, mogelijks veel, elementen in de onderliggende array moeten verschoven worden.



- bij een insert in een array moeten meerder elementen verschoven worden.

- bij een insert in een doubly linked list moeten enkel een paar referenties aangepast worden.

In onderstaande tabel zijn enkele belangrijke verschillen tussen ArrayList<E> en LinkedList<E> samengevat.

ArrayList	**LinkedList**
elementen in resizeable array	elementen in doubly linked list
constante toegangstijd voor elk element ⇒ random access	sequentiële toegang ⇒ starten vanaf eerste element
invoegen of verwijderen van element ⇒ veel verschuivingen	efficiënt toevoegen en/of verwijderen van element
bij voorkeur te gebruiken bij veel **opzoekingen**	bij voorkeur te gebruiken bij **veel invoegen/verwijderen** van elementen

6.3. Klasse Vector<E>

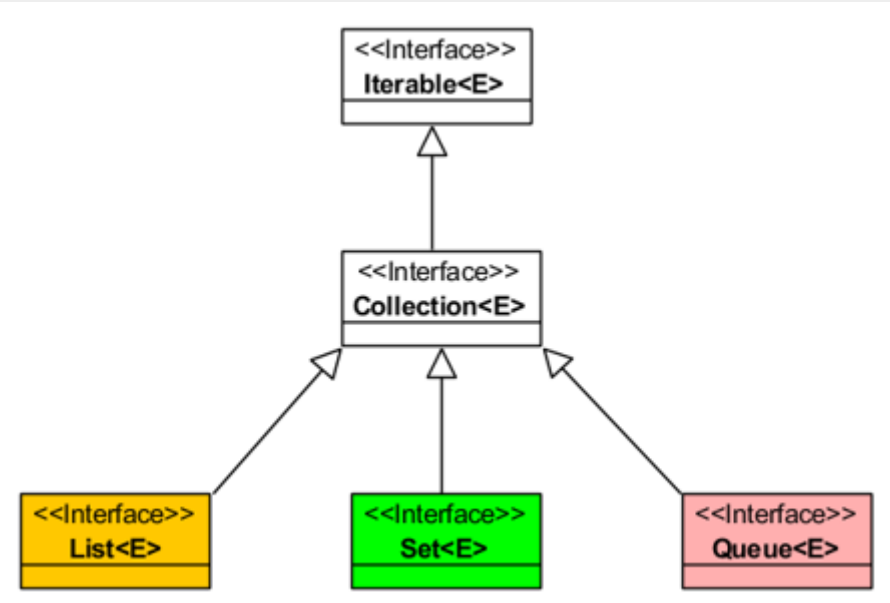
De klasse Vector<E> is een verouderde klasse. Er wordt aangeraden gebruik te maken van ArrayList<E> in plaats van Vector<E>. Een verschil tussen beide is wel dat Vector<E> gesynchroniseerd is en je dus een thread-safe implementatie hebt. Dit kan een reden zijn om te kiezen voor Vector<E> in multi-threaded code. Deze concepten worden behandeld in Advanced Software Development I.

6.4. Klasse Stack<E>

De klasse Stack<E> is een subklasse van Vector<E>. We zullen deze dan ook niet verder behandelen. Verderop zullen we zien hoe stacks beter kunnen geïmplementeerd worden adhv ArrayDeque<E>.

In een volgende sectie nemen we een tweede belangrijke sub-interface van Collection<E> onder de loep…

List<E> - Set<E> - Queue<E>



7. Interface Queue<E>

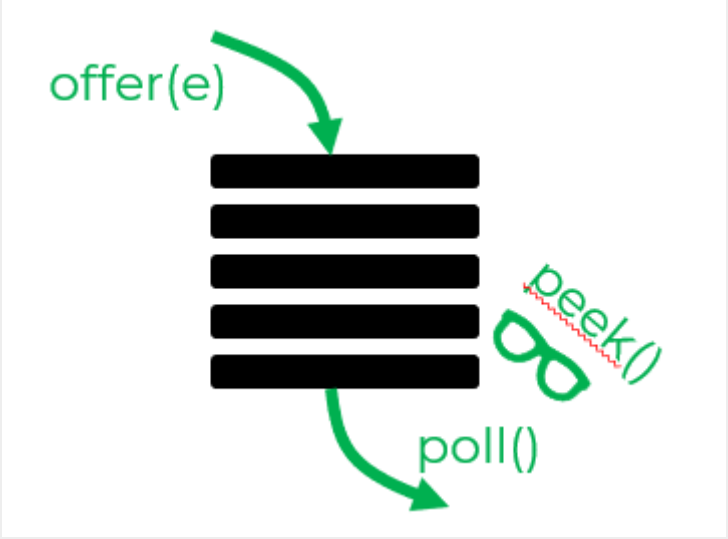
Een Queue<E> wordt typisch gebruikt om elementen bij te houden alvorens ze te verwerken. Een queue is een FIFO structuur, ook wel een wachtrij genoemd. De elementen in een queue zijn niet te benaderen via hun positie.

FIFO - First In First Out

enkele Queue<E> methodes

```
boolean offer(E e)
E peek()
E poll()
```

FIFO: offer, poll & peek



In de Queue<E> interface vind je ook de methodes add, remove & element. Deze methodes vormen het equivalent van offer, poll & peek maar werpen exceptions daar waar offer, peek & poll speciale waarden retourneren. Zo zal de poll-methode null retourneren wanneer de queue geen elementen bevat, terwijl in dezelfde situatie de methode remove een exception zal werpen. Raadpleeg de Java API voor meer details.

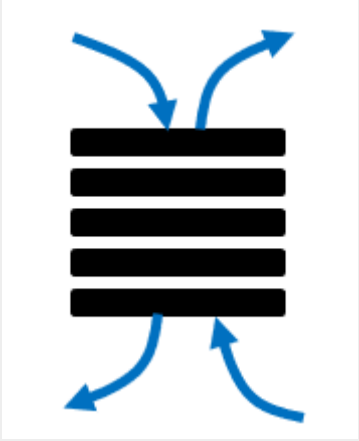
Een voorbeeld:

```
        Queue<Double> queue = new ArrayDeque<>();

        // insert elements to queue
        queue.offer(3.2);
        queue.offer(9.8);
        queue.offer(5.4);

        System.out.print("Polling from queue: ");
        // display elements in queue
        while (queue.size() > 0) {
            double element = queue.poll(); // verwijdert & retourneert het top element
            System.out.printf("%.1f ", element);
        } // end while
```

7.1. Interface Deque<E>

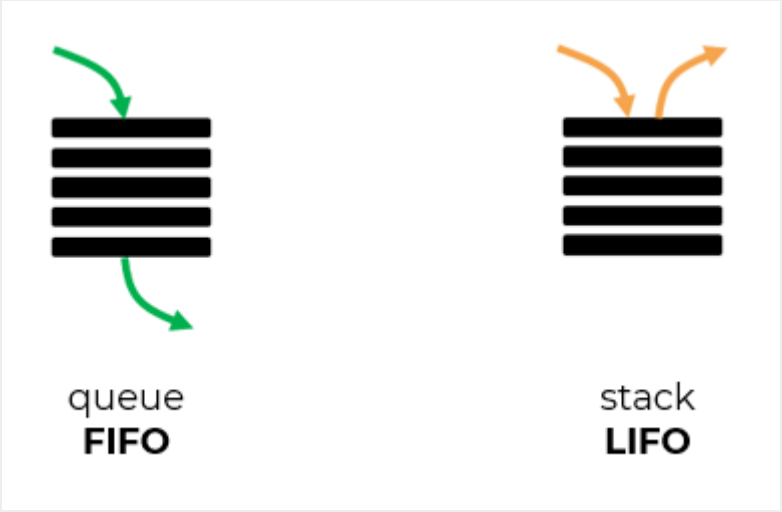
Deque<E> is een sub-interface van Queue<E> die een **double ended queue** voorstelt. Het toevoegen en verwijderen van elementen kan dus aan beide uiteinden gebeuren, we noemen deze uiteinden de **head** en de **tail**..

Deque



Dit maakt Deque<E> een ideale interface om zowel stacks als queues voor te stellen.

FIFO & LIFO

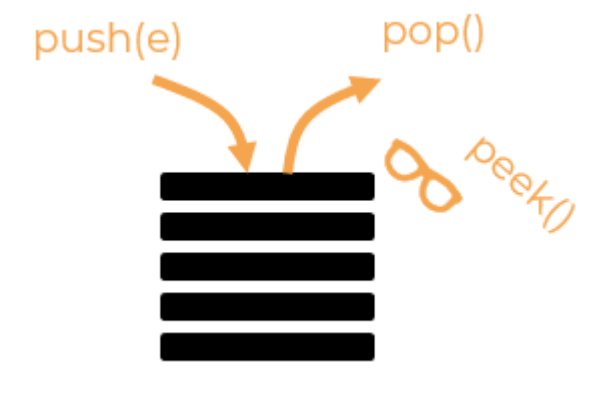


LIFO - Last In First Out

de Stack<E> methodes zitten ook in Deque<E>...

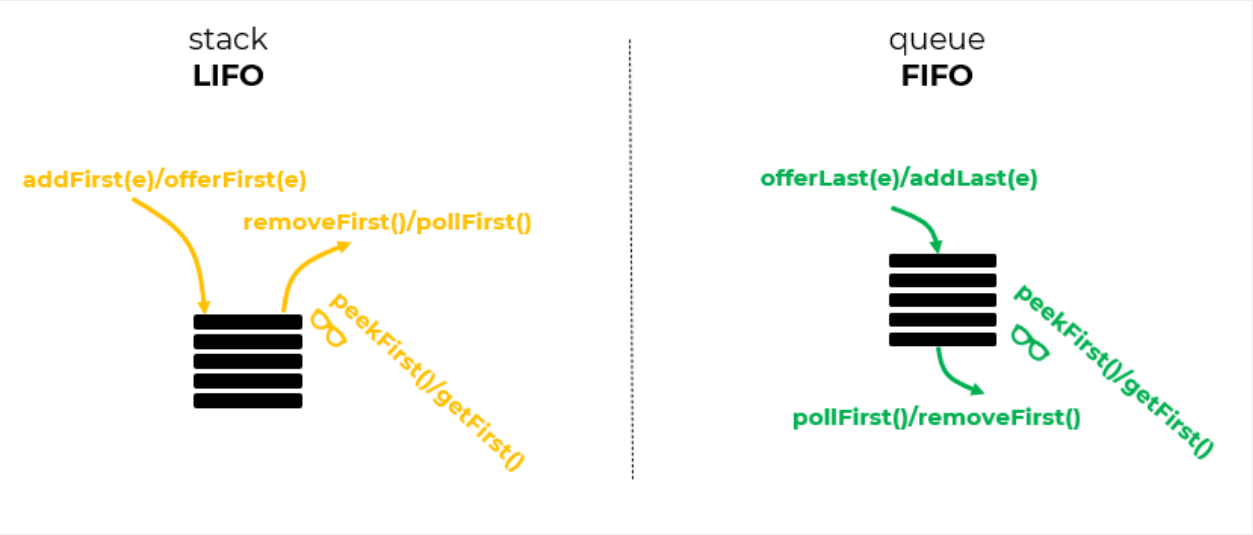
```
void push(E e)
E peek()
E pop()
```

Deque<E> voor LIFO



De interface Deque<E> kan tamelijk overweldigend zijn daar ze een aantal methodes bevat die gelijke of gelijkaardige functionaliteit aanbieden. Zo zijn naast de push/pop methodes die werden overgenomen uit Stack<E> ook andere typische Deque<E> methodes beschikbaar. Op onderstaande figuur zie je enkele alternatieve methodes uit Deque<E> om queues en stacks te beheren. Je ziet telkens twee methodes voor eenzelfde functionaliteit. Ze verschillen in de manier waarop ze exceptions werpen of speciale waarden retourneren. Even verderop vind je in de tabel met het overzicht deze details terug.

meer Deque<E> methodes voor LIFO & FIFO



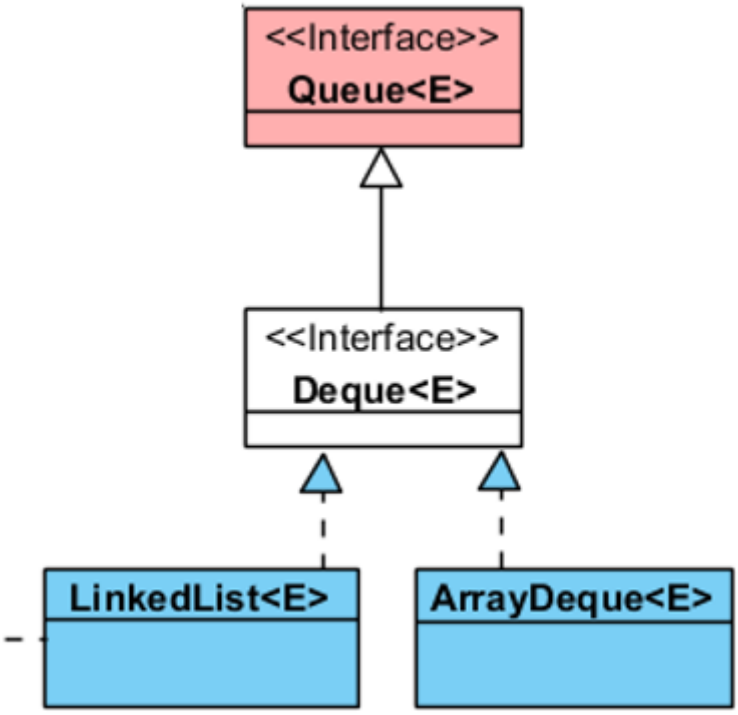
Je mag *offer/poll* methodes gebruiken samen met *add/remove* methodes maar waarschijnlijk wil je dit niet doen want het zal de leesbaarheid en onderhoudbaarheid van je code niet ten goede komen.

Onderstaande samenvatting uit de Java API geeft een mooi overzicht van alle **head** en **tail** methodes.

overzicht head & tail methodes

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

7.2. Klasse ArrayDeque<E>



Zoals de naam reeds suggereert is dit een resizeable array implementatie van de Deque<E> interface. Deze klasse zal performanter zijn dan de klasse Stack<E> voor een stack, of LinkedList<E> voor stacks en/of queues. Merk op dat de klasse LinkedList<E> hierboven reeds uitvoerig werd besproken. De klasse implementeert dus zowel de List<E> als de Deque<E> interface.

In volgend voorbeeld wordt een stack geïmplementeerd aan de hand van een ArrayDeque. Op de stack worden verschillende soorten getallen gezet. De elementen zijn van het type **Number**, dit is een superklasse van de primitieve wrapper klassen Integer, Long, Double en Float.

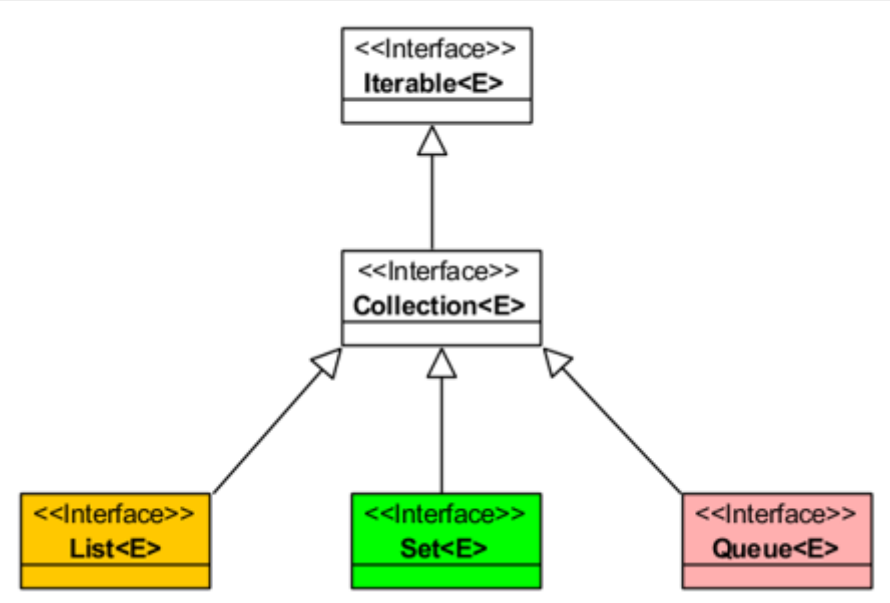
```
    public StackTest()
    {
        Deque<Number> stack = new ArrayDeque<>();
        Long longNumber = 12L;
        Integer intNumber = 34567;
        Float floatNumber = 1.0F;
        Double doubleNumber = 1234.5678;

        stack.push(longNumber);
        printStack(stack);
        stack.push(intNumber);
        printStack(stack);
        stack.push(floatNumber);
        printStack(stack);
        stack.push(doubleNumber);
        printStack(stack);

        try {
            Number removedObject = null;
            while (true)
            {
                removedObject = stack.pop();
                System.out.printf("%s popped%n", removedObject);
                printStack(stack);
            } // end while
        } // end try
        catch (NoSuchElementException noSuchElementException) {
            noSuchElementException.printStackTrace();
        } // end catch
    } // end StackTest constructor

    private void printStack(Deque<Number> stack)
    {
        if (stack.isEmpty())
            System.out.print("stack is empty\n\n"); // de stack is leeg
        else // de stack is niet leeg
        {
            System.out.print("stack contains: ");
            for (Number number : stack)
                System.out.printf("%s ", number);

            System.out.print("(top) \n\n");
        } // end else
    } // end method printStack
```

Hieronder zie je de uitvoer:

```
stack contains: 12 (top)

stack contains: 34567 12 (top)

stack contains: 1.0 34567 12 (top)

stack contains: 1234.5678 1.0 34567 12 (top)

1234.5678 popped
stack contains: 1.0 34567 12 (top)

1.0 popped
stack contains: 34567 12 (top)

34567 popped
stack contains: 12 (top)

12 popped
stack is empty

java.util.NoSuchElementException
    at java.base/java.util.ArrayDeque.removeFirst(ArrayDeque.java:362)
    at java.base/java.util.ArrayDeque.pop(ArrayDeque.java:593)
    at ui.StackTest.<init>(StackTest.java:31)
    at main.StartUp.main(StartUp.java:10)
```

Tijd om Set<E> onder de loep te nemen, een derde belangrijke sub-interface van Collection<E>...

List<E> - Set<E> - Queue<E>



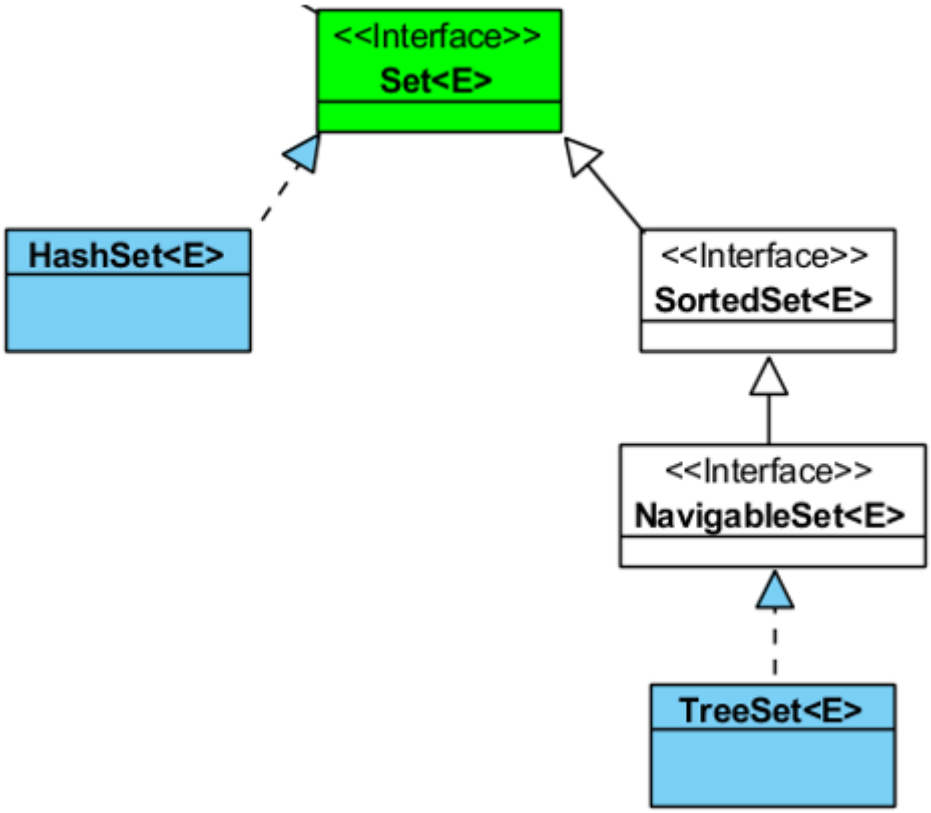
8. Interface Set<E>

De Set<E> interface benadert het best de notie die we hebben van het wiskundig begrip verzameling. Elementen in dergelijke verzamelingen zijn niet toegankelijk via hun positie, ze hebben **geen index**. Bovendien is het typisch voor dergelijke verzamelingen dat ze **geen dubbels** mogen bevatten. We benadrukken nogmaals dat deze gelijkheid gebaseerd is op de **equals** -methode. Mogelijks kunnen elementen in een set wel geordend zijn.

Een set bevat nooit twee elementen e1 en e2 waarvoor e1.equals(e2) true is.

In deze interface zijn er naast de methodes die geërfd worden uit de super-interface Collection<E> geen extra methodes gedefinieerd.

We dalen nu verder af in de hiërarchie en bespreken enkele sub-interfaces en concrete Set<E> implementaties.



8.1. Klasse HashSet<E>

HashSet<E> is een concrete implementatie van Set<E> die een hashtabel als onderliggende structuur heeft en waarbij elementen niet geordend zijn. Dieper ingaan op deze structuur valt buiten de scope van deze cursus maar het is handig om weten dat in dergelijke structuur de **hashcode** van een object wordt gebruikt als een index in een tabel. Dit betekent concreet dat elementen in deze structuur enorm snel te vinden zijn en dat ook het toevoegen en verwijderen van elementen heel performant kan gebeuren. Wat betreft het itereren over de elementen van de verzameling zullen andere collections dikwijls een betere performantie geven. Daar de elementen niet geordend zijn kan je trouwens ook niet voorspellen in welke volgorde de elementen

geretourneerd worden wanneer je over een HashSet itereert.

Een voorbeeld:

```
    private void printNonDuplicates(Collection<String> collection) {
        // een HashSet creëren
        Set<String> set = new HashSet<>(collection);

        System.out.println("\nNonduplicates are: ");
        for (String s : set)
            System.out.printf("%s ", s);
        System.out.println();
    }
```

8.2. Interface SortedSet<E>

Zoals uit de naam af te leiden valt is een SortedSet<E> een set met een **ordening van de elementen**. De ordening van de elementen kan op natuurlijke of op totale wijze zijn. Het ordenen van objecten werd uitvoerig behandeld in het hoofdstuk "Polymorfisme en Interfaces". Door de ordening van de elementen wordt het itereren over een SortedSet wel voorspelbaar: elementen worden van klein naar groot geretourneerd door de iterator.

- natuurlijk ordening maakt gebruik van de **compareTo**-methode
  - *de elementen in de set moeten Comparable<E> implementeren*
  - *deze ordening wordt gebruikt indien je geen comparator specifieert tijdens constructie van concrete SortedSet<E> implementaties*
- totale ordening maakt gebruik van een **comparator**
  - *er moet een comparator die Comparator<E> implementeert voorzien worden*
  - *de comparator kan aan de constructor van concrete SortedSet<E> implementaties meegegeven worden*

Het feit dat de elementen gesorteerd in deze verzameling zitten geeft dan ook aanleiding tot meer specifieke functionaliteit die van de volgorde van de elementen gebruik maakt:

enkele SortedSet<E> methodes

```
E first()
E last()
SortedSet<E> subset(E fromElement, E toElement)
SortedSet<E> headset(E toElement)
SortedSet<E> tailset(E fromElement)
```

8.3. Interface NavigableSet<E>

Een NavigableSet<E> heeft een extra notie van nabijheid. Dit vertaalt zich in volgende methodes.

enkele NavigableSet<E> methodes

```
    E floor(E e)    // het grootste element <= e (of null indien geen element <=
e)
    E lower(E e)    // het grootste element < e (of null indien geen element < e)
    E ceiling(E e)  // het kleinste element >= e (of null indien geen element >=
e)
    E higher(E e)   // het kleinste element > e (of null indien geen element > e)
```

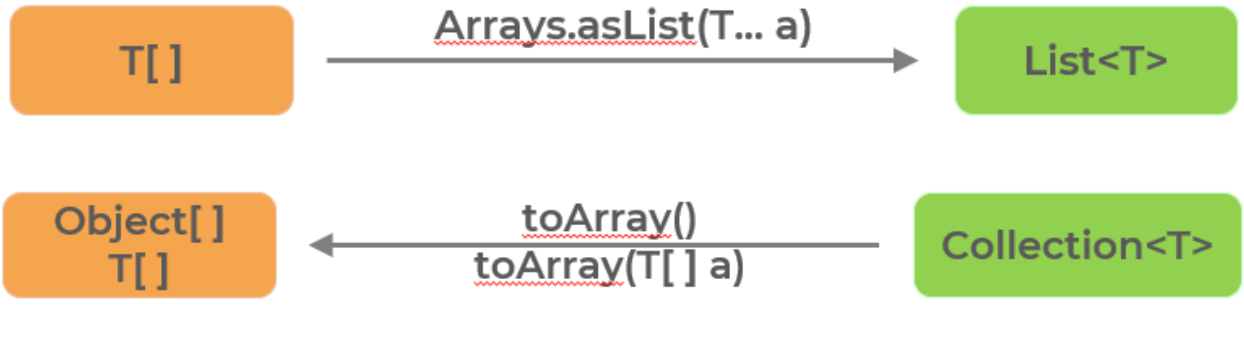
8.4. Klasse TreeSet<E>

Deze concrete implementatie van NavigableSet<E> is gebaseerd op een boomstructuur. We gaan op deze onderliggende structuur ook niet verder ingaan in dit hoofdstuk. Het is goed om weten dat ook deze implementatie performant opzoeken, toevoegen en verwijderen van elementen garandeert.

Een voorbeeld:

```
    private static final String NAMES[] = { "yellow", "green", "black", "tan", "grey",
"white", "orange", "red","green" };

    public SortedSetTest() {
        SortedSet<String> tree = new TreeSet<>(Arrays.asList(NAMES));
        System.out.println("sorted set: ");
        printSet(tree);

        // alle elementen die < zijn dan element "orange"
        System.out.print("\nheadSet (\"orange\"):  ");
        printSet(tree.headSet("orange"));

        // alle elementen die >= zijn dan element "orange"
        System.out.print("tailSet (\"orange\"):  ");
        printSet(tree.tailSet("orange"));

        // het eerste en het laatste element
        System.out.printf("first: %s%n", tree.first());
        System.out.printf("last : %s%n", tree.last());
    }


    private void printSet(SortedSet<String> set) {
        for (String s : set)
            System.out.printf("%s ", s);
        System.out.println();
    }
```

9. Een brug tussen arrays en collections



9.1. Van array naar collection

De **klasse Arrays** uit de java.util package is geen onderdeel van het collections framework. De klasse bevat heel wat interessante methodes om in arrays te zoeken, om arrays te sorteren, om arrays of delen ervan te kopiëren, enz. Het kan dan ook soms handig zijn om een collection te kunnen omvormen naar een array, en ook omgekeerd. Op dergelijke manier wordt een brug gelegd tussen array-based API's en collection-based API's.

van ARRAY naar LIST

```
static <T> List<T> asList(T... a)
```

- **T...**: deze zogenaamde **varargs** constructie laat toe de methode aan te roepen met een **willekeurig aantal argumenten van type T**. De argumenten worden allemaal verzameld in een array (de parameter 'a') waarvan dan binnen de methode gebruik kan gemaakt worden. Het is ook mogelijk een T[] argument op te geven. In dit geval bevat de parameter a gewoon de aangereikte array.

  Varargs is gelijkaardig aan de JavaScript rest parameter.

Een belangrijke opmerking bij deze conversie is dat de resulterende lijst een **fixed-sized** lijst is die de **oorspronkelijke array als onderliggende structuur** heeft. Dit betekent dan ook dat de veranderingen die je doorvoert in de verkregen lijst zullen gereflecteerd worden in de oorspronkelijke array, en omgekeerd!

Je kan een nieuwe List creëren met het resultaat van asList om dit te vermijden

```
List<String> fixedList = Arrays.asList("Arrays", "are", "fun");
fixedList.add("!"); // throws an exception

List<String> realList = new ArrayList<>(Arrays.asList("Arrays", "are", "fun"));
realList.add("!"); // is ok...
```

Door gebruik te maken van **conversion constructors** ben je dus in staat om een array om te vormen naar eender welke collection.

9.2. Van collection naar array

De **Collection<E> interface** biedt de mogelijkheid om eender welke collection om te vormen naar een array.

van COLLECTION naar ARRAY

```
Object[] toArray()

<T> T[] toArray(T[] a)
```

Als je de methode toArray aanroept op een collection dan is het resultaat een array die **alle elementen** uit de collection bevat. Wanneer bij gebruik van de getypeerde versie van deze methode, met een parameter T[], blijkt dat het array argument niet groot genoeg is om alle

elementen uit de collection te bevatten zal een nieuwe array, die voldoende groot is, gecreëerd worden. Indien de doorgegeven array meer ruimte heeft dan nodig is om alle elementen uit de collection te bevatten, dan zal in de eerste vrije ruimte in de array de waarde null geplaatst worden.

Indien je de methode aanroept met als argument **new T[0]** ben je zeker van een resulterende array die exact groot genoeg is om alle elementen van de collection te bevatten. Zo hoef je niet expliciet de lengte van de oorspronkelijke collectie op te geven.

```
Deque<Integer> stapel = new ArrayDeque<>(Arrays.asList(10, 20, 30, 40));
Integer[] arrayVanStapel = stapel.toArray(new Integer[0]);
System.out.println(arrayVanStapel.length);  // 4
```

```
String[] mijnArray = { "Hearts", "Diamonds", "Clubs", "Spades" };

// Array => Collection
List<String> mijnLijst = new ArrayList<>(Arrays.asList(mijnArray));

// Collection => Array
String[] mijnArray =  mijnLijst.toArray(new String[mijnLijst.size()]);
```

10. Klasse Collections

In deze klasse vind je heel wat static methods om collections te manipuleren. De algoritmes maken gebruik van polymorfisme wat betekent dat de meeste algoritmes op een waaier van collections toepasbaar zijn. De meeste van de methodes in deze klasse werken met de List<E> interface.

Enkele algoritmes die de klasse beschikbaar stelt:

- **sorteren** van List collections
- **shuffelen** van List collections
- routine data manipulatie
  - **omkeren, vullen, kopiëren** van List
  - **swappen van elementen** in een List
  - **toevoegen van meerdere elementen in 1 keer** aan een Collection
- **zoeken** naar elementen in een gesorteerde List via binarySearch

Enkele voorbeelden:

```
    private static final String SUITS[] = { "Hearts", "Diamonds", "Clubs", "Spades" };

    public void start() {
        List<String> list = Arrays.asList(SUITS);

        Collections.sort(list); ①
        System.out.printf("Sorted array elements:%n%s%n", list);
        for (String i : SUITS)
            System.out.printf("%s%n", i);

        Collections.sort(list, Collections.reverseOrder()); ②

        Collections.shuffle(list); // schud door elkaar     ③
    }
```

① Sorteervolgorde wordt bepaald door de implementatie van de bij het object horende **compareTo** methode. Wordt gedeclareerd in de interface Comparable, i.e. de **natuurlijke ordening**.

② Een andere sorteervolgorde verkrijg je door een tweede argument mee te geven, een Comparator object.

- Voor gedefinieerde comparator objecten zoals Collections.reverseOrder()
- Zelf gedefinieerde comparator objecten.

③ Verdeelt de elementen van een list in een willekeurige volgorde.

Voorbeeld 2:

```
    private Character[] LETTERS = { 'P', 'C', 'M' }, lettersCopy;
    private List<Character> copyList;

    public void start() {
        List<Character> list2 = Arrays.asList(LETTERS);

        lettersCopy = new Character[3];
        copyList = Arrays.asList(lettersCopy);

        System.out.println("Initial list: ");
        output(list2);

        Collections.reverse(list2); // reverse order ①
        System.out.println("\nAfter calling reverse: ");
        output(list2);
        Collections.copy(copyList, list2); // copy List ②
        Collections.fill(list2, 'R'); // fill list with R□s ③
    }

    private void output(List<Character> listRef) {
        System.out.print("The list is: ");
        for (Character elem : listRef)
            System.out.printf("%s ", elem);

        System.out.printf("%nMax: %s", Collections.max(listRef)); ④
        System.out.printf("  Min: %s%n", Collections.min(listRef)); ⑤
    }
```

① Keert de volgorde van de elementen in list2 om

② De inhoud van list2 wordt gekopieerd naar copyList

③ list2 wordt opgevuld met de letter R

④ Het "grootste" element wordt uit listRef opgehaald(= hier: hoogste unicode)

⑤ Het "kleinste" element wordt uit listRef opgehaald(= hier: laagste unicode)

Voorbeeld 3:

```
    private static final String COLORS[] = { "red", "white", "yellow", "green", "pink"
};
    private List<String> list;

    public CollectionsMethods3() {
        list = new ArrayList<>(Arrays.asList(COLORS));
        Collections.sort(list); // sort the ArrayList
        System.out.printf("Sorted ArrayList: %s%n", list);

        int result = Collections.binarySearch(list, "yellow"); ①
        System.out.printf("yellow: %d%n", result);

        result = Collections.binarySearch(list, "purple"); ②
        System.out.printf("purple: %d%n", result);

        String[] colors = { "red", "white", "yellow"};
        List< String > list = Arrays.asList( colors );

        List< String> arrayList = new ArrayList<>();
        arrayList.add( "red" );
        arrayList.add( "green" );

        arrayList.addAll(list);

        int frequency = Collections.frequency( arrayList, "red" ); ③
        System.out.printf("%n%nFrequency of red in arrayList: %d%n",frequency);



        // hebben list en arrayList al dan niet gemeenschappelijke
        // elementen:
        boolean disjoint = Collections.disjoint( list, arrayList); ④
        System.out.printf( "%nlist and arrayList %s elements in common%n",
                ( disjoint ? "do not have" : "have"  ) );
```

① yellow: 4

② purple: -3
negatieve index: purple komt niet voor in de lijst, dit resultaat wordt op volgende manier berekend:
-(indexwaarde_in_geval_van_invoegen) - 1
daar purple op index 2 in de lijst zou terechtkomen krijgen we -3

③ Telt het aantal voorkomens van "red" in de arrayList

④ Hebben list en arrayList gemeenschappelijke elementen?

11. Tot slot

Hier beëindigen we onze exploratie van het **Java Collections Framework**. Naarmate je meer en

meer met collections gaat werken en je je weg in de Java API vlotjes vindt zal je de flexibiliteit en de kracht van dit framework ook beter appreciëren en ten volle kunnen toepassen. Het verhaal over collections heeft ook een vervolg in een volgend hoofdstuk waarin we **Java streams** bespreken. **Collections** en **lambda expressies** die we in het gelijknamig hoofdstuk behandelden vormen de toegangspoort tot de mooie wereld van functioneel programmeren…