



Les collections en Java



François ANDRE



Objectifs



Comprendre la notion d'égalité en Java



Comprendre les notions élémentaires d'algorithmique



Découvrir les List, Set et Map



Egalité de deux éléments en Java

Cas des types primitifs

-
- **L'égalité des types primitifs se base sur leurs valeurs et utilise l'opérateur ==**

```
int a = 5;  
int b = 5;  
System.out.println(a==b); // true
```

- **Ce principe s'applique dans certains cas rares à la classe String**

```
String a = "toto";  
String b = "toto";  
System.out.println(a==b); // true  
System.out.println(a.toLowerCase()==b.toLowerCase()); // false
```

- Par contre **ce mécanisme ne s'applique pas aux classes enrobant les types primitifs**

```
Integer a = new Integer(5);  
Integer b = new Integer(5);  
System.out.println(a==b); // false
```



Egalité de deux éléments en Java

Cas des objets

-
- Dans le cas des objets, l'opérateur `==` compare uniquement les adresses mémoire.

```
StringBuffer a = new StringBuffer("toto");  
StringBuffer b = new StringBuffer("toto");  
System.out.println(a==b); // false
```

- On peut avoir besoin de vérifier si **deux instances distinctes correspondent au même état**. Pour cela il faut utiliser la méthode ***equals()***. Cette méthode est présente dans la classe `Object`. Son comportement par défaut est celui du `==`;
- Cette méthode devra donc être surchargée au sein de la classe concernée.
- La surcharge de la méthode ***equals()*** implique la surcharge de la méthode ***hashCode()***.

Egalité de deux éléments en Java

Exemple simplifié

Ville

```
public class Ville {  
    public String nom;  
  
    public Ville(String nom) {  
        this.nom = nom;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        return ((Ville) obj).nom .equals(this.nom);  
    }  
}
```

Surcharge de la méthode equals()

Main

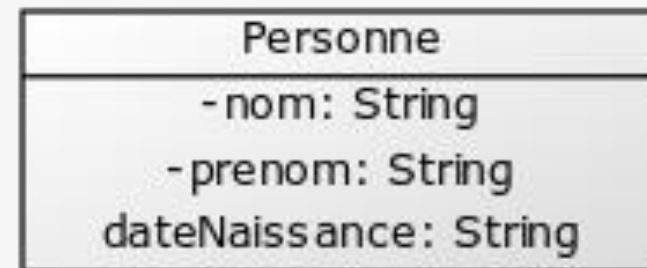
```
public static void main(String[] args) {  
    Ville a = new Ville("Toulouse");  
    Ville b = new Ville("Toulouse");  
    System.out.println(a.equals(b)); // True  
}
```

Egalité de deux éléments en Java

Méthode equals()

La méthode equals() est souvent fastidieuse à écrire/maintenir

- Elle doit vérifier que l'objet passé en paramètre n'est pas *null*
- Elle doit s'assurer que l'objet passé en paramètre à une classe convenable
- Elle enchaîne en général plusieurs comparaisons d'attributs



```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Personne other = (Personne) obj;
    if (dateNaissance == null) {
        if (other.dateNaissance != null)
            return false;
    } else if (!dateNaissance.equals(other.dateNaissance))
        return false;
    if (nom == null) {
        if (other.nom != null)
            return false;
    } else if (!nom.equals(other.nom))
        return false;
    if (prenom == null) {
        if (other.prenom != null)
            return false;
    } else if (!prenom.equals(other.prenom))
        return false;
    return true;
}
```

Egalité de deux éléments en Java

Méthode *equals()*

Il est préférable :

- De laisser l'IDE générer la méthode *equals()*.
- D'utiliser la librairie Apache Commons Lang 3 qui propose un builder facilitant la mise en place de la méthode *equals()*.
- Si la classe implémente *Comparable*, la méthode *equals()* doit utiliser la méthode *compareTo()*.



```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.8.1</version>
</dependency>
```

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Personne personne = (Personne) obj;
    return new EqualsBuilder().append(nom, personne.nom).append(prenom, personne.prenom)
        .append(dateNaissance, personne.dateNaissance).isEquals();
}
```




Egalité de deux éléments en Java

Méthode hashCode()

Une deuxième méthode de la classe *Object* intervient dans la notion d'égalité: **hashCode()**

Elle est utilisée pour calculer une clé représentant partiellement l'état d'un objet. Cette clé est par exemple utilisée dans les tables de hachage.

Règle 1

Si deux objets sont égaux (au sens de *equals*), la méthode hashCode doit retourner la même valeur.

Règle 2

Deux objets différents (au sens de *equals*) peuvent avoir le même hashCode.

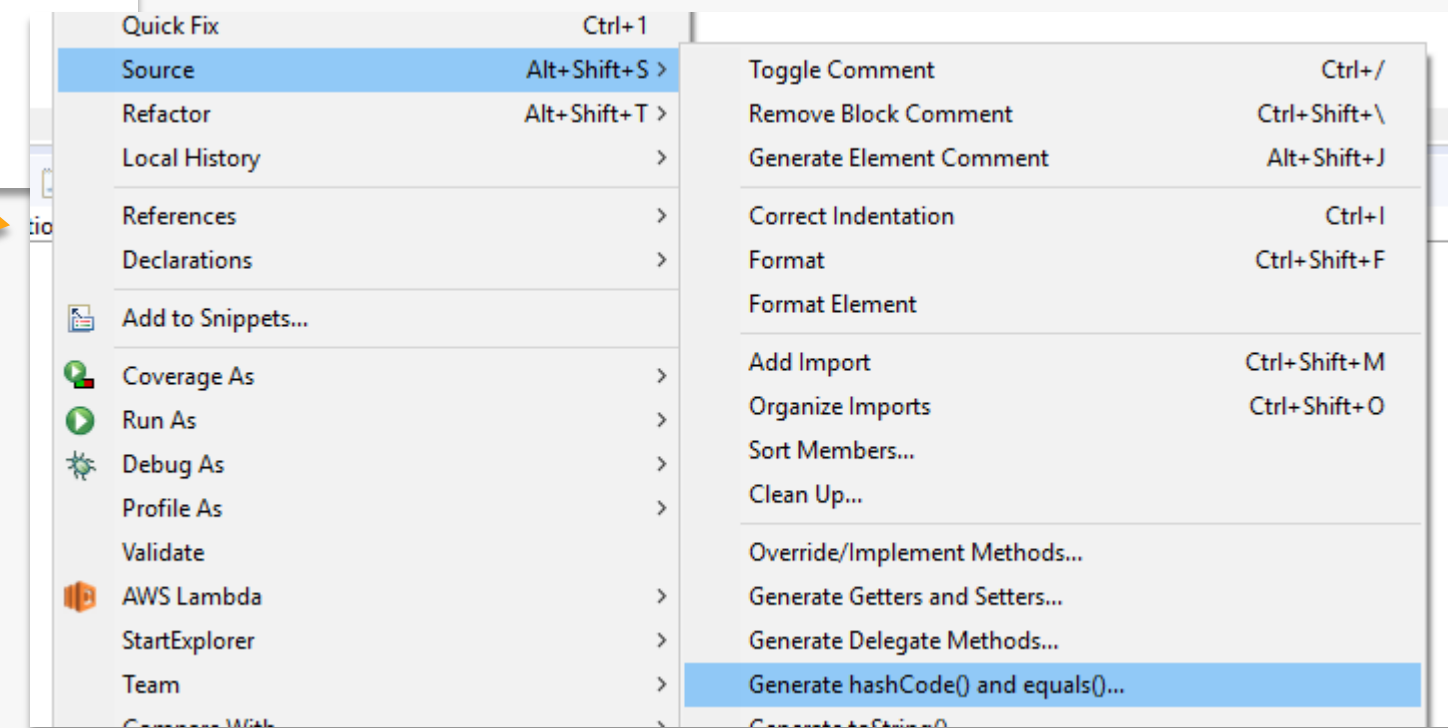
Egalité de deux éléments en Java

Méthode `hashCode()`

La méthode `hashCode` est assez complexe à mettre en œuvre manuellement.

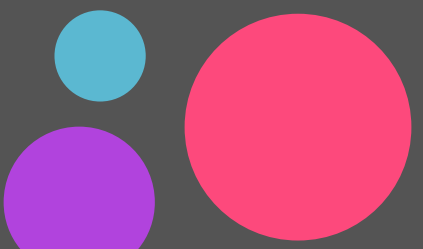
```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((dateNaissance == null) ? 0 : dateNaissance.hashCode());
    result = prime * result + ((nom == null) ? 0 : nom.hashCode());
    result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
    return result;
}
```

Il est préférable, là aussi, de laisser l'IDE générer cette méthode ou d'utiliser le **HashCodeBuilder** de la librairie Apache Commons Lang 3





Les collections





Les collections

Introduction

-
- Le Framework *Collection* est extrêmement utilisés en Java. Son but est de traiter des ensembles d'objets.
 - Elle a été introduite dès la version 2 de Java et a évolué au cours des différentes versions du JDK sans toutefois remettre en cause les éléments fondamentaux.
 - Les collections principalement utilisées sont:
 - Les listes et les ensembles: groupes parcourables d'éléments
 - Les Map: groupe d'éléments repérés sous la forme clé/valeurs
 - Pour chaque interface, le JDK propose un certain nombre d'implémentations. Certaines librairies externes proposent des implémentations plus optimisées pour certaines utilisations.
 - Le choix de l'implémentation dépend de l'utilisation souhaitée et repose sur des considérations algorithmiques.

- L'algorithmique étudie le coût (en temps mais aussi en mémoire) des traitements en fonction de la taille des données en entrée.
- Pour les collections
 - Les traitements vont concerner: le tri, la recherche, l'ajout, la modification ou la suppression
 - La taille des données, est le nombre n d'éléments dans la liste.
- Quelques exemples:
 - La recherche d'un élément dans une liste non triée est **$O(n)$**
 - Le tri d'une liste est **$O(n.\ln(n))$ (minimum théorique)**
 - La recherche d'un élément dans une liste triée est **$O(\ln(n))$**
 - L'accès à un élément dans une table de hachage est **$O(1)$**

Très informellement, $O()$ signifie de l'ordre de.

Javadoc
TreeSet

```
public class TreeSet<E>  
    extends AbstractSet<E>  
    implements NavigableSet<E>, Cloneable, Serializable
```

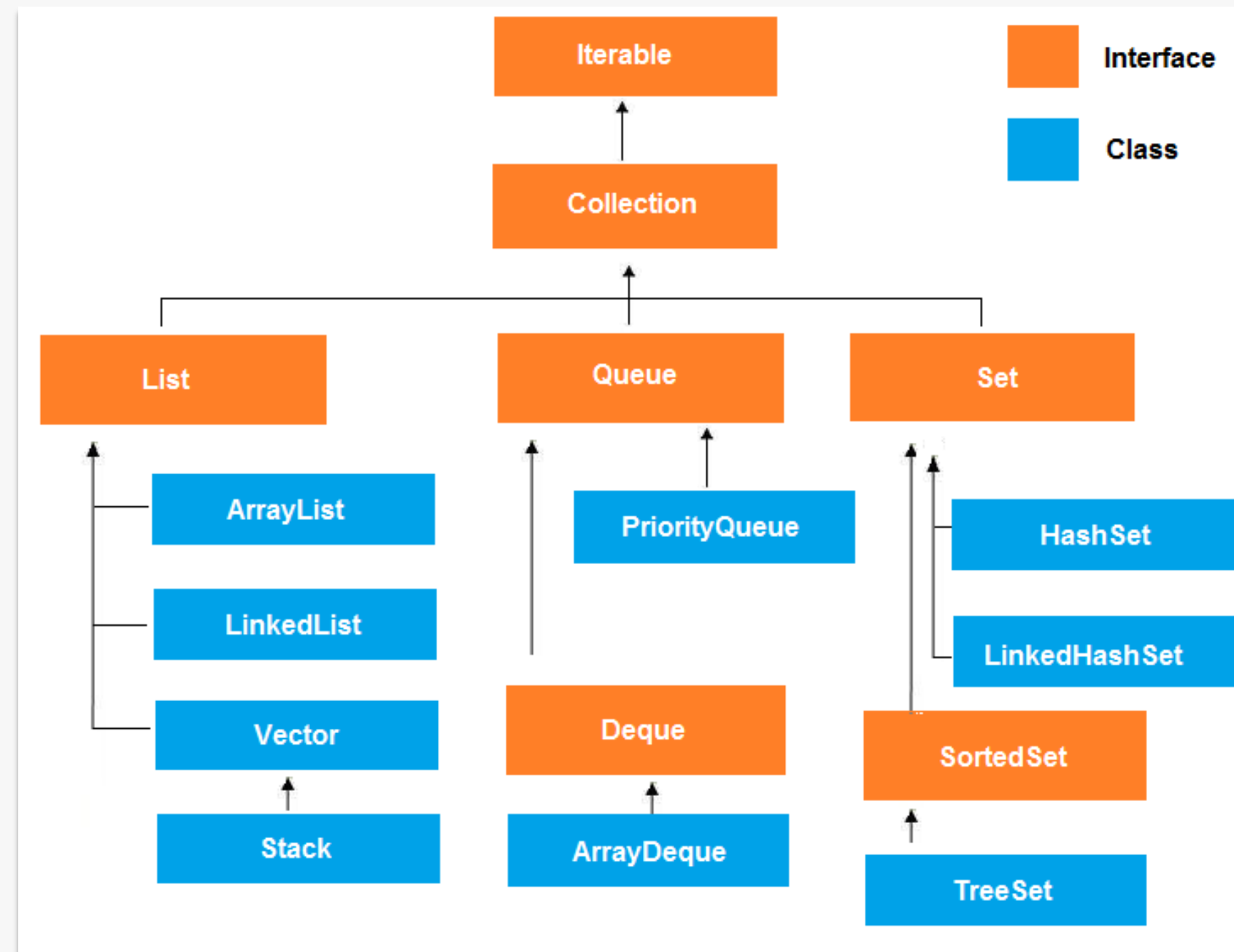
A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed $\log(n)$ time cost for the basic operations (add, remove and contains).

-
- Problème: Savoir si dans une liste de nombres deux éléments ont une somme égale à une valeur donnée V
 - Solution basique:
 - Deux boucles imbriquées:
 - 1^{ère} boucle: on parcourt la liste et, pour chaque élément X
 - 2^{ème} boucle: on parcourt la liste et, pour chaque élément Y on calcule $X+Y$ et on le compare à V .
 - Complexité: **$O(n^2)$**
 - Solution plus efficace:
 - On trie la liste
 - On parcourt la liste et, pour chaque élément X on recherche $(V-X)$
 - Complexité: **$O(n \cdot \ln(n))$**

Si $n = 10^4$, $n^2 = 10^8$, $n \cdot \ln(n) \approx 10^5$, le second algorithme est 1000 fois plus rapide.

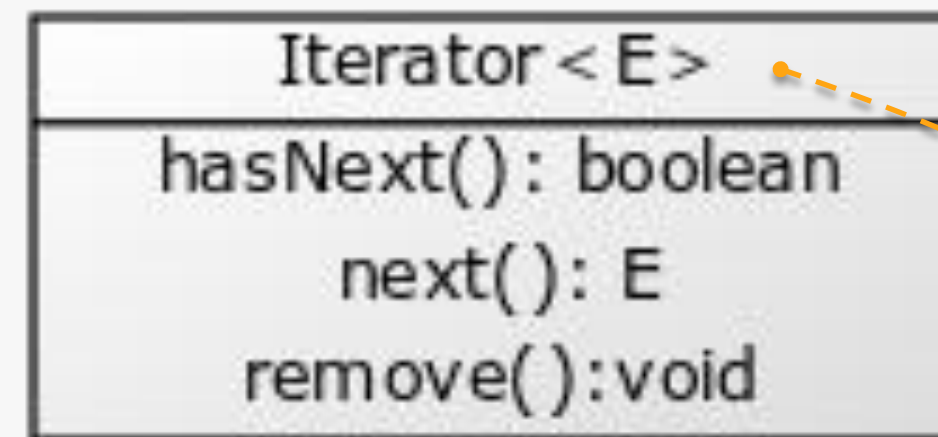
Collections



Iterable

- L'interface *Iterable* indique que la collection peut être **parcourue, qu'elle est capable de fournir successivement les différents éléments qu'elle contient.**
- Notamment, un *Iterable* est capable de fournir un *Iterator*

Iterator est un design pattern



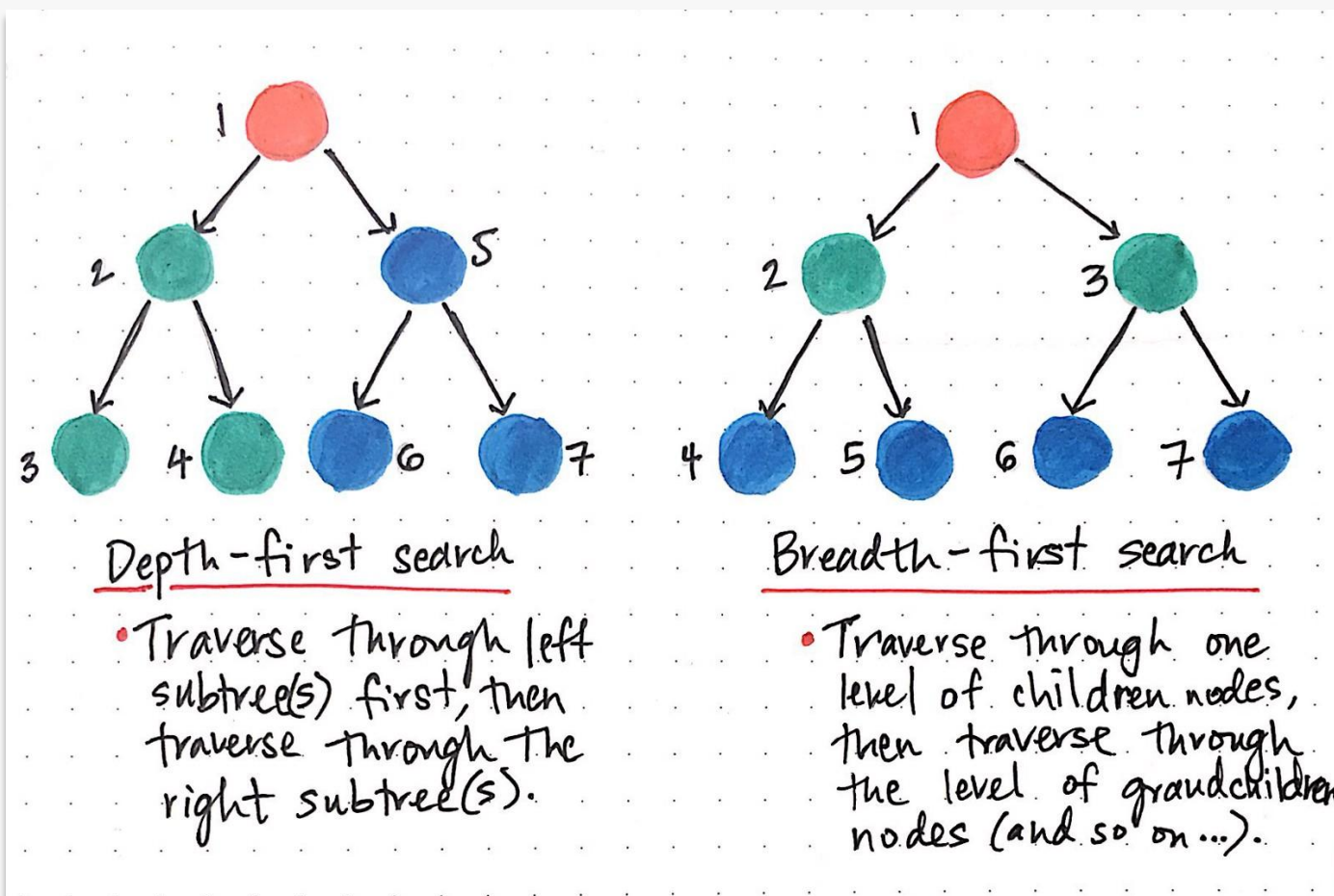
L'API utilise les *generics* pour indiquer le type de l'objet contenu dans une collection

- Un *Iterable* peut être parcouru avec la syntaxe *for*:
 `Iterable<String> resultat = ...`
 for(String s: resultat) {
 System.out.println(s);
 }

Iterable

Des Iterator partout...

- On peut définir plusieurs *Iterator* pour une même liste.
Par exemple
 - un itérateur pour parcourir les éléments du premier au dernier
 - un itérateur pour parcourir les éléments du dernier au premier
- On trouve aussi des *Iterator* dans des structures qui ne sont pas des listes



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.jgrapht.traverse

Class DepthFirstIterator<V,E>

java.lang.Object
org.jgrapht.traverse.AbstractGraphIterator<V,E>
org.jgrapht.traverse.CrossComponentIterator<V,E,DepthFirstIterator.VisitColor>
org.jgrapht.traverse.DepthFirstIterator<V,E>

Type Parameters:

V - the graph vertex type

E - the graph edge type

All Implemented Interfaces:

Iterator<V>, GraphIterator<V,E>

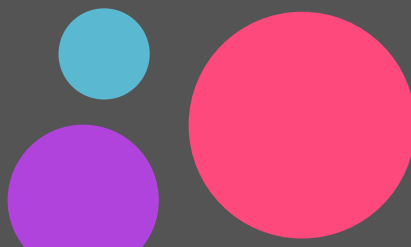


Collection

-
- L'interface *Collection* enrichit *Iterable* en ajoutant des méthodes d'ajout (*add*, *addAll*), de suppression (*remove*, *removeAll*), de présence (*contains*, *containsAll*), de taille (*size*).
 - Les méthodes comme *remove* ou *contains* utilise *equals* pour établir l'égalité entre les objets.



Les collections : les listes



- Une *List* est une *Collection* basant ses traitements sur un index. Ainsi, une *List* introduit des méthodes comme `get(index)`, `remove(index)`, `add(index, E)`, `indexOf(E)`,...
- Une *List* peut contenir des doublons ou des éléments *null*.
- Les *List* permettent d'obtenir un *ListIterator*. Cette interface est une extension d'*Iterator* qui permet de naviguer dans les deux sens, de modifier l'objet courant ou d'insérer un objet.
- Implémentations disponibles:
 - **ArrayList**: implémentation basée sur un système de tableau qui augmente sa capacité si nécessaire. Cette opération est coûteuse, ainsi il est préférable de fournir une capacité initiale adaptée. `ArrayList` est l'implémentation la plus utilisée.
 - **Vector**: implémentation *threadsafe* semblable à `ArrayList`
 - **LinkedList**: implémentation basée sur une liste chaînée. L'ajout en fin de liste est très rapide. `LinkedList` est donc très adapté pour les utilisations dans lesquelles on ajoute des éléments avant de les parcourir une seule fois.

Ces listes conservent l'ordre
d'insertion



List

Comparaison algorithmique

Operation	LinkedList	ArrayList
Get(int index)	$O(n)$	$O(1)$
Add(E element)	$O(1)$	$O(1)$ (mais $O(n)$ lorsqu'il y a augmentation de la capacité)
Add(int index, E element)	$O(n)$	$O(n)$
Remove(int index)	$O(n)$	$O(n)$
Iterator.remove	$O(1)$	$O(n)$
ListIterator.add(E element)	$O(1)$	$O(n)$

Exemple

Parcours avec un Iterator

```
public static void main(String[] args) {  
    Personne personne1 = new Personne().setNom("McCartney").setPrenom("Paul");  
    Personne personne2 = new Personne().setNom("Starr").setPrenom("Ringo");  
    Personne personne3 = new Personne().setNom("Lennon").setPrenom("John");
```

```
    List<Personne> beatles = new ArrayList<Personne>();  
    beatles.add(personne1);  
    beatles.add(personne2);  
    beatles.add(personne3);
```

```
    Iterator<Personne> iterator = beatles.iterator();
```

```
    while (iterator.hasNext()) {  
        Personne personne = iterator.next();  
        System.out.println(personne.getNom());  
    }  
}
```

La liste est typée, elle ne peut contenir que des objets qui sont des instances de *Personne*

On ne spécifie l'implémentation utilisée qu'au moment de la création de l'objet. Pour le reste de l'utilisation, il s'agit d'une *List*. Le reste du code est donc indépendant de l'implémentation choisie

L'iterator permet de parcourir les éléments sans connaître la manière dont sont stockés les objets

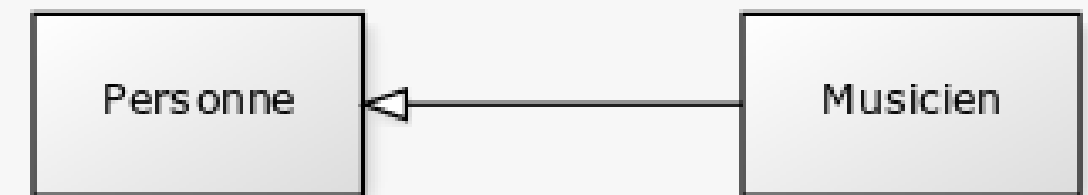
Exemple

Parcours avec un for

```
public static void main(String[] args) {  
    Personne personne1 = new Personne().setNom("McCartney").setPrenom("Paul");  
    Personne personne2 = new Personne().setNom("Starr").setPrenom("Ringo");  
    Personne personne3 = new Personne().setNom("Lennon").setPrenom("John");  
    Musicien personne4 = new Musicien().setNom("Harrison").setPrenom("George");
```

```
    List<Personne> beatles = new ArrayList<Personne>();  
    beatles.add(personne1);  
    beatles.add(personne2);  
    beatles.add(personne3);  
    beatles.add(personne4);
```

```
    for (Personne personne : beatles) {  
        System.out.println(personne.getNom());  
    }  
}
```



Trier une liste: Comparator

- L'interface *Comparator* permet de comparer deux objets entre eux et donc de les ordonner.
- La méthode *compare* doit retourner:
 - **une valeur négative** : si *e1* est inférieur à *e2*
 - **0** : si les deux éléments sont égaux
 - **une valeur positive** : si *e1* est supérieur à *e2*
- Ce comportement est similaire à la méthode *compareTo()* de la classe *String*.
- On peut définir plusieurs *Comparator* pour une même classe selon le besoin.
Par exemple on peut définir:
 - Un comparateur pour trier alphabétiquement par ordre croissant sensible à la casse
 - Un comparateur pour trier alphabétiquement par ordre décroissant sensible à la casse
 - Un comparateur pour trier alphabétiquement par ordre croissant non sensible à la casse...

Comparator < E >
compare(E e1, E e2): int

Exemple

Méthode `Collections.sort()`

```
public class PrenomComparator implements Comparator<Personne> {  
  
    public int compare(Personne o1, Personne o2) {  
        return o1.getPrenom().compareTo(o2.getPrenom());  
    }  
  
}
```

```
...  
List<Personne> beatles = new ArrayList<Personne>();  
beatles.add(personne1);  
beatles.add(personne2);  
beatles.add(personne3);  
beatles.add(personne4);
```

```
Collections.sort(beatles, new PrenomComparator());
```

```
for (Personne personne : beatles) {  
    System.out.println(personne.getPrenom());  
}
```

...

La liste est triée par le
comparator passé en
paramètre

George
John
Paul
Ringo



Comparable

-
- Il peut arriver qu'un ordre soit privilégié pour une classe donnée, on parle d'ordre *naturel*. Dans ce cas là, on peut injecter directement cet ordre au niveau de la classe via l'interface *Comparable*.

Comparable < E >
compare(E e): int

- L'interface *Comparable* est proche de *Comparator*. Cette fois c'est l'objet courant qui est comparé à l'objet passé en paramètre.
- Une classe ne peut implémenter qu'une seule fois *Comparable*.

Comparable

Exemple

```
public class Personne implements Comparable<Personne> {  
    ...  
    public int compareTo(Personne o) {  
        return this.getNom().compareTo(o.getNom());  
    }  
}  
...  
List<Personne> beatles = new ArrayList<Personne>();  
beatles.add(personne1);  
beatles.add(personne2);  
beatles.add(personne3);  
beatles.add(personne4);  
  
Collections.sort(beatles);  
  
for (Personne personne : beatles) {  
    System.out.println(personne.getNom());  
}
```

La liste est triée par le
Comparable

Harrison
Lennon
McCartney
Starr

Comparator et Java 8

Exemple

- Les comparators peuvent être fastidieux à écrire lorsque l'on doit enchaîner les comparaisons.
- La syntaxe Java 8 permet de simplifier cette tâche

```
Personne personne1 = new Personne().setNom("Jackson").setPrenom("Mickael");
Personne personne2 = new Personne().setNom("Jackson").setPrenom("Jermaine");
Personne personne3 = new Personne().setNom("Jackson").setPrenom("Tito");
Personne personne4 = new Personne().setNom("Jackson").setPrenom("Marlone");
Personne personne5 = new Personne().setNom("Jackson").setPrenom("Jackie");
```

```
List<Personne> jackson5 = new ArrayList<Personne>();
jackson5.add(personne1);
jackson5.add(personne2);
jackson5.add(personne3);
jackson5.add(personne4);
jackson5.add(personne5);
```

```
Comparator<Personne> compareur = Comparator.comparing(Personne::getNom).thenComparing(Personne::getPrenom);
```

```
Collections.sort(jackson5, compareur);
```

```
for (Personne personne : jackson5) {
    System.out.println(personne.getPrenom());
}
```



Jackie
Jermaine
Marlone
Mickael
Tito

SortedList

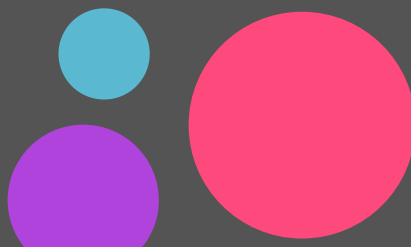
Existe-t-il une liste qui stockerait les éléments triés ?

- Il n'existe pas dans le JDK une implémentation de ce genre.
- Les solutions pour réaliser ce besoin sont:
 - D'utiliser d'autres Collections du JDK. En général, un *SortedSet* convient parfaitement. On peut dans ce cas adapter l'objet pour ne pas avoir de doublons
 - D'utiliser une implémentation tierce disponible (https://blog.scottlogic.com/2010/12/22/sorted_lists_in_java.html)
 - De créer une implémentation spécifique correspondant à son besoin.
- Dans tous les cas, il faut considérer le coût algorithmique du choix de l'implémentation

	add(Object elem)	remove(Object elem)	get(int index)	contains(Object elem)	multiple equal elements
ArrayList	$O(1)^*$	$O(n)$	$O(1)$	$O(n)$	YES
LinkedList	$O(1)$	$O(n)$	$O(n)$	$O(n)$	YES
TreeSet	$O(\log(n))$	$O(\log(n))$	N/A	$O(\log(n))$	NO
PriorityQueue	$O(\log(n))$	$O(n)$	N/A	$O(n)$	YES
SortedList	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	YES
* - <i>amortized constant time</i> (inserting n objects takes $O(n)$ time).					



Les collections : les ensembles



- **Un *Set* est une *Collection* qui ne contient pas de doublon.**
- La comparaison se base sur *equals()* - et donc *hashCode()*. Elle est vérifiée uniquement lors de l'ajout.
- Contrairement aux listes, un *Set* ne connaît pas de notion d'index. Il est toutefois iterable
- Les implémentations disponibles sont:
 - **HashSet**: implémentation basée sur une table de hachage.
 - **LinkedHashSet** : HashSet dont les éléments sont liés afin de pouvoir les itérer dans leur ordre d'insertion.
- Ces implémentations permettent des **performances constantes** - $O(1)$ - sur les actions Ajoute, Supprime, Contient. Ce résultat s'obtient grâce à une **occupation mémoire plus élevée que les List**.

Set

Mise en évidence de la gestion des doublons

```
Personne personne1 = new Personne().setNom("Jackson").setPrenom("Mickael");
Personne personne2 = new Personne().setNom("Jackson").setPrenom("Mickael");
Personne personne3 = new Personne().setNom("Jackson").setPrenom("Mickael");
Personne personne4 = new Personne().setNom("Jackson").setPrenom("Mickael");
Personne personne5 = new Personne().setNom("Jackson").setPrenom("Mickael");
```

```
Set<Personne> jacksonSolo = new HashSet<Personne>();
jacksonSolo.add(personne1);
jacksonSolo.add(personne2);
jacksonSolo.add(personne3);
jacksonSolo.add(personne4);
jacksonSolo.add(personne5);
```

```
for (Personne personne : jacksonSolo) {
    System.out.println(personne.getPrenom());
}
```

Mickael

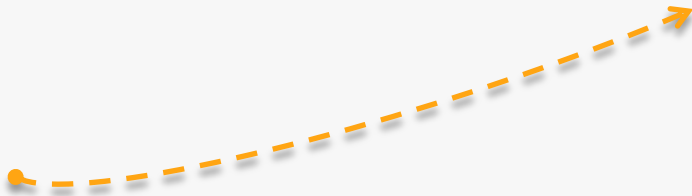
HashSet

Mise en évidence de la perte de l'ordre d'insertion

```
Personne personne1 = new Personne().setNom("Premier");  
Personne personne2 = new Personne().setNom("Deuxième");  
Personne personne3 = new Personne().setNom("Troisième");  
Personne personne4 = new Personne().setNom("Quatrième");  
Personne personne5 = new Personne().setNom("Cinquième");
```

```
Set<Personne> ensemble = new HashSet<Personne>();  
ensemble.add(personne1);  
ensemble.add(personne2);  
ensemble.add(personne3);  
ensemble.add(personne4);  
ensemble.add(personne5);
```

```
for (Personne personne : ensemble) {  
    System.out.println(personne.getNom());  
}
```



Deuxième
Quatrième
Cinquième
Premier
Troisième

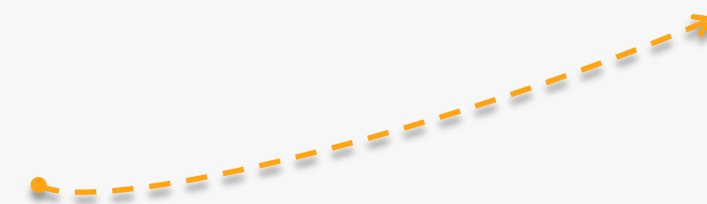
LinkedHashSet

Mise en évidence de la perte de l'ordre d'insertion

```
Personne personne1 = new Personne().setNom("Premier");  
Personne personne2 = new Personne().setNom("Deuxième");  
Personne personne3 = new Personne().setNom("Troisième");  
Personne personne4 = new Personne().setNom("Quatrième");  
Personne personne5 = new Personne().setNom("Cinquième");
```

```
Set<Personne> ensemble = new LinkedHashSet<Personne>();  
ensemble.add(personne1);  
ensemble.add(personne2);  
ensemble.add(personne3);  
ensemble.add(personne4);  
ensemble.add(personne5);
```

```
for (Personne personne : ensemble) {  
    System.out.println(personne.getNom());  
}
```



Premier
Deuxième
Troisième
Quatrième
Cinquième



SortedSet

- Un SortedSet est une sous-interface de *Set* qui permet de maintenir les éléments ordonnés en permanence.
- Cet ordre est effectué par un comparateur ou via un Comparable.
- Un SortedSet peut également extraire des sous-ensembles triés.
- L'implémentation disponible est :
 - **TreeSet**: cette implémentation permet des performances en $O(\ln(n))$ sur les actions Ajoute, Supprime, Contient.

TreeSet

Mise en évidence du tri – utilisation d'un Comparator

```
Personne personne1 = new Personne().setNom("Marx").setPrenom("Groucho");
Personne personne2 = new Personne().setNom("Marx").setPrenom("Karl");
Personne personne3 = new Personne().setNom("Hugo").setPrenom("Victor");
Personne personne4 = new Personne().setNom("Hugo").setPrenom("Boss");
Personne personne5 = new Personne().setNom("Christie").setPrenom("Agatha");
```

```
Set<Personne> ensemble = new TreeSet<Personne>(new PrenomComparator());
ensemble.add(personne1);
ensemble.add(personne2);
ensemble.add(personne3);
ensemble.add(personne4);
ensemble.add(personne5);
```

```
for (Personne personne : ensemble) {
    System.out.println(personne.getPrenom());
}
```



Agatha
Boss
Groucho
Karl
Victor

TreeSet

Mise en évidence du tri – utilisation d'un Comparable basé uniquement sur le nom

```
Personne personne1 = new Personne().setNom("Marx").setPrenom("Groucho");
Personne personne2 = new Personne().setNom("Marx").setPrenom("Karl");
Personne personne3 = new Personne().setNom("Hugo").setPrenom("Victor");
Personne personne4 = new Personne().setNom("Hugo").setPrenom("Boss");
Personne personne5 = new Personne().setNom("Christie").setPrenom("Agatha");
```

```
Set<Personne> ensemble = new TreeSet<Personne>();
ensemble.add(personne1);
ensemble.add(personne2);
ensemble.add(personne3);
ensemble.add(personne4);
ensemble.add(personne5);
```

```
for (Personne personne : ensemble) {
    System.out.println(personne.getNom() + " " + personne.getPrenom());
}
```

Dans ce cas TreeSet utilise compareTo avant equals...il faut que les deux méthodes soient en cohérence !

Christie Agatha
Hugo Victor
Marx Groucho

TreeSet

Mise en évidence du tri – utilisation d'un Comparable basé uniquement sur le nom/prénom

```
Personne personne1 = new Personne().setNom("Marx").setPrenom("Groucho");
Personne personne2 = new Personne().setNom("Marx").setPrenom("Karl");
Personne personne3 = new Personne().setNom("Hugo").setPrenom("Victor");
Personne personne4 = new Personne().setNom("Hugo").setPrenom("Boss");
Personne personne5 = new Personne().setNom("Christie").setPrenom("Agatha");
```

```
Set<Personne> ensemble = new TreeSet<Personne>();
ensemble.add(personne1);
ensemble.add(personne2);
ensemble.add(personne3);
ensemble.add(personne4);
ensemble.add(personne5);
```

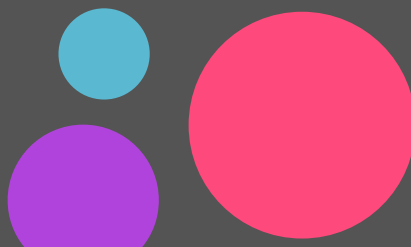
```
for (Personne personne : ensemble) {
    System.out.println(personne.getNom() + " " + personne.getPrenom());
}
```



Christie Agatha
Hugo Boss
Hugo Victor
Marx Groucho
Marx Karl



Les collections : les Map





Les Map

- L'interface Map repose sur le principe d'associer deux objets entre eux: la clé (K) et la valeur(V).
- L'utilisation classique est de déposer des couples clé/valeur : opération **put**
- De retrouver une valeur particulière à partir de sa clé: opération **get**
- De supprimer un couple à partir de sa clé
- Une Map va donc proposer des méthodes comme:
 - Ajout de couple(s): put, putAll
 - Recherche d'une valeur: get (null si la clé n'est pas présente)
 - Suppression d'un couple: remove
 - Présence d'une clé, d'une valeur: containsKey, containsValue
 - Taille: size
 - Ensemble des clés: keySet (→ une clé est unique, éventuellement nulle)
 - Collection des valeurs: values (les valeurs ne sont pas uniques)



Les Map

Proximité avec les sets

- Il y a une forte similitude entre les Map et les Set.

- Elle se traduit par

- une correspondance entre les classes

Set	Map
HashSet	HashMap
LinkedHashSet	LinkedHashMap
TreeSet	TreeMap

- L'utilisation des méthodes hashCode et equals pour les clés
- Performances algorithmiques comparables :
 - Les opérations Ajout, Suppression et Recherche d'une valeur à partir de sa clé s'effectuent en temps constant – $O(1)$ pour les HashMap et LinkedHashMap. Là aussi, ces performances s'appuient sur une plus forte consommation de mémoire.
 - Opération en $O(\ln(n))$ pour la TreeMap

Exemple

Type de la clé

Type de la valeur

```
public static void main(String[] args) {  
    Map<String, String> departements = new HashMap<>();  
    departements.put("01", "Ain");  
    departements.put("02", "Aisne");  
    departements.put("03", "Allier");  
  
    System.out.println(departements.get("02"));  
    System.out.println(departements.get("31"));  
  
    departements.remove("02");  
    System.out.println(departements.get("02"));  
}
```

Aisne
null
null

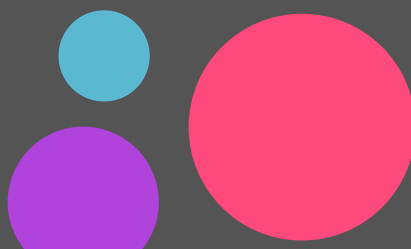


Les Map

- Les Map sont extrêmement utilisées car:
 - le mécanisme clé/valeur est très présent:
 - Formats de fichiers (ex: Json)
 - Tables de paramètres
 - ...
 - les clés peuvent être très efficaces pour représenter certains modèles de données (exemple: tableaux avec peu de valeurs)
 - la clé va permettre de découpler la relation entre la valeur et son consommateur



Divers



Librairies tierces de collections


FastUtil est une librairie basée sur la performance

Guava propose des types avancés et des méthodes élégantes

Trove est spécialisée dans les types primitifs.

Java Collections Cheat Sheet

For more awesome cheat sheets visit rebellabs.org!



Notable Java collections libraries

Fastutil
<http://fastutil.di.unimi.it/>
Fast & compact type-specific collections for Java
Great default choice for collections of primitive types, like int or long. Also handles big collections with more than 2^{31} elements well.

Guava
<https://github.com/google/guava>
Google Core Libraries for Java 6+
Perhaps the default collection library for Java projects. Contains a magnitude of convenient methods for creating collection, like fluent builders, as well as advanced collection types.

Eclipse Collections
<https://www.eclipse.org/collections/>
Features you want with the collections you need
Previously known as gs-collections, this library includes almost any collection you might need: primitive type collections, multimaps, bidirectional maps and so on.

JCTools
<https://github.com/JCTools/JCTools>
Java Concurrency Tools for the JVM.
If you work on high throughput concurrent applications and need a way to increase your performance, check out JCTools.

What can your collection do for you?

Collection class	Thread-safe alternative	Your data				Operations on your collections						
		Individual elements	Key-value pairs	Duplicate element support	Primitive support	Order of iteration			Performant 'contains' check	Random access		
						FIFO	Sorted	LIFO		By key	By value	By index
HashMap	ConcurrentHashMap	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗
HashMap (Guava)	Maps.synchronizedBiMap (new HashMap())	✗	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗
ArrayListMultimap (Guava)	Maps.synchronizedMultiMap (new ArrayListMultimap())	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
LinkedHashMap	Collections.synchronizedMap (new LinkedHashMap())	✗	✓	✗	✗	✓	✗	✗	✓	✓	✗	✗
TreeMap	ConcurrentSkipListMap	✗	✓	✗	✗	✗	✓	✗	✓*	✓*	✗	✗
Int2IntMap (Fastutil)		✗	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓
ArrayList	CopyOnWriteArrayList	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✓
HashSet	Collections.newSetFromMap (new ConcurrentHashMap<>())	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗
IntArrayList (Fastutil)		✓	✗	✓	✓	✓	✗	✓	✗	✗	✗	✓
PriorityQueue	PriorityBlockingQueue	✓	✗	✓	✗	✗	✓**	✗	✗	✗	✗	✗
ArrayDeque	ArrayBlockingQueue	✓	✗	✓	✗	✓**	✗	✓**	✗	✗	✗	✗

* $O(\log(n))$ complexity, while all others are $O(1)$ - constant time ** when using Queue interface methods: `offer()` / `poll()`

How fast are your collections?

Collection class	Random access by index / key	Search / Contains	Insert
ArrayList	$O(1)$	$O(n)$	$O(n)$
HashSet	$O(1)$	$O(1)$	$O(1)$
HashMap	$O(1)$	$O(1)$	$O(1)$
TreeMap	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$


Remember, not all operations are equally fast. Here's a reminder of how to treat the Big-O complexity notation:

$O(1)$ - constant time, really fast, doesn't depend on the size of your collection

$O(\log(n))$ - pretty fast, your collection size has to be extreme to notice a performance impact

$O(n)$ - linear to your collection size: the larger your collection is, the slower your operations will be

BROUGHT TO YOU BY



Merci

Des questions?

