ITI 1521. Introduction à l'informatique II

Liste : traitement itératif des listes chaînées.

by

Marcel Turcotte

Version du 15 mars 2020

Préambule

Préambule

Aperçu

Aperçu

Liste : traitement itératif des listes chaînées.

Nous comparons le temps de calcul nécessaire pour traverser une liste chaînée lorsque les énoncés ont accès aux noeuds de la liste par rapport à l'implémentation à l'aide des méthodes de l'interface de la liste. Nous explorons une implémentation efficace sans accéder aux noeuds de la liste directement.

Objectif général :

- Cette semaine, vous serez en mesure d'expliquer et d'utiliser un itérateur.

Avant-propos

- Les sujets abordés dans ce module renforceront les notions d'**encapsulation** et de **programmation orientée objet**, notamment la notion d'**état** de l'objet, ainsi que les **interfaces**.
- C'est aussi l'occasion d'introduire informellement la **complexité du calcul** (**analyse asymptotique**) qui vous sera présentée dans le cours de structures de données.

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Comparer** le temps de calcul nécessaire pour traverser une liste chaînée, discutez le cas où les énoncés ont accès aux noeuds de liste par rapport à l'implémentation n'ayant accès qu'aux méthodes de son interface.
- **Comparer** les classes imbriquées statiques et non statiques de Java.
- **Modifier** l'implémentation d'un itérateur afin d'y ajouter une méthode.

Lectures :

- Pages 89-96, 103-112 de E. Koffman et P. Wolfgang.

Préambule

Plan du module

Plan

Motivation

Motivation

Problème

Motivation

- Vous devez concevoir une méthode pour **traverser** une **liste chaînée**.

Motivation

Détails

Détails

- Nous travaillons avec une implémentation **simplement chaînée** de l'interface **List**.

```
public interface List<E> {
    boolean add(E element);
    E get(int index);
    boolean remove(E element);
    int size();
}
```

- Les difficultés seraient les mêmes si la liste était **doublement chaînée**.
- Nous nommerons cette implémentation **LinkedList**.

Un exemple de liste

```
List<String> colors;
colors = new LinkedList<String>();

colors.add("bleu");
colors.add("blanc");
colors.add("rouge");
colors.add("jaune");
colors.add("vert");
colors.add("orange");
```

Motivation

Implémentation interne

Implémentation A : à l'intérieur de la classe LinkedList

- À l'intérieur de la classe **LinkedList**, nous avons accès aux détails de l'implémentation. En particulier, nous avons **accès aux noeuds**.
  - **Donnez une implémentation :**

Motivation

Implémentation externe

Implémentation B : à l'extérieur de la classe LinkedList

- À l'extérieur de la classe **LinkedList**, nous n'avons pas accès aux détails de l'implémentation. En particulier, nous **n'avons pas accès aux noeuds**.
  - **Donnez une implémentation :**

Remarque

- De l'extérieur de la classe **LinkedList**, nous devons utiliser **E get(int pos)** afin d'accéder aux éléments de la liste.

Motivation

Temps de calcul

Discussion

- **Comparez** le **temps d'exécution** des deux implémentations (intérieur et extérieur).
  - L'implémentation de l'**intérieur** est-elle plus **rapide** ou plus **lente** ?
  - Les **différences** sont **mineures** ou **majeures** ?

Temps de calcul

Voici les **temps d'exécution** en **nanosecondes** pour des listes de longueur croissante.

# noeuds	A	B
20 000	73 214	523 248 106
40 000	138 208	2 054 870 866
80 000	277 909	8 430 799 795
160 000	671 434	36 546 381 116
320 000	1 461 222	157 744 738 581
640 000	3 428 519	655 822 468 389
1 280 000	5 922 119	**45 minutes !**

Pour 1 280 000 éléments, il faut environ **45 minutes** pour traverser la liste à l'aide d'appels à **get(pos)**, alors qu'il suffit **5.92 millisecondes** pour l'approche **A**.

Motivation

Discussion

Discussion

- **Comment** expliquer cette différence ?
- Pour chaque implémentation, quelle **relation mathématique** y a-t-il entre le nombre d'éléments dans la liste **n** et le **temps de calcul** ?
  - **Complétez la phrase :** chaque fois que le nombre d'éléments **n** double, le **temps de calcul** ...

- **Donnez** l'implémentation de la méthode **E get(int pos)**.
- Ainsi, l'implémentation **B**

```
for (int i=0; i < colors.size(); i++) {
    System.out.println(colors.get(i));
}
```

- Est **équivalente à ceci** :

Nombre de noeuds visités

Appel	# de noeuds visités
get(0)	1
get(1)	2
get(2)	3
get(3)	4
...	
get(n-1)	n

Motivation

Conclusion

Discussion

- Implémentation **A** visite n noeuds.
- Implémentation **B** visite n^2 noeuds !

Concept

Concept

Objectif

Concept

Objectif : Concevoir une approche afin de traverser la liste **une et une seule fois**.

- L'utilisateur de la liste n'aura pas accès à l'implémentation (**p.next** et autres) !
- La solution proposée sera applicable dans un contexte bien spécifique, **lorsque tous les noeuds de la liste sont visités de façon séquentielle**.
- **Ce n'est pas une solution générale pour accélérer get(i)**.

Itérateur

- L'itérateur est mécanisme **uniforme** et **général** afin de traverser une variété de structures de données, telles que les listes, mais aussi les arbres et autres (voir CSI2510) ;
- Donne accès aux éléments **un élément à la fois** ;
- Fait partie des collections de Java.

Concept

Interface

Interface Iterator

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Discussion : implémentation

- Quelle classe implémentera **Iterator** ?
- Comment **créer** et **initialiser** un itérateur ?
- Comment **déplacer** l'itérateur ?
- Comment **détecter la fin** de l'itération ?

Implémentation 1.0

- Développons une première implémentation qui sera assez **différente** de l'**implémentation finale**.
- Elle sera cependant une bonne **étape intermédiaire**.
- La classe **LinkedList** implémente l'interface **Iterator**.

Implémentation 1.0

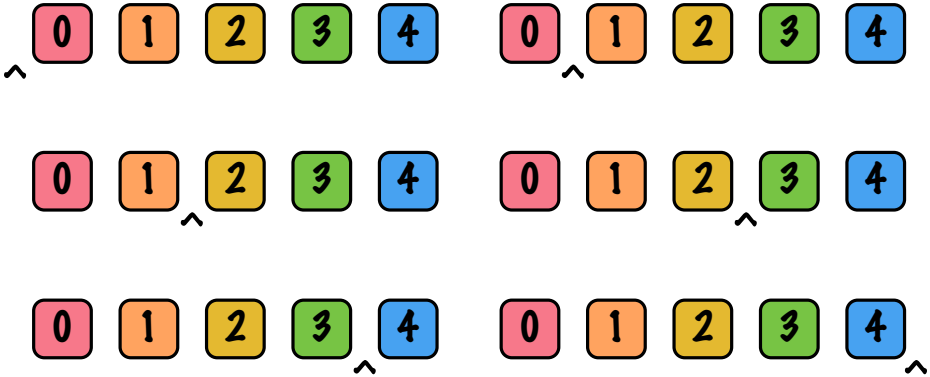
Implémentation 1.0

```
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;

    // ...

    public E next() { ... }

    public boolean hasNext() { ... }

}
```

Implémentation 1.0

Exemple

Contrat

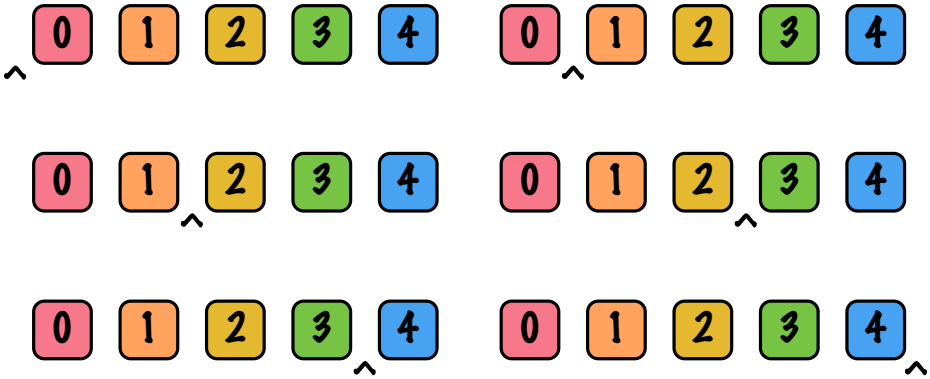


- Conceptuellement, l'itérateur est **à gauche du premier élément** au début de l'itération.
- Lors d'un appel à la méthode **next** :
  1. L'itérateur se **déplace** vers l'avant ;
  2. **Retourne** la valeur de l'élément visité.
- Un appel à la méthode **next** lorsque la liste est vide ou en fin d'itération (lorsque

Exemple

```
List<Integer> l;
l = new LinkedList<Integer>();

for (int i=0; i<5; i++) {
    l.add(new Integer(i));
}

int sum = 0;

while (l.hasNext()) {
    Integer v = l.next();
    sum += v.intValue();
}

System.out.println("sum = " + sum);
```

Exemple

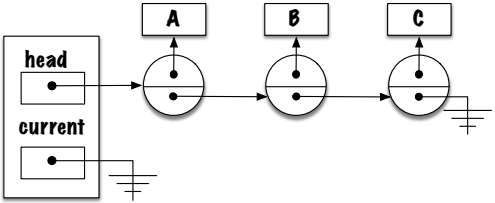



```
int sum = 0;

while (l.hasNext()) {
    Integer v = l.next();
    sum += v.intValue();
}
```

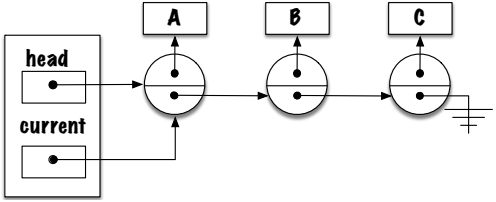
Implémentation 1.0

Discussion

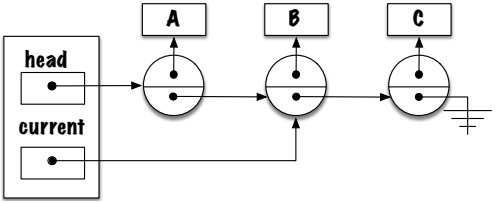
- Quelles sont les **variables d'instance** nécessaires ?
- Quel est le **type** de la variable **current** ?
- Quelle sera la **valeur** initiale de **current** ?
- Lors du premier appel,
- Pour chaque appel subséquent,



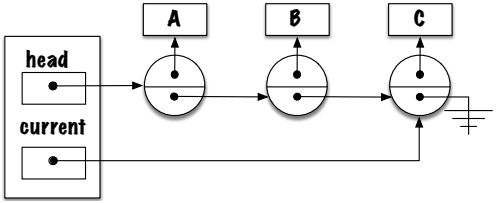
Avant l'itération



suite à .next()



suite à .next()



suite à .next()

Implémentation 1.0

Variable d'instance

Implémentation 1.0

```
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;
    private Node<E> current;

    // ...

    public E next() { ... }

    public boolean hasNext() { ... }

}
```

Implémentation 1.0

next

Implémentation 1.0

```
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;
    private Node<E> current;

    public E next() {









    }

}
```

Implémentation 1.0

Implémentation 1.0

hasNext

Implémentation 1.0

```
public class LinkedList<E> implements List<E>, Iterator<E> {

    private static class Node<E> { ... }

    private Node<E> head;
    private Node<E> current;

    public boolean hasNext() {









    }

}
```

Implémentation 1.0

Implémentation 1.0

Discussion

Discussion

- Quelle la plus grande **restriction** de notre implémentation ?
- Que faut-il afin de palier à cette **limitation** ?

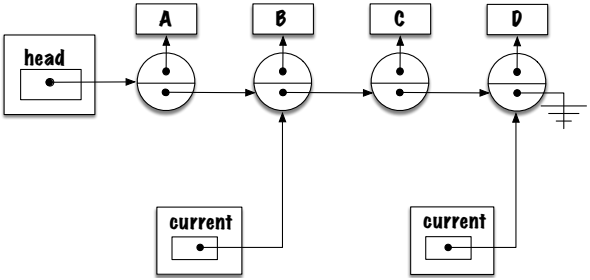
Implémentation 2.0

Implémentation 2.0

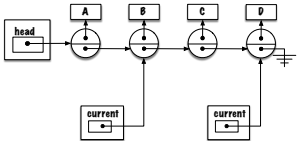
Diagramme de mémoire

Diagramme de mémoire

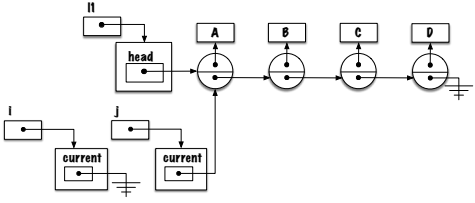
- **Discutez** ce diagramme de mémoire.

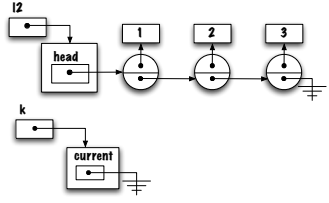


Discussion



- **LinkedList** n'implémente pas l'interface **Iterator**.
- L'itérateur est un objet qui possède une variable d'instance, **current**, de type **Node<E>**.
- Autant d'itérateurs qu'il en faut.
- L'itérateur doit avoir **accès aux éléments** de la liste.
- Une classe de **premier niveau** n'aurait pas accès aux éléments.
- **Suggestions ?**
- C'est ça, l'itérateur est une **classe imbriquée**.



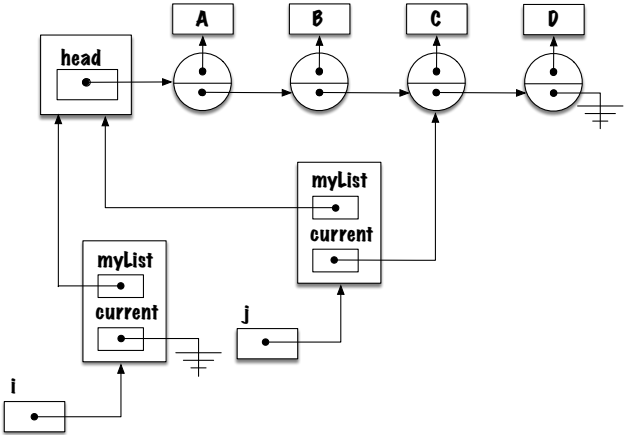


- Un **itérateur** doit appartenir à liste donnée.
- Un **itérateur** doit accéder à la variable **head** de sa liste.

```
public E next() {
    if (current == null) {
        current = head;
    } else {
        current = current.next;
    }
    if (current == null) {
        throw new NoSuchElementException();
    }
    return current.value;
}
```

Diagramme de mémoire

- **Discutez** ce diagramme de mémoire.



Implémentation 2.0

Variables d'instance et constructeur

```
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        private ListIterator(LinkedList<E> myList) {
            this.myList = myList;
            current = null;
        }

        public boolean hasNext() { ... }

        public E next() { ... }
    }

    private Node<E> head;

}
```

Implémentation 2.0

next

```
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        public E next() {
            if (current == null) {
                current =                    ;
            } else {
                current = current.next;
            }
            if (current == null) {
                throw new NoSuchElementException();
            }
            return current.value;
        }

        public boolean hasNext() { ... }
    }

    private Node<E> head;
}
```

hasNext

```
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        public E next() { ... }

        public boolean hasNext() {
            if (current == null &&              != null) {
                return true;
            } else if (current != null && current.next != null) {
                return true;
            } else {
                return false;
            }
        }
    }

    private Node<E> head;
}
```

Implémentation 2.0

iterator

```
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private static class ListIterator<E> implements Iterator<E> {

        private Node<E> current;
        private LinkedList<E> myList;

        private LinkedListIterator(LinkedList<E> myList) {
            this.myList = myList;
            current = null;
        }
        public E next() { ... }
        public boolean hasNext() { ... }

    }

    public Iterator<E> iterator() {


    }

    private Node<E> head;
}
```

Implémentation 2.0

Exemple

```java
LinkedList<Integer> l;
l = new LinkedList<Integer>();

// ...

Iterator<Integer> i;
i = l.iterator();

while (i.hasNext()) {

    Integer v1 = i.next();

    Iterator<Integer> j;
    j = l.iterator();

    while (j.hasNext()) {

        Integer v2 = j.next();

        System.out.println("("+v1+","+v2+")");
    }
}
```

Implémentation 2.0

Temps de calcul

Est-ce c'est rapide ?

Voici les **temps d'exécution** en **nanosecondes** pour des listes de longueur croissante.

# noeuds	Intérieur	Itérateur
20 000	73 214	113 817
40 000	138 208	167 639
80 000	277 909	324 540
160 000	671 434	758 642
320 000	1 461 222	1 760 357
640 000	3 428 519	3 717 519
1 280 000	5 922 119	7 239 676

Pour 1 280 000 éléments, le temps de calcul est de **7.2 millisecondes**, à peine **13 %** plus lent que l'implémentation ayant accès aux éléments !

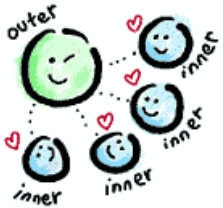
Implémentation 3.0

Implémentation 3.0

Classe interne

«Getting in Touch with your Inner Class»

- www.javaranch.com/campfire/StoryInner.jsp



Définition

Une **classe interne** (en anglais «***inner** class*») est une classe imbriquée non «static».

- Un objet d'une classe interne (non static) a **accès aux variables** et **méthodes** de l'objet de la classe externe à partir duquel il a été créé.

Implémentation 3.0

next

Implémentation 3.0

hasNext

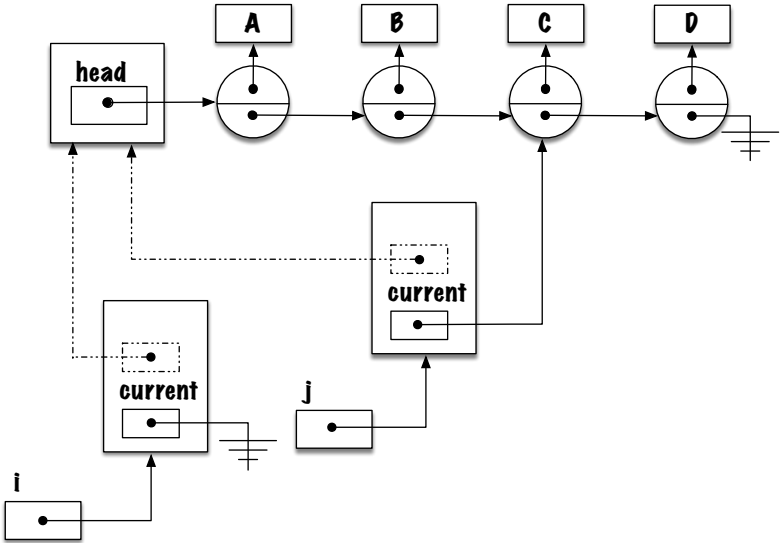
Implémentation 3.0

iterator

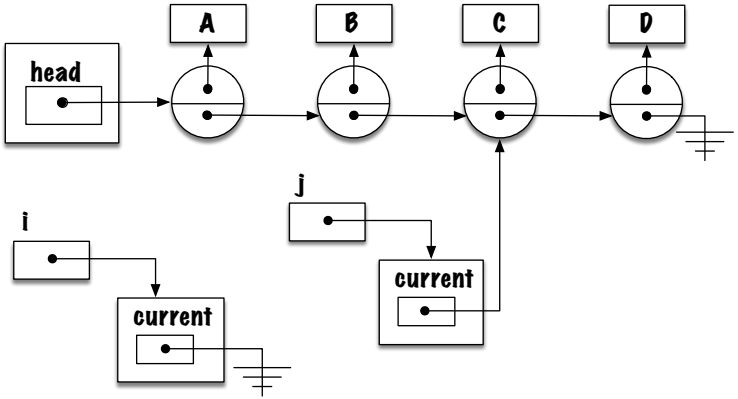
Implémentation 3.0

Diagramme de mémoire

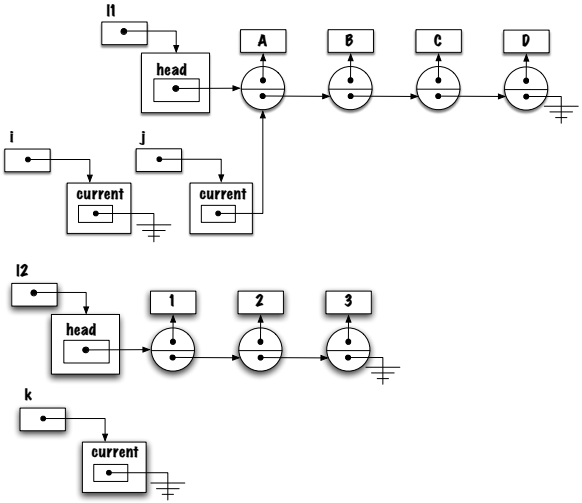
Classe interne



Classe interne

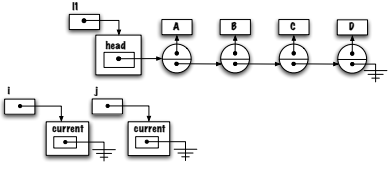


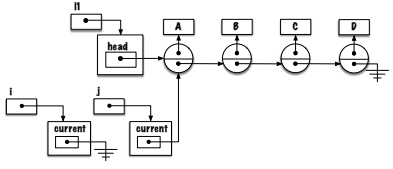
Exemple

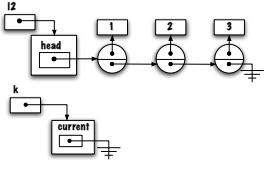


Implémentation 3.0

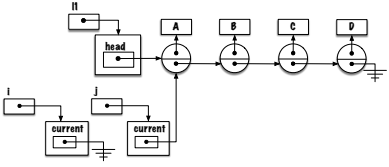
Exemple

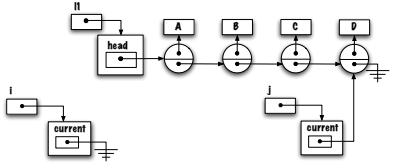


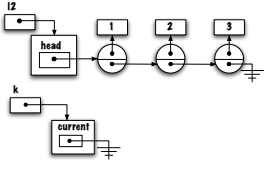




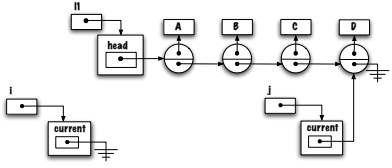

```
if (j.hasNext()) {
    String o = j.next();
}
```

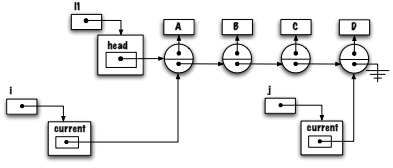


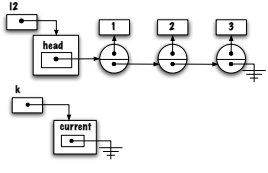




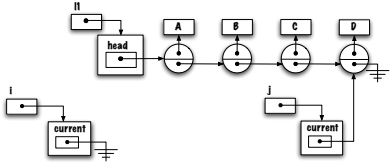

```
while (j.hasNext()) {
    String o = j.next();
}
```

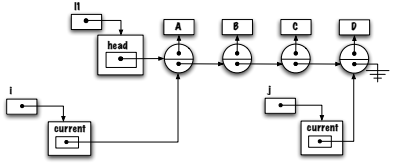


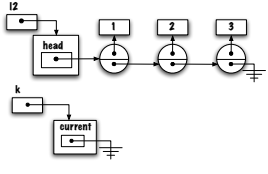


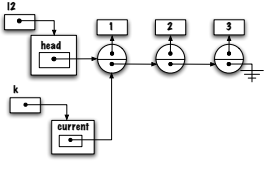



```
if (i.hasNext()) {
    String o = i.next();
}
```

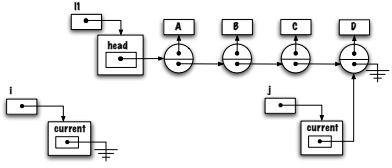


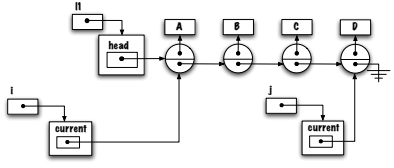


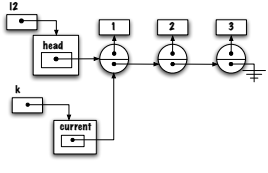


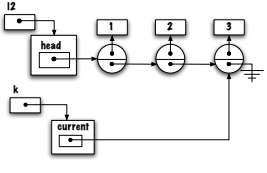



```
if (k.hasNext()) {
    Integer o = k.next();
}
```










```
while (k.hasNext()) {
    Integer o = k.next();
}
```

Temps de calcul

Voici les **temps d'exécution** en **nanosecondes** pour des listes de longueur croissante.

# noeuds	Intérieur	Itérateur	Get
10 000	43 508	66 849	1.118841e+08
20 000	49 233	66 986	4.619370e+08
40 000	99 714	108 464	1.873445e+09
80 000	240 057	252 130	8.404544e+09
160 000	592 818	615 779	2.892314e+10
320 000	1 039 555	1 142 309	1.401875e+11
640 000	2 328 335	2 448 321	6.258633e+11
1 280 000	5 124 979	4 896 708	2.753671e+12
2 560 000	11 500 576	11 700 579	1.476815e+13

- Pour 2 560 000 éléments, **get(pos)** est **1 million de fois plus lent** que l'itérateur ! 1.48e+13 ns = 4.1 heures.

Prologue

Résumé

- L'itérateur est un mécanisme permettant de **traverser** une liste **un élément à la fois**.
- La méthode **hasNext** retourne **true** si un appel à la méthode **next** est possible.
- La méthode **next** retourne le prochain élément dans l'itération.
- Une classe **interne** est une classe imbriquée non static.
- Les objets des classes internes ont **accès** aux **variables** et **méthodes** de la classe externe.

Prochain module

- **Listes** : traitement récursif

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa