Les collections et les listes d'associations

Algorithmique et langage de programmation

Gaël Thomas

Depuis le début du module

- Vous avez mis en œuvre 4 grandes structures de données
  - Des tableaux dynamiques avec l'armée de monstres (CI3)
  - Des listes chaînées avec l'armée de monstres (CI3)
  - Des arbres binaires de recherche avec la banque Lannister (CI4)
  - Des tables de hachage pour votre voyage à Bilbao (CI7)

Bravo !

Vous avez donc gagné le droit d'utiliser
la bibliothèque Java qui vous fournit
ces structures de données ☺️

La bibliothèque `java.util`

- Fournit un ensemble d'interfaces abstrayant les structures de données les plus fréquemment utilisées
- Fournit un ensemble de classes mettant en œuvre ces structures de données
- Toutes ces classes et interfaces sont génériques

Avant de commencer (1/2)

- La bibliothèque de structures de données Java nécessite des méthodes utilitaires
  - Pour effectuer une comparaison par valeurs entre objets
    (comme avec `CityId.equals` dans nos tables de hachage)
  - Pour connaître le code hachage d'un objet
    (comme avec `CityId.hashCode` dans nos tables de hachage)
  - Pour savoir si un objet est plus petit qu'un autre
    (comme avec la banque Lannister avec les comptes classés suivant un ordre lexicographique dans l'arbre binaire de recherche)

Avant de commencer (1/2)

- La bibliothèque de structures de données Java nécessite des méthodes utilitaires
  - Pour effectuer une comparaison par valeurs entre objets
    Offert directement par `Object` via la méthode `equals`
  - Pour connaître le code hachage d'un objet
    Offert directement par `Object` via la méthode `hashCode`
  - Pour savoir si un objet est plus petit qu'un autre
    Offert par l'interface `Comparable` via la méthode `compareTo`

Deux familles de structures de données

- La **collection** stocke une collection d'éléments
  - Tableaux extensibles (stocke des monstres)
  - Listes chaînées (stocke des monstres)

- La **liste d'association** associe des clés et des valeurs
  - La table de hachage (associe des noms à des villes)
  - L'arbre binaire de recherche (associe des noms à des comptes)

Deux familles de structures de données

- La **collection** stocke une collection d'éléments
  - Tableaux extensibles — `java.util.ArrayList<E>`
  - Listes chaînées — `java.util.LinkedList<E>`

- La **liste d'association** associe des clés et des valeurs
  - La table de hachage — `java.util.HashMap<K, V>`
  - L'arbre binaire de recherche (rouge-noir) — `java.util.TreeMap<K, V>`

La collection d'éléments

- L'interface `Collection<E>` représente une collection
  - `boolean add(E e)` : ajoute l'élément `e`
  - `boolean remove(Object o)` : supprime l'élément `o`
  - `boolean contains(Object o)` : vrai si collection contient `o`

  - `contains` et `remove` prennent en argument un `Object` et non un `E` pour permettre à l'utilisateur de tester la présence ou de supprimer un élément en utilisant une instance d'une autre classe

Exemple d'utilisation des collections

```
/* ArrayList ⇒ tableau extensible (cas ici) */
/* LinkedList ⇒ liste chaînée */

Collection<Monster> col = new ArrayList<>();
Monster m = new Monster("Pikachu");

col.add(m);
if(col.contains(m)) {
  System.out.println("Les collections, c'est facile !");
}
```

La classe `Iterator`

- Un `Iterator` permet de parcourir une collection
  - `E next()` : renvoie l'élément suivant et avance dans la collection
  - `boolean hasNext()` : renvoie vrai si il existe un suivant
  - `void remove()` : supprime l'élément courant

- Exemple d'utilisation

```
Collection<Monster> col = new LinkedList<>();
… /* ajoute des monstres à la collection ici */
Iterator<Monster> it = col.iterator();
while(it.hasNext()) {
  Monster cur = it.next();
  System.out.println("Monstre: " + cur.name);
}
```

La boucle « pour chaque »

- Sucre syntaxique introduit dans Java 5

```
for(Monster cur: col) {
  /* fait quelque chose avec cur */
}

Iterator<Monster> it = col.iterator();
while(it.hasNext()) {
  Monster cur = it.next();
  System.out.println("Monstre: " + cur.name);
}
```

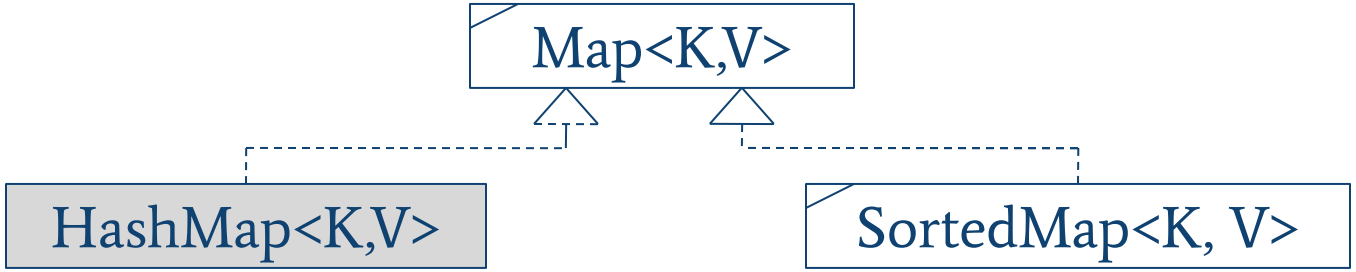
La boucle « pour cha

Bon à savoir

on peut aussi utiliser une boucle « pour chaque »
pour parcourir un tableau

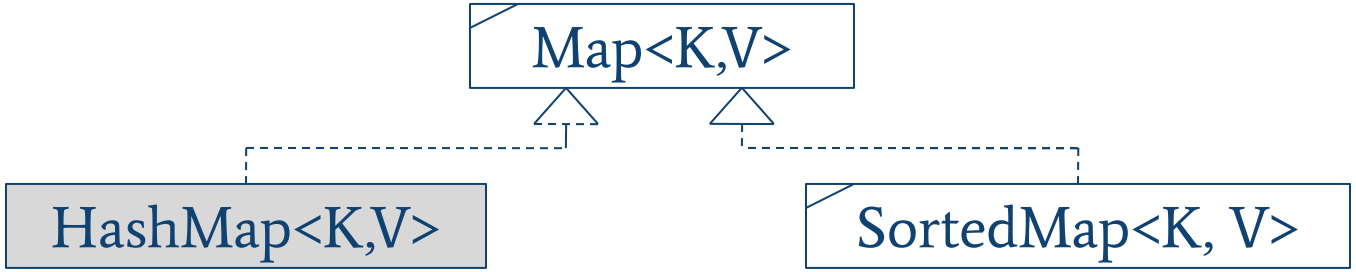
```
String[] tab = { "a", "b" };

for(String s: tab) {
    /* … */
}
```

Les tables d'association en Java (1/2)



- `Map` définit principalement quatre méthodes
  - `V put(K k, V v)` : associe `k` à `v` (renvoie ancienne valeur ou `null`)
  - `V get(K k)` : renvoie la valeur associée à `k` ou `null`
  - `boolean remove(K k)` : supprime l'association
  - `Collection<V> values()` : renvoie la collection des valeurs

Les tables d'association en Java (1/2)



- Deux principales mises en œuvre
  - La table de hachage (`HashMap`)
    - `K` doit mettre en œuvre `equals` et `hashCode`
    - Table d'association + éléments non triés
  - L'arbre binaire rouge-noir (`TreeMap` de type `SortedMap`)
    - `K` doit mettre en œuvre `Comparable`
    - Table d'association + éléments triés suivant l'ordre des clés

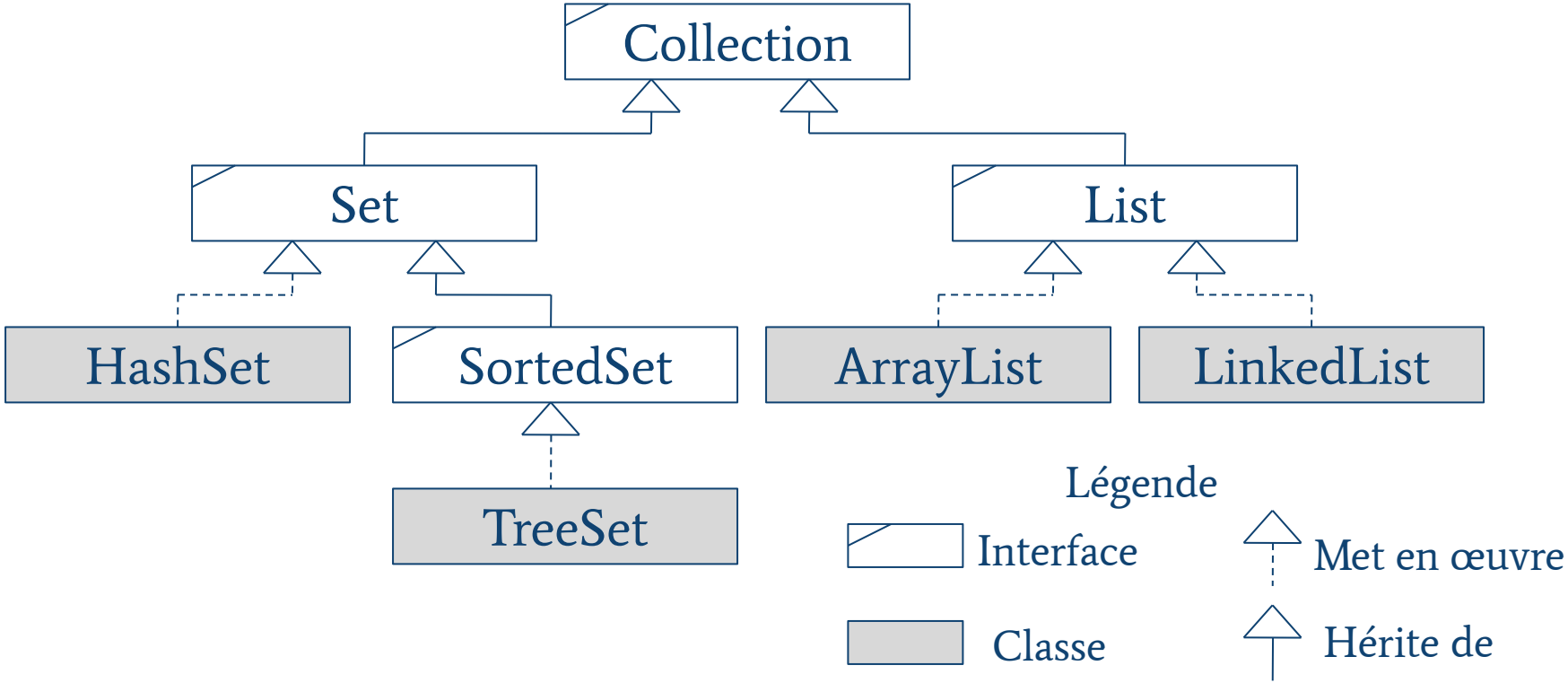
Notions clés

- `Collection` pour stocker des collections d'éléments
  - Deux principales mises en œuvre
    - `ArrayList` : tableau extensible, ajout lent (extension du tableau), accès aléatoire rapide
    - `LinkedList` : liste chaînée, ajout rapide, accès aléatoire lent
  - Parcours avec `Iterator` ou boucle « pour chaque »

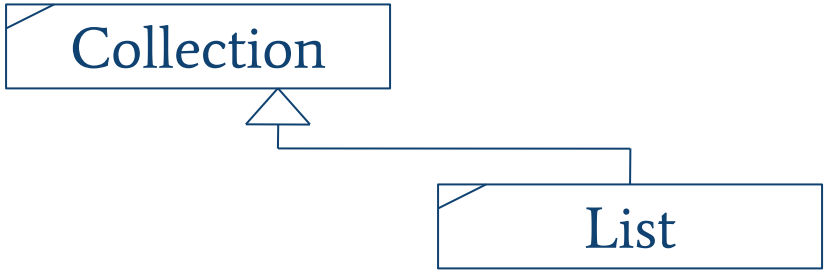
- `Map` pour associer clé et valeur
  - Deux principales mises en œuvre
    - `HashMap` : table de hachage, non triée
    - `TreeMap` : arbre binaire rouge-noir, trié suivant les clés

Pour approfondir

L'arbre d'héritage de `Collection`



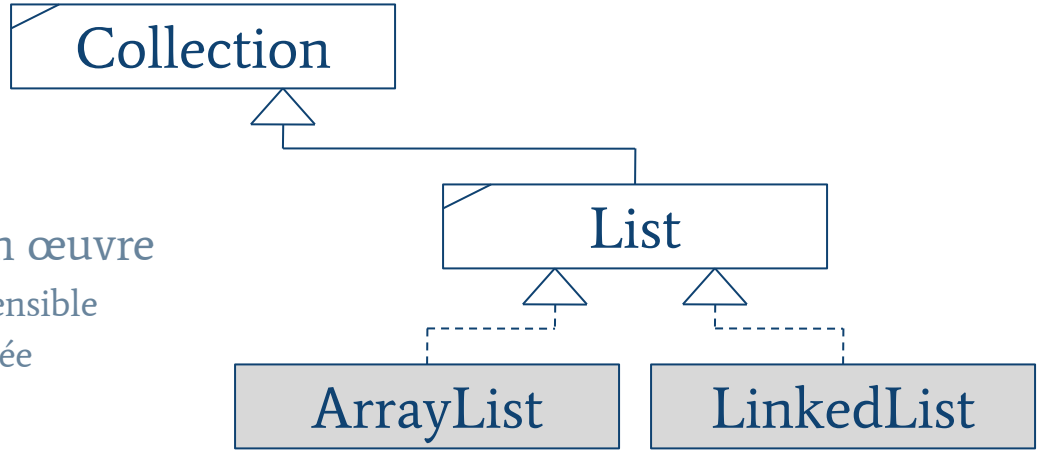
L'interface `List` (1/2)



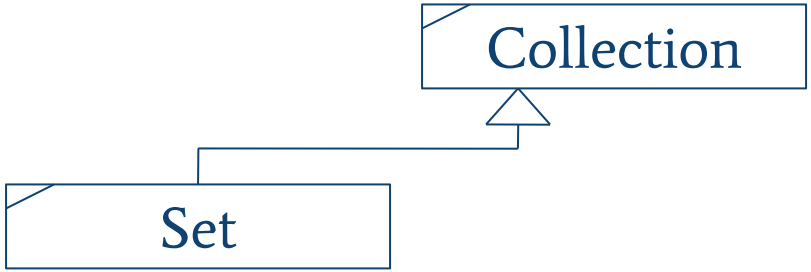
- Propriétés :
  - Les éléments possèdent un indice ⇒ généralisation des tableaux
  - Duplication d'éléments autorisée
- Ajoute principalement quatre nouvelles méthodes
  - `void add(int i, E e)`
  - `void set(int i, E e)`
  - `E get(int i)`
  - `E remove(int i)`

L'interface `List` (2/2)

- Deux principales mises en œuvre
  - `ArrayList` : tableau extensible
  - `LinkedList` : liste chaînée

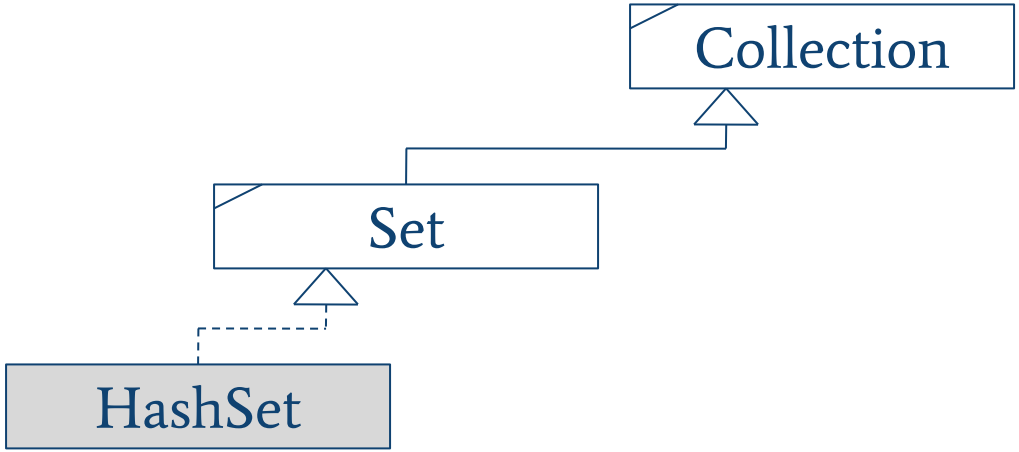


L'interface `Set` (1/2)



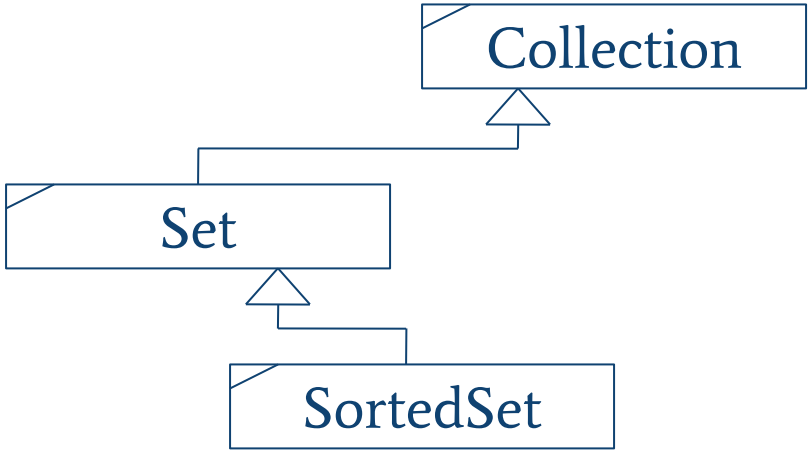
- Propriétés :
  - Les éléments ne possèdent pas d'indice
  - Duplication d'éléments interdite
- Pas de nouvelle méthode ajoutée

L'interface `Set` (2/2)



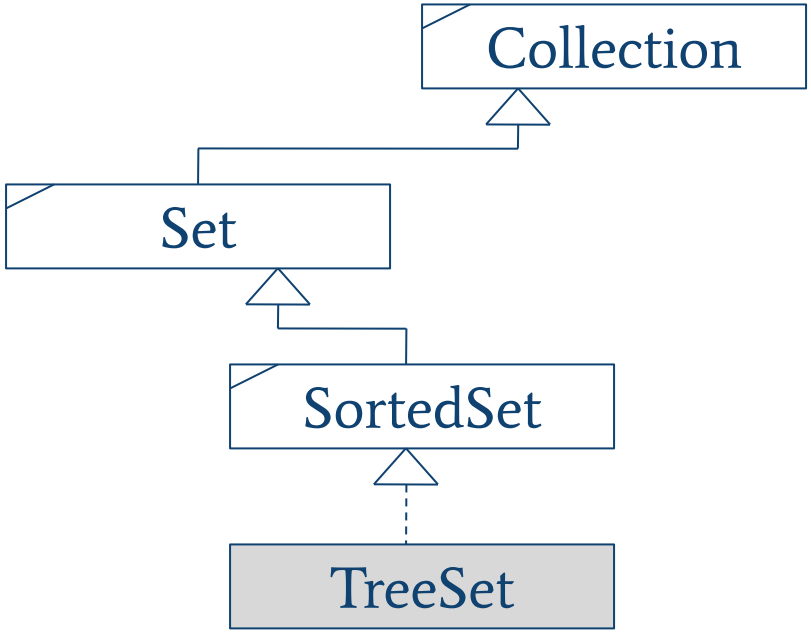
- `HashSet` = table de hachage dans laquelle clé == valeur
  - Les éléments doivent mettre en œuvre `equals` et `hashCode`
  - Les éléments sont non triés

L'interface `SortedSet` (1/2)



- Propriétés :
  - Les éléments sont triés (mettent en œuvre `Comparable`)
- Pas de nouvelle méthode ajoutée

L'interface `SortedSet` (2/2)



- `TreeSet` = arbre binaire rouge-noir dans lequel clé == valeur

Complexités des collections de Java

Structure	add(el)	add(idx, el) put(key, val)	get(idx/ key)	remove(el)	remove(0) it.remove()	contains(el/ key)	next
ArrayList	$\mathcal{O}(1)$ *or* $\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
LinkedList	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
HashMap	X	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	X	$\mathcal{O}(1)$	$\mathcal{O}(h/n)$
TreeMap	X	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	X	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$
HashSet	$\mathcal{O}(1)$	X	X	$\mathcal{O}(1)$	X	$\mathcal{O}(1)$	$\mathcal{O}(h/n)$
TreeSet	$\mathcal{O}(log(n))$	X	X	$\mathcal{O}(log(n))$	X	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$

h = capacité = taille de la table interne (c.f. CI 7)

Choisir la bonne structure de donnée

- Les complexités nous donnent des indications générales sur les structures à choisir en fonction de nos besoins:
  - Je veux souvent regarder si un élément fait partie de ma structure : Set
  - Je veux pouvoir rapidement parcourir tous les éléments : List
  - Je veux associer une clef et une valeur : Map
- Cependant, les complexités restent approximatives et ne garantissent pas les performances temporelles en pratique dans tous les cas. Il faut donc:
  - Comprendre les implémentations. Par exemple, `HashSet` n'est pas thread-safe, alors que `ConcurrentSkipListSet` si. `LinkedHashSet` a de meilleures performances que `HashSet` si l'on doit parcourir le set... La Javadoc est très utile pour cela !
  - Mesurer les performances réelles de votre programme (profiling). On chronomètre des tests en fonction de la structure utilisée. L'encapsulation est primordial ici !

Exemple de profiling

```
static int maxElements = 10000;
static int nbExperiments = 10000;

static double testAdd(List<Integer> l){
        double start = System.nanoTime();
        for (int i = 0; i < maxElements; i++){
                l.add(i);
        }
        double time = System.nanoTime() - start;
        return time / maxElements;
}

static double testRemove(List<Integer> l){
        double start = System.nanoTime();
        for (int i = 0; i < maxElements; i++){
                l.remove(0);
        }
        double time = System.nanoTime() - start;
        return time / maxElements;
}
```

```
public static void main(String[] args) {
        double totalAddAL = 0;
        double totalAddLL = 0;
        double totalRemoveAL = 0;
        double totalRemoveLL = 0;
        for (int i = 0; i < nbExperiments; i++){
                ArrayList<Integer> arrayList = new ArrayList<>();
                LinkedList<Integer> linkedList = new LinkedList<>();
                totalAddAL += testAdd(arrayList);
                totalAddLL += testAdd(linkedList);
                totalRemoveAL += testRemove(arrayList);
                totalRemoveLL += testRemove(linkedList);
        }
        totalAddAL /= nbExperiments;
        totalAddLL /= nbExperiments;
        totalRemoveAL /= nbExperiments;
        totalRemoveLL /= nbExperiments;
        System.out.println("Average time to add element in ArrayList: "
                        + totalAddAL);
        System.out.println("Average time to add element in LinkedList: "
                        + totalAddLL);
        System.out.println("Average time to remove first element in
ArrayList: " + totalRemoveAL);
        System.out.println("Average time to remove first element in
LinkedList: " + totalRemoveLL);
}
```

Exemple de profiling

```
public static void main(String[] args) {
        double totalAddAL = 0;
        double totalAddLL = 0;
        double totalRemoveAL = 0;
        double totalRemoveLL = 0;
        for (int i = 0; i < nbExperiments; i++){
```

```
static i
static i

static d
```

```
}

static d
```

```
Average time to add element in ArrayList: 5.076008439999996
Average time to add element in LinkedList: 5.5253209499999745
```

Dans notre cas, les `ArrayList` sont 10% plus rapides, malgré une complexité similaire.

```
Average time to remove first element in ArrayList: 146.90044398000063
Average time to remove first element in LinkedList: 3.127785470000008
```

Pour des complexités différentes, les différences sont flagrantes :
`LinkedList` est 47 fois plus rapide !

```
        double time = System.nanoTime() - start;
        return time / maxElements;
}
```

```
totalRemoveAL);
        System.out.println("Average time to remove first element in
LinkedList: " + totalRemoveLL);
}
```

Le profiling est plus compliqué que ça...

- Beaucoup d'impacts extérieurs au programme :
  - Hardware/OS de l'ordinateur
  - Autres applications tournent en parallèle ?
  - Utilisation aléatoire du garbage collector
  - ...
- Mise en œuvre non triviale :
  - Écrire des tests pour tous les cas possibles d'utilisations
  - Dépendances de performances entre les fonctions
  - ...
- Des outils avancés existent (non vus dans ce cours)