HoGent
BEDRIJF
EN
ORGANISATIE

Hoofdstuk 16 :

GENERIC COLLECTIONS

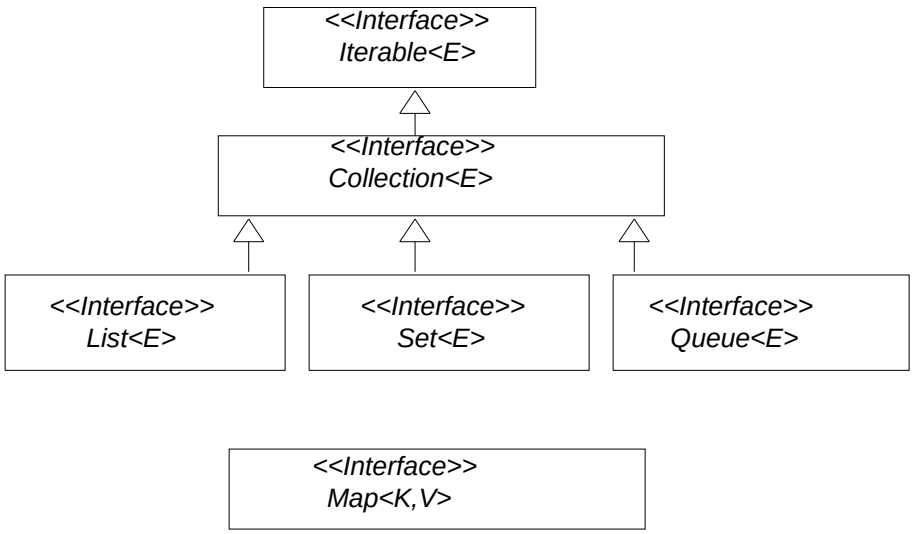
1. INLEIDING

- Collections framework
  - Meest voorkomende datastructuren gestandaardiseerd en efficient geïmplementeerd.
  - Goed voorbeeld van herbruikbare code

2. OVERZICHT COLLECTIONS FRAMEWORK

- **Collection**
  - Datastructuur (object) die referenties naar andere objecten bijhoudt.
- **Collections framework**
  - Interfaces die de operaties declareren voor verschillende collection types en verschillende implementaties (classes) daarvan.
  - Behoren tot het package `java.util`
    - `Collection<E>`
    - `Set<E>`
    - `List<E>`
    - `Map<K,V>`
    - `Queue<E>`

2. OVERZICHT COLLECTIONS FRAMEWORK



2. OVERZICHT COLLECTIONS FRAMEWORK

- Interface Iterator<E>
  - boolean hasNext()
  - E next()
  - void remove() //optional
- Interface ListIterator<E> extends Iterator<E>
  - void add(E o) //optional
  - boolean hasPrevious()
  - E previous()
  - int nextIndex()
  - int previousIndex()
  - void set(E o) //optional

HoGent

2. OVERZICHT COLLECTIONS FRAMEWORK

- Extra tools
  - Klasse Arrays → methode asList + allerlei bewerkingen op arrays.
  - Klasse Collections → allerlei bewerkingen op collecties.

3. CLASS ARRAYS

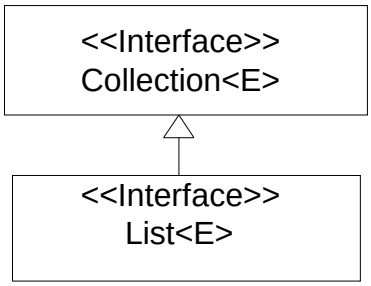
- **Zie OOPII – H7_Deel2**

4. INTERFACE COLLECTION EN KLASSE COLLECTIONS

- **Interface `Collection<E>`**
  - Bevat *bulk operations* (op de volledige container)
    - "Adding", "clearing", "comparing" en "retaining objects".
  - Interfaces `Set<E>`, `List<E>` en Queue<E> extends interface `Collection<E>`
  - Voorziet een `Iterator<E>:` dient om alle elementen van een collection te doorlopen. De Iterator kan ook elementen verwijderen.

- **Klasse `Collections` (→ zie paragraaf 7)**
  - Voorziet `static` methoden die collections manipuleren
  - Polymorfisme wordt hierbij ondersteund

5. LIST INTERFACE



ArrayList<E>, **LinkedList<E>** en **Vector<E>** zijn implementatie klassen van interface List.

5. LIST INTERFACE

- List<E>
  - Geordende Collection waarbij duplicaten toegelaten zijn: SEQUENCE.
  - List index start ook van nul.
  - Implementatie klassen:
    - ArrayList<E> (resizable-array implementatie)
    - LinkedList<E> (linked-list implementatie)
    - Vector<E> (zoals ArrayList maar synchronized)

5.1 Voorbeeld : ARRAYLIST

```
import java.util.*;
public class CollectionTest
{
  private static final String[ ] COLORS =
                { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
  private static final String[ ] REMOVECOLORS = { "RED", "WHITE", "BLUE" };

  public CollectionTest()
  {
    List< String > list = new ArrayList< >();

    // opvullen van de Arraylist “list”
    for ( String color : COLORS )
       list.add( color );

    List< String > removeList =
                    new ArrayList< >( Arrays.asList(REMOVECOLORS) );

     // OF ( maar minder efficient, zoals bij list)
     // List< String > removeList = new ArrayList< >();
     // for ( String color : REMOVECOLORS )
     //       removeList.add( color );
```

5.1 Voorbeeld : ARRAYLIST

```
    // afdrukken
    // ----------
    System.out.println( "ArrayList: " );
    printList(list);

    // verwijder alle strings uit "list" die in "removeList"
    // voorkomen
    removeColors( list, removeList );

    //opnieuw afdrukken
    System.out.println( "\n\nArrayList after calling
                                        removeColors: " );
    printList(list);

  } // end CollectionTest constructor
```

5.1 Voorbeeld : ARRAYLIST

```
public void printList(Collection< String > collection)
{
  for ( String color : collection )
          System.out.printf( "%s ", color );
  System.out.println();

      // OF
      // Iterator< String > iterator = list.iterator();
      //  while ( iterator.hasNext() )
      //        System.out.printf( "%s ", iterator.next() );
      // GEEN for-lus met teller
}
```

5.1 Voorbeeld : ARRAYLIST

```
// verwijder alle strings uit "collection1" die in
// "collection2" voorkomen
private void removeColors( Collection< String > collection1,
                           Collection< String > collection2 )
{
   // get iterator
   Iterator< String > iterator = collection1.iterator();

   while ( iterator.hasNext() )
      if ( collection2.contains( iterator.next() ) )
         iterator.remove(); // remove String object

} // end method removeColors

public static void main( String args[] )
{
   new CollectionTest();
} // end main

} // end class CollectionTest
```

Bulk operaties

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

Oefening : bulk operaties en arrayList

(OefFruit1_opgave.java)

String arX[] = {"appel", "peer", "citroen", "kiwi"},
arY[] = {"banaan", "mango", "citroen", "kiwi", "zespri"}

Behandel arX en arY als Collections en maak gebruik van de bulk operaties om volgende output te leveren:

In y zit extra [banaan, mango, zespri]

In x zit extra [appel, peer]

x en y hebben gemeenschappelijk [citroen, kiwi]

Oefening: bulk operaties en arrayList

Constructor Summary: ArrayList:

- **ArrayList(Collection<? extends E> c)**
  Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Interface Collection:

boolean removeAll(Collection<?> c)
Removes all this collection's elements that are also contained in the specified collection (optional operation).

boolean retainAll(Collection<?> c)
Retains only the elements in this collection that are contained in the specified collection (optional operation).

HoGent

5.2 Voorbeeld : LINKEDLIST

```
import java.util.*;
public class ListTest
{
 private static final String COLORS[ ] = { "black", "yellow",
    "green", "blue", "violet", "silver" };
 private static final String COLORS2[ ] = { "gold", "white",
    "brown", "blue", "gray", "silver" };

  // aanmaak en manipulatie van LinkedList objecten
 public ListTest()
 {
   List< String > list1 = new LinkedList< >();

   // opvullen van de LinkedList "list1"
   for ( String color : COLORS )
      list1.add( color );

   // opvullen list2 via constructor: efficienter
   List< String > list2 =
         new LinkedList< >(Arrays.asList(COLORS2));
```

HoGent

5.2 Voorbeeld : LINKEDLIST

```
    // 'plak' link2 achter link1
    list1.addAll( list2 ); // concatenatie van lists
    link2 = null; // release resources

    printCollection(list1);

    convertToUppercaseStrings(list1);
    printCollection(list1);

    System.out.print( "\nDeleting elements 4 to 6..." );
    removeItems(list1, 4, 7 );
    printCollection(list1);

    printReversedList(list1);
  } // einde constructor ListTest
```


5.2 Voorbeeld : LINKEDLIST

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

/* OF
public void printCollection (Collection< String > col)
{   Iterator< String > iterator = col.iterator();
    while ( iterator.hasNext() )
        System.out.printf("%s ", iterator.next() );
    System.out.println();
}
*/
```

5.2 Voorbeeld : LINKEDLIST

```
// localiseer String objecten en converteer naar
// hoofdletters
private void convertToUppercaseStrings( List<String> list )
{
   ListIterator< String > iterator = list.listIterator();

   while ( iterator.hasNext() )
   {
      String color = iterator.next();  // get item
      iterator.set( color.toUpperCase() );
   } // end while

}
```



```
// gebruik sublist view om een reeks element te verwijderen
//van start t.e.m. end-1
private void removeItems( List< String > list, int start, int end )
{
   list.subList( start, end ).clear(); // verwijder items
}

// druk de lijst van eind naar begin
private void printReversedList( List< String > list )
{
   ListIterator< String > iterator = list.listIterator( list.size() );

   System.out.println( "\nReversed List:" );

   // print list in reverse order
   while ( iterator.hasPrevious() )
      System.out.printf( "%s ", iterator.previous() );
}

public static void main( String args[] )
   { new ListTest(); }
} // end class ListTest
```

5.2 Voorbeeld : LINKEDLIST

```
list:
black yellow green blue violet silver gold white brown blue gray silver

list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER

Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```


5.2 Voorbeeld : LINKEDLIST

'TE VERMIJDEN'

```
import java.util.*;
public class UsingToArray
{
  // create LinkedList, add elements and convert to array
  public UsingToArray()
  {
    String COLORS[] = { "black", "blue", "yellow" };

    LinkedList< String > links =
        new LinkedList< >( Arrays.asList( COLORS ) );

    links.addLast( "red" );   // add als laaste item

    links.add( "pink" );      // add als laaste item

    links.add( 3, "green" );  // insert op de 3de index plaats

    links.addFirst( "cyan" ); // add als eerste item
```

5.2 Voorbeeld : LINKEDLIST

```
// get LinkedList elements als een array
  colors = links.toArray( new String[ links.size() ] );
// ...
```

Levert volgende lijst: cyan, black, blue, yellow, green, red, pink

Argument van <T> T[] toArray(T[] a)
- als a te klein is : er wordt een nieuwe array van hetzelfde runtime type aangemaakt.
- als a groter is: de meegegeven array wordt gebruikt, de overige originele elementen blijven behouden behalve het element op plaats a[size] wordt null.

List Implementaties

Extra methoden Interface List:
void **add**(int index, E element); boolean **addAll**(Collection<? extends E> c);
E **get**(int index); int **indexOf**(Object o); **lastIndexOf**(Object o);
ListIterator<E> **listIterator**(); ListIterator<E> **listIterator**(int index); E **remove**(int index);
E **set**(int index, E element); List<E> **subList**(int fromIndex, int toIndex);

Extra methoden LinkedList:
LinkedList(); **LinkedList**(Collection<? extends E> c);void **addFirst**(E o);
void **addLast**(E o); Object **clone**(); E **getFirst**(); E **getLast**(); E **removeFirst**();
E **removeLast**()

Extra methoden ArrayList:
ArrayList(); **ArrayList**(int initialCapacity); **ArrayList**(Collection<? extends E> c);
Object **clone**(); void **ensureCapacity**(int minCapacity); void **trimToSize**();

Oefening: List Implementaties

Er zijn een aantal kisten met fruit... (OefFruit2_opgave.java)

• Voeg de verschillende kisten samen in een ArrayList list.

• Verwijder alle fruit dat met de letter 'p' begint uit de list. Gebruik hiervoor een eigen klasse CollectionOperaties met een methode verwijderOpLetter.

• Verwijder alle fruit uit de list vanaf het eerste voorkomen van kiwi tot het laatste voorkomen van kiwi, gebruik eveneens een methode verwijderSequence uit je klasse CollectionOperaties.

• Plaats het resultaat terug in een array mand en sorteer die oplopend.

```
String kist[][] = { {"appel", "peer", "citroen", "kiwi", "perzik"},
                    {"banaan", "mango", "citroen", "kiwi", "zespri", "pruim"},
                    {"peche", "lichi", "kriek", "kers", "papaya"} };
List <String> list;
String mand[];
```

5.3 Klasse Vector – klasse ArrayList

De klasse **Vector<E>** is een verouderde klasse en wordt vervangen door de nieuwe klasse ArrayList<E>.

Klasse **ArrayList<E>** bevat bijna dezelfde methodes als de klasse Vector<E>.
Een groot verschil is dat ArrayList<E> **niet gesynchroniseerd** is en een Vector<E> wel.

5.4 SYNCHRONIZED COLLECTION

- De collections van het Collections framework zijn unsynchronized.

- Via synchronization wrapper klasse converteren we collections tot synchronized versies.

  public static Collections methoden:
  - <T> Collection<T> synchronizedCollection (Collection<T> c)
  - <T> List<T> synchronizedList(List<T> aList)
  - <T> Set<T> synchronizedSet(Set<T> s)
  - <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
  - <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
  - <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)

  Vb: List<String> list =
  Collections.synchronizedList(new ArrayList<>());

6. ALGORITME: public class Collections extends Object

- Het Collections framework voorziet een aantal algoritmen (static methoden)
  - **List algoritmen:**
    - sort
    - binarySearch
    - reverse
    - shuffle
    - fill
    - Copy
  - **Collection algoritmen:**
    - min
    - max
    - addAll
    - frequency
    - disjoint

HoGent

6.1. ALGORITME : sort

- Sorteervolgorde : wordt bepaald door de implementatie van de bij het object horende compareTo methode. Wordt gedeclareerd in de interface **Comparable**. Deze sorteervolgorde wordt de 'natuurlijke ordening' genoemd.
- De sorteermethode is 'stable', bij gelijke waarden blijft de oorspronkelijke volgorde behouden.
- Een andere sorteervolgorde verkrijg je door een tweede argument mee te geven : dit is een **Comparator** object.
  - Voorgedefinieerde comparator objecten.
    - Collections.reverseOrder()
  - Zelfgedefinieerde comparator objecten.

HoGent

6.1. ALGORITME : sort

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

- **Volgens natuurlijke ordening: Comparable:**

```
List<String> list = Arrays.asList( SUITS );//create List
Collections.sort( list );

// output list
System.out.printf( "Sorted array elements:%n%s%n", list );
```

```
Sorted array elements:
[Clubs, Diamonds, Hearts, Spades]
```

```
for (String i:SUITS)
    System.out.printf( "%s%n",i );
```

```
Clubs
Diamonds
Hearts
Spades
```

HoGent

6.1. ALGORITME : sort

- **Met voorgedefinieerde Comparator (dalend):**

```
List< String > list = Arrays.asList( SUITS ); // create List
Collections.sort( list, Collections.reverseOrder() );

System.out.printf( "Sorted list elements:%n%s%n", list );
```

```
Sorted list elements:
[Spades, Hearts, Diamonds, Clubs]
```

- **Met zelfgedefinieerde Comparator:**
  vb: een ArrayList van Time2 objecten.

```
public class Time2  //zie eerste jaar.
{    private int hour, minute, second;
     //constructors, accessors, mutators, toString ...
}
```

6.1. ALGORITME : sort : zelfgedefinieerde Comparator

```
import java.util.*;
public class Sort3
{
  public void printElements()
  {
    List< Time2 > list = new ArrayList< >(); // create List
    list.add( new Time2(  6, 24, 34 ) );
    list.add( new Time2( 18, 14, 58 ) );
    list.add( new Time2(  6, 05, 34 ) );
    list.add( new Time2( 12, 14, 58 ) );
    list.add( new Time2(  6, 24, 22 ) );

    // output List elements
    System.out.printf( "Unsorted array elements:%n%s%n", list );
    // sort in order using a comparator
    Collections.sort( list, new TimeComparator() );
    // output List elements
    System.out.printf( "Sorted list elements:%n%s%n", list );
  }
   public static void main( String args[] )
  { new Sort3().printElements();
  }
```

6.1. ALGORITME : sort : zelfgedefinieerde Comparator

```
public class TimeComparator implements Comparator< Time2 >
{

    // >0 ➔ time1>time2   <0 ➔ time1 < time2    =0 ➔ time1= time2
    public int compare( Time2 time1, Time2 time2 )
   {
        int hourCompare = time1.getHour() - time2.getHour();
        if ( hourCompare != 0 ) // test het uur eerst
            return hourCompare;

        int minuteCompare = time1.getMinute() - time2.getMinute();
        if ( minuteCompare != 0 ) // dan de minuten
            return minuteCompare;

        int secondCompare = time1.getSecond() - time2.getSecond();
        return secondCompare;  // tenslotte de seconden
   }

}
```

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

6.2 ALGORITME : shuffle

- **Algoritme shuffle.**

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

```
class Card
{
        public static enum Face { Ace, Deuce, Three, Four, Five, Six,
                        Seven, Eight, Nine, Ten, Jack, Queen, King  };
        public static enum Suit { Clubs, Diamonds, Hearts, Spades };

        private final Face face; // face of card
        private final Suit suit; // suit of card

        public Card( Face cardFace, Suit cardSuit )
        {
         face = cardFace;
         suit = cardSuit;
        }

        public Face getFace() { return face; }
        public Suit getSuit() { return suit; }

        public String toString()
        {
           return String.format( "%s of %s", face, suit );
        }
}
```

```
public class DeckOfCards
{
  private List< Card > list; //list zal de speelkaarten bevatten

  // maak stapel kaarten en schud door elkaar
  public DeckOfCards()
  {
    Card[] deck = new Card[ 52 ];
    int count = 0; // number of cards

    // populate deck with Card objects
    for ( Card.Suit suit : Card.Suit.values() )
    {
      for ( Card.Face face : Card.Face.values() )
      {
        deck[ count ] = new Card( face, suit );
        count++;
      }
    }

    list = Arrays.asList( deck ); // get List
    Collections.shuffle( list );  // schud door elkaar
  } // end DeckOfCards constructor
```

```
  // toon de 52 speelkaarten in twee kolommen
  public void printCards()
  {  int i = 0;

      for ( Card kaart : list )
        System.out.printf( "%-20s%s", kaart,
          ( ( ++i) % 2 == 0 ) ? "%n" : "" );
  } // einde methode printCards


  public static void main( String args[] )
  {
     DeckOfCards cards = new DeckOfCards();
     cards.printCards();
  } // einde methode main


} // einde klasse DeckOfCards
```

6.2 ALGORITME : shuffle voorbeeld

King of Diamonds	Jack of Spades
Four of Diamonds	Six of Clubs
King of Hearts	Nine of Diamonds
Three of Spades	Four of Spades
Four of Hearts	Seven of Spades
Five of Diamonds	Eight of Hearts
Queen of Diamonds	Five of Hearts
Seven of Diamonds	Seven of Hearts
Nine of Hearts	Three of Clubs
Ten of Spades	Deuce of Hearts
Three of Hearts	Ace of Spades
Six of Hearts	Eight of Diamonds
Six of Diamonds	Deuce of Clubs
Ace of Clubs	Ten of Diamonds
Eight of Clubs	Queen of Hearts
Jack of Clubs	Ten of Clubs
Seven of Clubs	Queen of Spades
Five of Clubs	Six of Spades
Nine of Spades	Nine of Clubs
King of Spades	Ace of Diamonds
Ten of Hearts	Ace of Hearts
Queen of Clubs	Deuce of Spades
Three of Diamonds	King of Clubs
Four of Clubs	Jack of Diamonds
Eight of Spades	Five of Spades
Jack of Hearts	Deuce of Diamonds

```
import java.util.*;
public class Algorithms1
{
   private Character[] LETTERS = { 'P', 'C', 'M' },lettersCopy;
   private List< Character > list;
   private List< Character > copyList;

  public Algorithms1()
  {
    list = Arrays.asList( LETTERS ); // get List

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

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

    Collections.reverse( list ); // reverse order
    System.out.println( "\nAfter calling reverse: " );
    output( list );
```

6.3 ALGORITME : reverse, fill, copy, min, max

```
    Collections.copy( copyList, list ); //copy List
   /*overschrijft de elementen van copyList :
     dupliceert de objectreferenties
     als er copyList.size() > list.size():
                overige (laatste) elementen blijven ongewijzigd
     als er copyList.size() < list.size():
                IndexOutOfBoundsException */

    System.out.println( "\nAfter copying: " );
    output( copyList );


    Collections.fill( list, 'R' ); // fill list with R's
    System.out.println( "\nAfter calling fill: " );
    output( list );
  }
```

```
 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 ));
  System.out.printf("  Min: %s%n", Collections.min( listRef ));
 } // er is ook een overloaded versie max en min met Comparator

public static void main( String args[] )
{ new Algorithms1(); }
}
```

```
Initial list:
The list is: P C M
Max: P  Min: C

After calling reverse:
The list is: M C P
Max: P  Min: C
```

```
After copying:
The list is: M C P
Max: P  Min: C

After calling fill:
The list is: R R R
Max: R  Min: R
```

6.4 ALGORITME : binarySearch

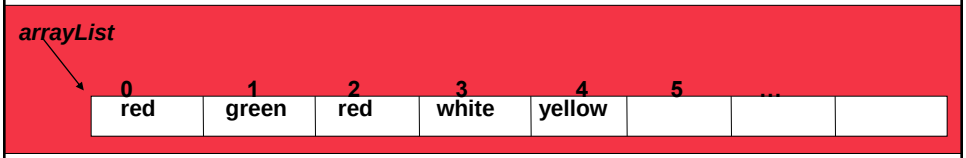
- **binarySearch**(List<? extends T> list, T key)
  - Zelfde gedrag als bij Arrays
  - Ook overloaded met als derde argument een Comparator.
  - Indien er meerdere gelijke sleutelwaarden zijn is er geen garantie welke sleutel ervan wordt geselecteerd.

6.4 ALGORITME : binarySearch

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

   public BinarySearchTest()
   {
      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" );
      System.out.printf( "yellow: %d%n", result );

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

   Sorted ArrayList: [green, pink, red, white, yellow]
   yellow: 4
   purple: -3 // -3
               // = de indexwaarde_in_geval_van_invoegen*-1 – 1 = 2
```

6.5 ALGORITME : frequency en disjoint

```
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" );
System.out.printf("%n%nFrequency of red in arrayList: %d%n",frequency);
```

2

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

```
list and arrayList have elements in common
```

Methoden klasse Collections:

```
static <T> boolean addAll(Collection<? super T> c, T[] a);
static int binarySearch(List<? extends T>  , T key);
static int binarySearch(List<? extends T>, T key, Comparator<? super T> c );
static <T> void copy(List<? super T> dest, List<? extends T> src);
static <T> Enumeration<T> enumeration(Collection<T> c);
static <T> void fill(List<? super T> list, T obj); static int indexOfSubList(List<?> source, List<?> target);
static int lastIndexOfSubList(List<?> source, List<?> target);static <T> ArrayList<T> list(Enumeration<T> e)
static <T extends Object, Comparable<? super T>> T max(Collection<? extends T> coll);
static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp);
static <T extends Object, Comparable<? super T>> T min(Collection<? extends T> coll);
static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp);
static <T> List<T> nCopies(int n, T o); static <T> boolean replaceAll(List<T> list, T oldVal, T newVal);
static void reverse(List<?> list); public static Comparator<Object> reverseOrder();
static <T> Comparator<T> reverseOrder(Comparator<T> cmp); static void rotate(List<?> list, int distance);
static void shuffle(List<?> list);static void shuffle(List<?> list, Random rnd);static <T> Set<T> singleton(T o)
static <T> List<T> singletonList(T o); static <K,V> Map<K,V> singletonMap(K key, V value);
static <T extends Comparable<? super T>> void sort(List<T> list);
static <T> void sort(List<T> list, Comparator<? super T> c);  static void swap(List<?> list, int i, int j);
static <T> Collection<T> synchronizedCollection(Collection<T> c);
                          //ook voor List, Map, Set, SortedMap, SortedSet
static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);
                          //ook voor List, Map, Set, SortedMap, SortedSet
```

Oefening: Algoritme (OefFruit2_opgave.java)

Gebruik de ArrayList list van fruit van vorige oef...

- Sorteer de fruit list oplopend.
- Voeg een nieuw fruit sapodilla toe aan de lijst. Gebruik hiervoor de methode addOrdered die je toevoegt in je eigen klasse CollectionOperaties.

48

7. KLASSE Stack

- Klasse java.util.Stack<E>

  – In een stack kunnen we objecten plaatsen (**push()**) en objecten afhalen (**pop()**).

  – Stacks werken volgens het **LIFO-principe** (Last In, First Out), wat betekent dat het laatste object dat we op de stack hebben geplaatst (push()) het eerste is dat we met de methode pop() ontvangen.

  – De klasse Stack<E> **is een subklasse van Vector<E>**.

7.1 KLASSE ArrayDeque (als stack)

```
public class ArrayDeque<E>
extends AbstractCollection<E>
implements Deque<E>, Cloneable, Serializable
```

- De klasse **ArrayDeque** is performanter dan klasse **Stack**
- Voorbeeld

  Deque<String> stack = new **ArrayDeque<>**();

7.1.1 KLASSE StackTest

```
import java.util.Stack;
import java.util.EmptyStackException;
public class StackTest
{
   public StackTest()
   {
      Deque< Number> stack = new ArrayDeque<>();
```

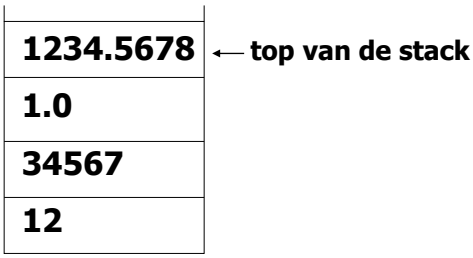
Een lege stack van type `Number` wordt gecreëerd. Klasse `Number` is de superklasse van Long, Integer, Double, Float

HoGent

```
Long longNumber = 12L;
Integer intNumber = 34567;
Float floatNumber = 1.0F;
Double doubleNumber = 1.5678;

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

methode `push` voegt een `Number` op de top van de `Stack` toe.

```
stack contains: 12 (top)

stack contains: 12 34567 (top)

stack contains: 12 34567 1.0 (top)

stack contains: 12 34567 1.0 1.5678 (top)
```



HoGent

```
// alle elementen van de stack verwijderen:
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
```

```
1234.5678 popped
stack contains: 12 34567 1.0 (top)

1.0 popped
stack contains: 12 34567 (top)

34567 popped
stack contains: 12 (top)

12 popped
stack is empty

java.util.NoSuchElementException
 at java.util.ArrayDeque.removeFirst
 at java.util.ArrayDeque.pop
 at StackTest.<init>(StackTest.java:36)
 at StackTest.main(StackTest.java:65)
```

```
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
```

8. INTERFACE Queue

- Interface java.util.Queue<E>
  - In een queue kunnen we objecten plaatsen (**offer()**) en objecten afhalen (**poll()**).
  - Queues werken volgens het **FIFO-principe** (First In, First Out), wat betekent dat het eerste object dat we op de queue hebben geplaatst (offer()) het eerste is dat we met de methode poll() ontvangen.
  - Voorbeeld:

    Queue<String> queue = new **ArrayDeque<>**();

HoGent

8. INTERFACE Queue

- Klasse java.util.PriorityQueue<E>
  - In een prioriteitenqueue kunnen we objecten plaatsen (**offer()**) en objecten afhalen (**poll()**).

    De objecten worden gesorteerd volgens de 'natuurlijke ordening' (methode compareTo() van interface Comparable)

    public **PriorityQueue**()

    of volgens een Comparator-object

    public **PriorityQueue**(int initialCapacity, Comparator<? super E> comparator)

8.1 KLASSE PriorityQueueTest

```
import java.util.PriorityQueue;

public class PriorityQueueTest
{
  public static void main( String args[] )
  {
    Queue< Double > queue = new PriorityQueue< >();

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

Een lege `Prioriteitenqueue` van type `Double` wordt gecreëerd.

methode `offer` voegt een `Double` toe aan de `Prioriteitenqueue`.

HoGent

```
    System.out.print( "Polling from queue: " );

    // display elements in queue
    while ( queue.size() > 0 )
    {
      System.out.printf( "%.1f ", queue.peek() );
            // geeft het top-element weer

      queue.poll(); // verwijdert het top element

    } // end while
  } // end main
} // end class PriorityQueueTest
```

```
Polling from queue: 3.2  5.4  9.8
```

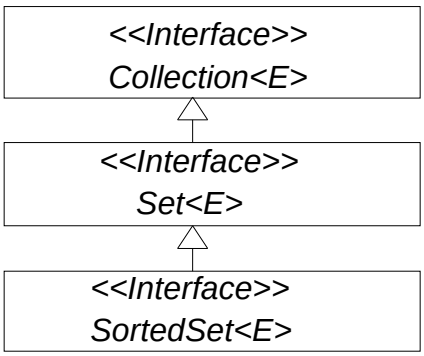
8.2 KLASSE ArrayDeque (als queue)

```
public class ArrayDeque<E>
extends AbstractCollection<E>
implements Deque<E>, Cloneable, Serializable
```

- De klasse **ArrayDeque** is performanter dan een queue, opgebouwd uit een **LinkedList**.
- Voorbeeld

  Queue<String> queue = new **ArrayDeque<>**();

9. SET INTERFACE



HashSet<E> is een implementatie-klasse van interface Set.

TreeSet<E> is een implementatie-klasse van interface SortedSet.

Sinds 1.6 NavigableSet(extends SortedSet) 60

9. SET INTERFACE

- Collectie die unieke elementen bevat.
- HashSet<E>
  - Implementatie op basis van hashtabel.
- TreeSet<E>
  - Implementatie op basis van een boomstructuur.
  - Implementeert SortedSet<E>: subinterface van Set<E>.
    Sinds 1.6 NavigableSet(extends SortedSet)

9. SET INTERFACE

```
// Voorbeeld: gebruik HashSet om dubbels te verwijderen.
import java.util.*;
public class SetTest
{
  private static final String COLORS[] = { "red", "white",
    "blue","green", "gray", "orange",
    "tan", "white", "cyan","peach", "gray", "orange" };

 public SetTest()
 {
    List< String > list = Arrays.asList( COLORS);
    System.out.printf( "ArrayList: %s%n", list );
    printNonDuplicates( list );
 }
```

9.1 SET : HashSet

```
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();
}
```

9.1 SET : HashSet

```
 public static void main( String args[] )
 {
     new SetTest();
 }
}
```

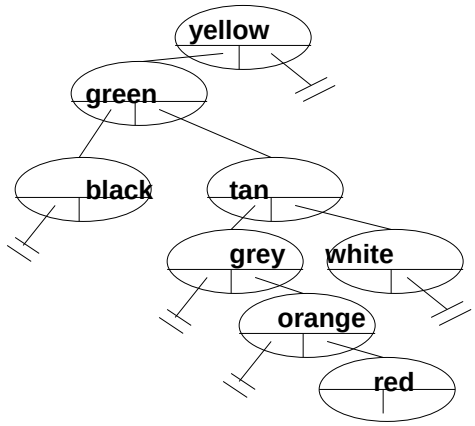
```
ArrayList: [red, white, blue, green, gray, orange, tan, white,
cyan, peach, gray, orange]

Nonduplicates are:
red cyan white tan gray green orange blue peach
```

9.2 SET : TreeSet

```
import java.util.*;
// Gebruik TreeSet om een array te sorteren en dubbels te
// elimineren.
// Illustreert 'range view' methoden: selectie van een
// deel van de Collection

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


9.2 SET : TreeSet

```
public SortedSetTest()
{
  //array names = { "yellow","green","black","tan",
  // "grey","white","orange","red","green" };
  SortedSet< String > tree =
       new TreeSet< >( Arrays.asList( names ) );
  System.out.println( "sorted set: " ); printSet( tree );
```



9.2 SET : TreeSet

```
// 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() ); }
```

HoGent

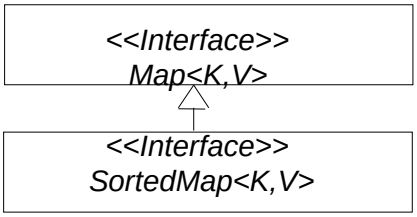
9.2 SET : TreeSet

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

   System.out.println();
}

public static void main( String args[] )
{
   new SortedSetTest();
}
}
```

```
Sorted set:
black green grey orange red tan white yellow

headSet ("orange"):  black green grey
tailSet ("orange"):  orange red tan white yellow
first: black
last : yellow
```

HoGent

10. MAP INTERFACE



HashMap<K,V> en **Hashtable<K,V>** zijn implementatie-klassen van Map.

TreeMap<K,V> is een implementatie-klasse van SortedMap. Sinds 1.6 NavigableMap(extends SortedMap)

De klasse **Properties** is een **subklasse van Hashtable<K,V>**.

10. MAP INTERFACE

- Verzameling van key-value paren. Bij elke sleutel hoort precies één waarde.
- Implementaties van interface Map<K,V>
  - HashMap<K,V>
    - Elementen opgeslagen in hash-tabel.
  - Hashtable<K,V>
    - Zoals HashMap maar verouderde versie.
    - Werpt een NullPointerException indien de sleutel of value null is.
  - TreeMap<K,V>
    - Elementen opgeslagen in boomstructuur.
    - Implementatie van SortedMap subinterface van Map. Gebruik de natuurlijke volgorde of een Comparator.

10.1 MAP: HashMap : inleiding

- Klasse java.util.HashMap<K,V>
  - Een hashmap wijst sleutels toe aan waarden.
  - **Vb.: hashMap "Werknemers"**

sleutel	waarde
123 566 411	KETERS SANDRA
899 455 178	WAERLOP JURGEN
…	

10.2 MAP: HashMap : methode get

- Methode **get**
  - Aan de hand van een sleutel kunnen we de overeenkomstige waarde ophalen

    ➔ **public V get(K sleutel)**

    Geeft de overeenkomstige waarde terug of geeft null terug indien de sleutel niet in de hash-tabel voorkomt.
    - Zowel de sleutel als de value kunnen null zijn.

10.3 MAP: HashMap : hashing

– Een hash-tabel is een gegevensstructuur die gebruik maakt van hashing:
  - De sleutel wordt omgezet naar een array index. Met deze index wordt de waarde opgezocht.
    ➔ Deze basistechniek heet **hashing**.
  - Indien twee verschillende sleutels dezelfde array index opleveren dan spreekt men van een **collision**.

10.4 MAP: HashMap : hash bucket

– Om collisions te voorkomen maakt JAVA gebruik van "hash bucket (= emmer)":
  - De hash-tabel bestaat uit cellen. Elke cel is een **"hash bucket"**. Een **koppel sleutel-waarde** wordt toegekend aan een bepaalde **"bucket"**. De techniek **hashing** bepaalt in welke bucket het koppel sleutel-waarde terechtkomt.

10.5 MAP: HashMap : methode put

- Methode **put**
  - We kunnen een koppel sleutel-waarde in de hash-tabel zetten door de methode put

➔ **public V put(K key, V value)**

- Geeft de voorgaande waarde terug of geeft null terug indien de sleutel nog niet in de hash-tabel voorkwam.

10.6 MAP: HashMap : capaciteit en laadfactor

- Hash-tabellen hebben een **capaciteit** en een **laadfactor**.

- De **capaciteit** is het aantal emmers ("buckets") dat de hash-map bevat.
- De **laadfactor** is een getal tussen 0 en 1. Deze vertelt ons hoe vol een hash-map kan worden voordat de capaciteit wordt verhoogd. Als we de laadfactor niet opgeven wanneer we een hash-map creëren, wordt deze ingesteld op 0.75. Dit betekent dat als het aantal vermeldingen 75% van de capaciteit bereikt, de capaciteit wordt verhoogd met de methode **rehash()**.

10.7 MAP: HashMap : constructoren

- **Er zijn meerdere constructoren voor de klasse HashMap:**
  - **HashMap()**
    Creëert een lege hash-tabel met een capaciteit van 16 en een laadfactor 0.75.
  - **HashMap(int initialCapacity)**
    Creëert een lege hash-tabel met een capaciteit aangegeven door de integer “capaciteit“ en een laadfactor 0.75.
  - **HashMap(int initialCapacity, float loadFactor)**
    Creëert een lege hash-tabel met een capaciteit aangegeven door de integer “capaciteit“ en een laadfactor aangegeven door de float “laadfactor“.
  - **HashMap(Map<? extends K,? extends V> m)**
    Creëert hash-tabel met dezelfde mapping als hashMap1

10.8 MAP: HashMap : voorbeeld

- Voorbeeld

– Van een ingegeven zin wordt er afgebeeld hoeveel keer een bepaald woord erin voorkomt.

  – De **sleutels** zijn de woorden. De **overeenkomstige waarden** zijn gehele getallen. Deze getallen stellen het aantal keer, dat een woord in de zin voorkomt, voor.

10.8 MAP: HashMap : voorbeeld

```
Enter a string:
To be or not to be: that is the question Whether 'tis nobler to suffer
Map contains:
Key     value
'tis            1
be          1
be:         1
is          1
nobler          1
not         1
or          1
question        1
suffer          1
that            1
the         1
to          3
Whether         1

size:13
isEmpty:false
```

HoGent

10.8 MAP: HashMap : voorbeeld

```
import java.util.Scanner;
public class WordTypeCount
{
  private Map< String, Integer > map;
  private Scanner scanner;

  public WordTypeCount()
  {
    map = new HashMap< >();
    scanner = new Scanner( System.in );
       // een scanner creëren
    createMap(); // de map opvullen
    displayMap(); // de map tonen
  }
```

Een lege hashMap creëren: sleutel is van type String en de waarde is van type Integer.

10.8 MAP: HashMap : voorbeeld

```
/* vraagt de tekst aan de gebruiker, telt het voorkomen van elk woord uit
de invoertekst en bergt de frequentie op in de hashmap */
private void createMap()
{
   System.out.println( "Enter a string:" );
   String input = scanner.nextLine();
   String[] tokens = input.split(“ “);

   for ( String token : tokens )
   {
    String word = token.toLowerCase();
    if ( map.containsKey( word ) ) // als de map het woord bevat
    {
          int count = map.get( word ); // get value en
          map.put( word, count + 1 );  // verhoog
    }
    else // anders voeg nieuw woord toe in map met een value 1
          map.put( word, 1 );
   }
}
```

Uitleg: zie volgende slide

10.8 MAP: HashMap : voorbeeld

```
if ( map.containsKey( word ) )
{
  // overeenkomstige waarde opvragen en met één verhogen:
  int count = map.get( word );
  map.put( word, count + 1 );
}
else // de sleutel "word" en waarde 1 in de hash-map zetten
  map.put( word, 1 );
```

- **Indien het woord "word" (= sleutel) reeds in de hash-map voorkomt,** dan wordt zijn overeenkomstige waarde (count) met één verhoogd en terug in de hash-tabel geplaatst.
- **Anders** wordt het woord "word" (=sleutel) en de waarde 1 in de hash-tabel geplaatst.

10.8 MAP: HashMap : voorbeeld

```
// toont de inhoud van de hash-tabel op het scherm. De sleutel worden in
alfabetische volgorde weergegeven.
private void displayMap()
{
  Set< String > keys = map.keySet();
                  // geeft alle sleutels terug

  // de sleutels sorteren:
  TreeSet< String > sortedKeys =
          new TreeSet< >( keys );

  System.out.println( "Map contains:\nKey\t\tValue" );

  // doorloop alle sleutels:
  for ( String key : sortedKeys )
    System.out.printf( "%-10s%10s%n", key, map.get(key));

  System.out.printf( "%nsize:%d%nisEmpty:%b%n",
                     map.size(),map.isEmpty() );
}
```

10.9 Klasse Hashtable – klasse HashMap

De klasse Hashtable<K,V> is een verouderde klasse en wordt vervangen door de nieuwe klasse HashMap<K,V>.

Klasse HashMap<K,V> bevat bijna dezelfde methodes als de klasse Hashtable<K,V>. Een groot verschil is dat HashMap<K,V> niet gesynchroniseerd is en een Hashtable<K,V> wel.

Oefening: Set en Map

Methoden interface Map<K,V>
void **clear**(); boolean **containsKey**(Object key); boolean **containsValue**(Object value);
Set<Map.Entry<K,V>> **entrySet**() ; boolean **equals**(Object o); V **get**(Object key);
int **hashCode**(); boolean **isEmpty**(); Set<K> **keySet**(); V put (K key, V value) ;
void **putAll**(Map<? extends K,? extends V> t) ; V **remove**(Object key); int **size**();
Collection<V> **values**();

Methoden interface Map.Entry<K,V>:
boolean **equals**(Object o); K **getKey**(); V **getValue**(); int hashCode();
V **setValue**(V value);

Oefening: Set en Map (OefFruit3_opgave.java)

- Berg de fruit list van vorige oefeningen in een boom op zodat dubbels geëlimineerd worden. Er moet ook de mogelijkheid zijn de bijhorende prijs (decimale waarde) bij te houden.
- Doorloop de boom in lexicaal oplopende volgorde en vraag telkens de bijhorende prijs, die je mee in de boom opbergt.
- Druk vervolgens de volledige lijst in twee kolommen (naam : prijs) in lexicaal oplopende volgorde af op het scherm.

- De collections van het Collections framework zijn unsynchronized.

- Via synchronization wrapper klasse converteren we collections tot synchronized versies.

  public static Collections methoden:
  - <T> Collection<T> synchronizedCollection (Collection<T> c)
  - <T> List<T> synchronizedList(List<T> aList)
  - <T> Set<T> synchronizedSet(Set<T> s)
  - <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
  - <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
  - <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)

  Vb:

```
List<String> list = Collections.synchronizedList(
                                new ArrayList<>() );
```

11. UNMODIFIABLE WRAPPERS

- Convertie naar niet wijzigbare collections.
  - Throw UnsupportedOperationException bij poging tot wijzigen van collectie.

public static Collections methoden:
- <T> Collection<T> unmodifiableCollection (Collection<T> c)
- <T> List<T> unmodifiableList(List<T> aList)
- <T> Set<T> unmodifiableSet(Set<T> s)
- <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)
- <K,V> Map<K,V> unmodifiableMap(Map<K,V> m)
- <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,V> m)

12. CHECKED WRAPPERS

- Zijn collecties waarvan at runtime gecontroleerd wordt of er elementen van het juiste type worden toegevoegd.
  => ClassCastException bij foutief type

- List<Integer> lijst=new ArrayList< >();

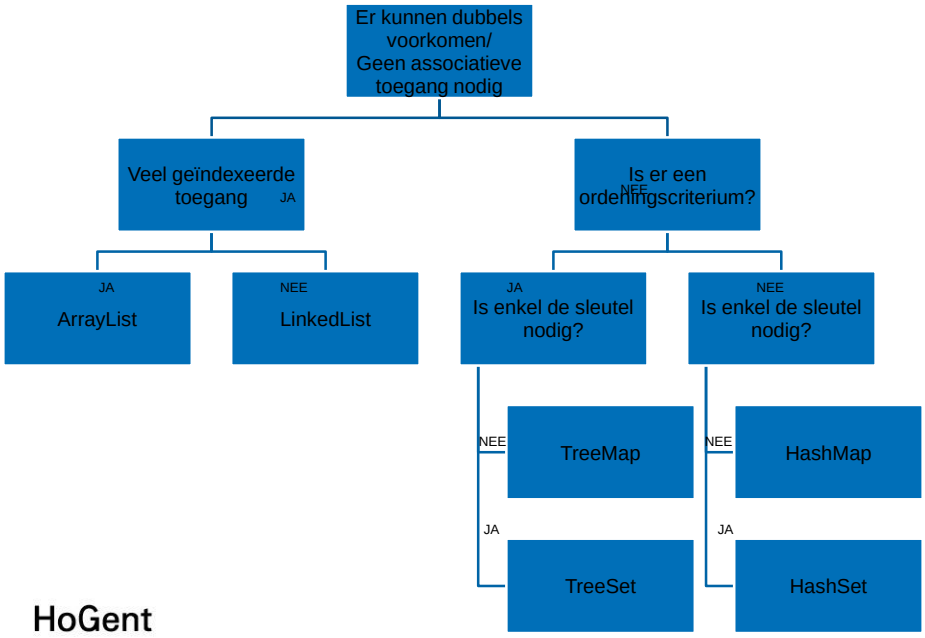
Vervangen door:

```
List<Integer> checkedlijst=
    Collections.checkedList(
        new ArrayList< >(),Integer.class);
```


13. ABSTRACT IMPLEMENTATIONS

- Er zijn abstracte implementaties van de collection interfaces die als basis voor een zelfgedefiniëerde implementatie kunnen dienen.
  - AbstractCollection
  - AbstractList
  - AbstractMap
  - AbstractSequentialList
  - AbstractSet

14. OVERZICHT



HoGent

Nieuw in Java7 : DIAMOND operator

```
List<String> colorList = new ArrayList<>();
```

=> Bij creatie (instantiëring) hoeft het type niet meer vermeld te worden.

Java6

```
List<String> colorList = new ArrayList< String >();
```

HoGent

92