

Initiation à la programmation Java

IP2 - Séance No 6

Yan Jurski

Discussion : comparaison Tableaux vs Listes

Ces deux types permettent de manipuler des ensembles de valeurs.

Pourquoi utiliser l'un ou l'autre ? Avantages, inconvénients ?

- Tableaux :
 - taille fixe (complicque les opérations d'ajouts suppressions)
 - + accès direct à un élément lorsque la position est connue
- Listes chaînées :
 - mise en place initiale un peu lourde (3 classes)
 - précautions lors de la gestion des liaisons entre cellules
 - + structure plus adaptée aux opérations classiques

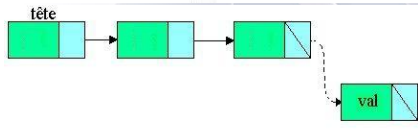
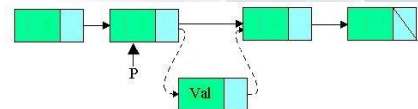
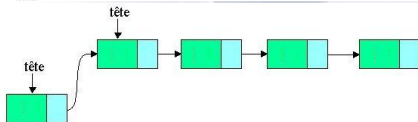
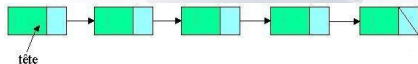
L'utilisateur choisira son implémentation en évaluant coût/gain

Variations autour des listes

Plan de cette séance

- Question : ce qui change avec l'ajout d'une seconde liaison ?
 - coût de gestion supplémentaire souvent faible
 - simplification de certaines opérations précédemment complexes
- L'utilisateur choisira son implémentation en évaluant coût/gain
- Cas envisageables :
 - Listes simplement chaînées avec référence sur début et fin
 - Listes simplement chaînées circulaires
 - Listes doublement chaînées avec référence sur début
 - Idem avec référence sur début et fin
 - Avec une tête de lecture
 - Piles, Files
 - Tout le long de ce cours : solutions récursives ou itératives

Rappels : opérations sur les listes simplement chaînées



Exemple

Ajout / Suppression en tête

Ajout / Suppression intermédiaire

Ajout / Suppression en fin

Rappel du cahier des charges

que vous connaissez bien...

Fichier ListIP2.java

```
public interface ListIP2 {  
    void clear();  
    boolean isEmpty();  
  
    // méthodes de parcours  
    int size();  
    boolean contains(E x);  
    E get(int index);  
    E set(int index, E x);  
    int indexOf(E x);  
    int lastIndexOf(E x);  
  
    // méthodes qui modifient la structure de la liste  
    void add(E x); // ajoute en fin  
    void add(int index, E x);  
    E remove(int index);  
    boolean remove(E x);  
}
```

Rappel du modèle à 3 classes

Fichier MaList.java

```
public class MaList implements ListIP2{  
    private Cellule first;  
}
```

Fichier Cellule.java

```
public class Cellule{  
    private E content;  
    private Cellule next;  
}
```

Fichier E.java - le contenu

```
public class E{  
    // peu importe  
}
```

Rappel du modèle à 3 classes



Première variation : observateur sur le dernier



Première variation : observateur sur le dernier

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{  
    private CelluleFL first;  
    private CelluleFL last;  
}
```

Fichier CelluleFL.java

```
public class CelluleFL{  
    private E content;  
    private CelluleFL next;  
}
```

Fichier E.java - le contenu

```
public class E{  
    // peu importe  
}
```

Ancienne version

Construction de la liste vide

Fichier MaList.java

```
public class MaList implements ListIP2{
    private Cellule first;
    public MaList(){
        this.first=null;
    }
    public boolean isEmpty() {
        return ( this.first==null );
    }
    public void clear() {
        this.first=null;
    }
}
```

Variante avec Début et Fin

liste vide, test et reset

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public MaList(){
        this.first=null;
        this.last=null;
    }
    public boolean isEmpty() {
        return ( this.first==null ); // suffisant, sinon on s'est trompé ...
    }
    public void clear() {
        this.first=null;
        this.last=null;
    }
}
```

Fichier ListIP2.java

```
public interface ListIP2 {  
    ...  
    int size();  
    boolean contains(E x);  
    E get(int index);  
    E set(int index, E x);  
    int indexOf(E x);  
    int lastIndexOf(E x);  
    ...  
}
```

- La nouvelle liaison vers le dernier :
 - ne nous empêche pas d'utiliser l'ancienne
 - ne semble pas permettre d'améliorations

Ancienne Version

Exemple de parcours

Fichier MaList.java

```
public class MaList implements ListIP2{
    private Cellule first;
    public int size(){
        if (this.isEmpty()) return 0;
        else return first.size(); // on responsabilise
    }
}
```

Fichier Cellule.java

```
public class Cellule{
    private E content;
    private Cellule next;
    public int size(){
        if (next==null) return 1;
        return 1+next.size();
    }
}
```

Variante avec Début et Fin

Exemple de parcours - Mêmes algorithmes

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public int size(){
        if (this.isEmpty()) return 0;
        else return first.size(); // on responsabilise
    }
}
```

Fichier CelluleFL.java

```
public class CelluleFL{
    private E content;
    private CelluleFL next;
    public int size(){
        if (next==null) return 1;
        return 1+next.size();
    }
}
```

Méthodes qui modifient la structure de la liste

...last devra être modifié aussi ... parfois ...

Fichier ListIP2.java

```
public interface ListIP2 {  
    ...  
    void add(E x); // ajoute en fin  
    void add(int index, E x);  
    E remove(int index);  
    boolean remove(E x);  
}
```

Ancienne version : ajout en fin

Fichier MaList.java

```
public class MaList implements ListIP2{
    private Cellule first;
    public void add(E x){
        if (this.isEmpty()) first=new Cellule(x); else first.add(x);
    }
}
```

Fichier Cellule.java

```
public class Cellule{
    public Cellule(E x){
        this.content=x;
        next=null;
    }
    public void add(E x){
        Cellule tmp=this; // tout un mécanisme pour trouver le dernier
        while(tmp.next != null) tmp=tmp.next;
        tmp.next=new Cellule(x);
    }
}
```


Variante avec Début et Fin

Fichier MaListFirstLast.java

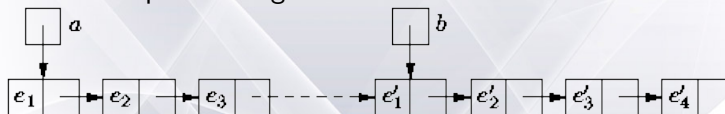
```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public void add(E x){
        CelluleFL aux=new CelluleFL(x);
        if (this.isEmpty()) { first= aux; last=aux;}
        else { last.setNext(aux); last=aux;} // opération en temps constant
    }
}
```

Fichier CelluleFL.java

```
public class CelluleFL{
    public CelluleFL(E x){
        this.content=x;
        next=null;
    }
    public void setNext(CelluleFL c){ next=c; }
}
```

Avantage de la variante

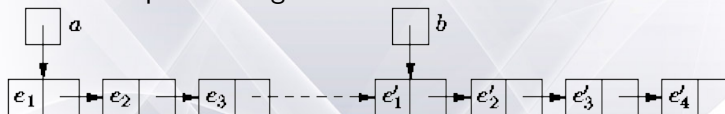
- Lorsque les opérations portent sur le dernier
 - Notre variante l'a immédiatement à disposition
 - La première version doit, elle, le recalculer
- Autre exemple avantageux : la concaténation



- Dans notre première implémentation il faut :
 - trouver le dernier élément de la liste a
 - le relier au premier élément de la liste b
 - ne pas oublier les cas limites (a ou b vide)

Avantage de la variante

- Lorsque les opérations portent sur le dernier
 - Notre variante l'a immédiatement à disposition
 - La première version doit, elle, le recalculer
- Autre exemple avantageux : la concaténation



- Dans notre première implémentation il faut :
 - trouver le dernier élément de la liste a
 - le relier au premier élément de la liste b
 - ne pas oublier les cas limites (a ou b vide)

Faisons le en exercice, dans un contexte dynamique (a est `this`)

Fichier MaList.java

```
public class MaList implements ListIP2{
    private Cellule first;
    public void concat(MaList b){ // ajoute b en fin de la liste courante
        if (this.isEmpty()) first=b.first;
        else first.concat(b.first); // opération déléguée aux Cellules
    }
}
```

Fichier Cellule.java

```
public class Cellule{
    private E content;
    private Cellule next;
    public void concat(Cellule x){ // x peut être null, peu importe
        Cellule tmp = this;
        while ( tmp.next != null ) tmp=tmp.next;
        tmp.next=x;
    }
}
```

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public void concat(MaList b){ // ajoute b en fin de la liste courante
        if (this.isEmpty()) {
            this.first=b.first;
            this.last=b.last;
        }
        else {
            this.last.setNext(b.first);
            if (b.last != null) this.last=b.last;
        }
    }
}
```

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public void concat(MaList b){ // ajoute b en fin de la liste courante
        if (this.isEmpty()) {
            this.first=b.first;
            this.last=b.last;
        }
        else {
            this.last.setNext(b.first);
            if (b.last != null) this.last=b.last;
        }
    }
}
```

- Et c'est tout ! (on peut vérifier la robustesse sur b vide ou singleton)

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public void concat(MaList b){ // ajoute b en fin de la liste courante
        if (this.isEmpty()) {
            this.first=b.first;
            this.last=b.last;
        }
        else {
            this.last.setNext(b.first);
            if (b.last != null) this.last=b.last;
        }
    }
}
```

- Et c'est tout ! (on peut vérifier la robustesse sur b vide ou singleton)
- Si on veut être prudent sur le partage de cellules entre b et `this`, on peut ajouter à la fin de concat : `b.clear()`;

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public void concat(MaList b){
        if (this==null) { // question stupide (ERREUR)
            this=b; // ??? (ERREUR)
        }
        ...
    }
}
```

- Supposons (par l'absurde) que `this` puisse être `null`...
 - se poserait-il seulement la question ?
 - pourrait-il réagir ?
- Supposons (par l'absurde) que `vous (this)` puissiez devenir votre camarade ...
 - **Vous** feriez qq chose ensuite ?

Variante avec Début et Fin (suite)

Fichier ListIP2.java

```
public interface ListIP2 {  
    ...  
    void add(int index, E x);  
    E remove(int index);  
    boolean remove(E x);  
}
```

- pas d'autres avantages à tirer de la connaissance du dernier
- prix à payer : assurer la gestion correcte du champ last

Variante avec Début et Fin (suite)

Prix à payer - Exemple de remove

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public boolean remove(E x){
        if ( this.isEmpty() ) return false;
        if ( first.getContent()== x) {
            first=first.getNext();
            // dessiner le cas limite d'une liste à une seule cellule !
            ...
        }
    }
}
```

Variante avec Début et Fin (Suite)

Prix à payer - Exemple de remove

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public boolean remove(E x){
        if ( this.isEmpty() ) return false;
        if ( first.getContent()== x) {
            first=first.getNext();
            if (first==null) last=null;
            return true;
        }
        // autres cas
        ...
    }
}
```

Variante avec Début et Fin (Suite)

Prix à payer - Exemple de remove

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public boolean remove(E x){
        if ( this.isEmpty() ) return false;
        if ( first.getContent()== x) {
            first=first.getNext();
            if (first==null) last=null;
            return true;
        } else {
            return first.remove(x); // en toute logique on veut délèguer
            // mais comment savoir si last est impacté ?
        }
    }
}
```

- Tester si last contient x n'apporte rien (il faudrait le prédécesseur)
- On fait en sorte que remove de Cellule retourne le prédécesseur!



Variante avec Début et Fin (Suite)

Prix à payer - Exemple de remove

Fichier MaListFirstLast.java

```
public class MaListFirstLast implements ListIP2{
    private CelluleFL first;
    private CelluleFL last;
    public boolean remove(E x){
        if ( this.isEmpty() ) return false;
        if ( first.getContent()== x) {
            first=first.getNext();
            if (first==null) last=null;
            return true;
        } else {
            CelluleFL precedent_de_X = first.remove(x); // à écrire dans
                CelluleFL
            if (precedent_de_X==null) return false; // cahier des charges
            if (precedent_de_X.getNext()==null) this.last=precedent_de_X;
            return true;
        }
    }
}
```

Variante avec Début et Fin (Suite)

Prix à payer - Exemple de remove

Fichier CelluleFL.java

```
public class CelluleFL{
    private E content;
    private CelluleFL next;
    public CelluleFL remove(E x){ // appelée lorsque this.content != x
        // retourne le prédécesseur de x s'il existe, null sinon
        if ( next==null ) return null;
        if ( next.content !=x) return next.remove(x); // version récursive
        next=next.next; // retire
        return this; // retourne le prédécesseur de l'élément supprimé
    }
}
```

Variante avec Début et Fin (Bilan)

- L'introduction de la liaison vers la fin de liste :
 - Permet un gain important lors de :
 - l'ajout en fin
 - la concaténation
 - A un coût sur les autres opérations
 - Qui reste faible (même ordre de grandeur)
 - Mais le programmeur doit travailler très sérieusement !

Listes Chaînées - Application aux piles



Fichier PileIP2.java – Ecrite pour des éléments de type E

```
public interface PileIP2 {  
    void push(E x); // Empile l'élément  
    boolean isEmpty();  
    E top (); // retourne l'élément en haut de la pile, null si pile vide  
    E pop (); // même chose que top() mais en plus le retire de la pile  
}
```


Fichier PileIP2.java – Ecrire pour des éléments de type E

```
public interface PileIP2 {  
    void push(E x); // Empile l'élément  
    boolean isEmpty();  
    E top (); // retourne l'élément en haut de la pile, null si pile vide  
    E pop (); // même chose que top() mais en plus le retire de la pile  
}
```

- Une pile peut être vue comme un cas particulier des listes
- Puisque les opérations sont moins générales, autant les optimiser !

Fichier PileIP2.java – Ecrite pour des éléments de type E

```
public class MaPile implements PileIP2 {
    private ListeIP2 liste; // rq : on utilise l'interface comme type
    public MaPile(){
        this.liste=new... //MaListe() ou MaListeFirstLast()? On va y réfléchir
    }
    void push(E x){ // Empile l'élément
        liste.add(0,x);
    }
    boolean isEmpty() {
        return liste.isEmpty();
    }
    E top (){ // retourne l'élément en haut de la pile, null si pile vide
        return liste.get(0);
    }
    E pop (){ // même chose que top() mais en plus le retire de la pile
        return liste.remove(0);
    }
}
```

Listes Chaînées - Application aux piles

Fichier PileIP2.java – Ecrite pour des éléments de type E

```
public class MaPile implements PileIP2 {
    private ListeIP2 liste; // rq : on utilise l'interface comme type
    public MaPile(){
        this.liste=new... //MaListe() ou MaListeFirstLast()? On va y réfléchir
    }
    void push(E x){ // Empile l'élément
        liste.add(0,x);
    }
    boolean isEmpty() {
        return liste.isEmpty();
    }
    E top (){ // retourne l'élément en haut de la pile, null si pile vide
        return liste.get(0);
    }
    E pop (){ // même chose que top() mais en plus le retire de la pile
        return liste.remove(0);
    }
}
```

- Le lien vers last ne semble d'aucune utilité!

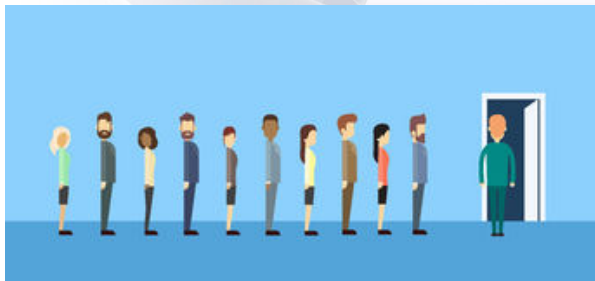
Listes Chaînées - Application aux piles

Fichier PileIP2.java – Ecrite pour des éléments de type E

```
public class MaPile implements PileIP2 {  
    private ListeIP2 liste;  
    public MaPile(){  
        this.liste= new MaListe(); // notre liste simplement chaînée  
    }  
    void push(E x){ // Empile l'élément  
        liste.add(0,x);  
    }  
    boolean isEmpty() {  
        return liste.isEmpty();  
    }  
    E top (){ // retourne l'élément en haut de la pile, null si pile vide  
        return liste.get(0);  
    }  
    E pop (){ // même chose que top() mais en plus le retire de la pile  
        return liste.remove(0);  
    }  
}
```

- Aucun parcours n'est requis. Les opérations ont un coût constant!

Listes Chaînées - Application aux files



[Link : autres exemples de Files ...](#)

Fichier FileIP2.java – Ecrite pour des éléments de type E

```
public interface FileIP2 {  
    void arrive(E x); // x se place en fin de file  
    boolean isEmpty();  
    E whoIsNext (); // retourne l'élément à servir, null si file vide  
    E getNext (); // même chose que whoIsNext(), en plus le retire  
}
```

Fichier FileIP2.java – Ecrite pour des éléments de type E

```
public class MaFile implements FileIP2 {
    private ListeIP2 liste;
    public MaFile(){
        this.liste= new ... // MaListe() ou MaListeFirstLast() ?
    }
    void arrive(E x){
        liste.add(x); // en fin de liste
    }
    boolean isEmpty() { return liste.isEmpty(); }
    E whoIsNext (); { return liste.get(0); }
    E getNext (); { return liste.remove(0); }
}
```

- L'ajout en fin peut être optimisé

Fichier FileIP2.java – Ecrite pour des éléments de type E

```
public class MaFile implements FileIP2 {  
    private ListeIP2 liste;  
    public MaFile(){  
        this.liste= new MaListeFirstLast();  
    }  
    void arrive(E x){  
        liste.add(x); // en fin de liste  
    }  
    boolean isEmpty() { return liste.isEmpty(); }  
    E whoIsNext (); { return liste.get(0); }  
    E getNext (); { return liste.remove(0); }  
}
```

- L'ajout en fin peut être optimisé

Fichier FileIP2.java – Ecrite pour des éléments de type E

```
public class MaFile implements FileIP2 {  
    private ListeIP2 liste;  
    public MaFile(){  
        this.liste= new MaListeFirstLast();  
    }  
    void arrive(E x){  
        liste.add(x); // en fin de liste  
    }  
    boolean isEmpty() { return liste.isEmpty(); }  
    E whoIsNext (); { return liste.get(0); }  
    E getNext (); { return liste.remove(0); }  
}
```

- L'ajout en fin peut être optimisé
- Aucun parcours n'est requis. Les opérations ont un coût constant !

Etude de cas - Enregistreur à bande



Etude de cas - Enregistreur à bande



Fichier Enregistreur.java

```
public interface Enregistreur {  
    void avanceRapide(); // à une extrémité  
    void retourRapide(); // à l'autre  
    void avance(int x); // avance de x pas  
    void recule(int x); // recule de x pas  
    E litInfo(); // de la tête de lecture  
    void ecritInfo(E x);  
}
```

Courage on a vu tout le nécessaire



- Tête de lecture
- Bidirectionnelle
- Opérations spécifiques pour atteindre les extrémités
- On ajoute : un mouvement au delà des limites sera créateur de ruban

Modèle à 3 classes

Fichier MonMagneto.java

```
public class MonMagneto implements Enregistreur{  
    private CelluleTape first;  
    private CelluleTape last;  
    private CelluleTape head;  
}
```

Fichier CelluleTape.java

```
public class CelluleTape{  
    private E content;  
    private CelluleTape next;  
    private CelluleTape previous;  
}
```

Fichier E.java - le contenu

```
public class E{  
    // peu importe  
}
```

Fichier MonMagneto.java

```
public class MonMagneto implements Enregistreur{
    private CelluleTape first;
    private CelluleTape last;
    private CelluleTape head;
    public MonMagneto(){ // pour commencer
        head=new CelluleTape(null); // dont le contenu est vide... à faire
        first=head; last=head;
    }
}
```

Fichier MonMagneto.java

```
public class MonMagneto implements Enregistreur{
    private CelluleTape first;
    private CelluleTape last;
    private CelluleTape head;
    public MonMagneto(){ // pour commencer
        head=new CelluleTape(null); // dont le contenu est vide... à faire
        first=head; last=head;
    }
    public void avanceRapide(){ head=last; }
    public void retourRapide(){ head=first; }
}
```

Fichier MonMagneto.java

```
public class MonMagneto implements Enregistreur{
    private CelluleTape first;
    private CelluleTape last;
    private CelluleTape head;
    public MonMagneto(){ // pour commencer
        head=new CelluleTape(null); // dont le contenu est vide... à faire
        first=head; last=head;
    }
    public void avanceRapide(){ head=last; }
    public void retourRapide(){ head=first; }
    public E litInfo(){
        return head.getContent(); // head n'est jamais null
    }
    public void ecritInfo(E x){
        head.setContent(x); // head n'est jamais null
    }
}
```

Les cas créateurs / destructeurs sont plus délicats

Ici seulement des créations

Fichier MonMagneto.java

```
public class MonMagneto implements Enregistreur{
    public void avance(int x){
        if (x<=0) return; // rien à faire
        // donc ici x >0
        if (head.getNext() == null) { // c'est qu'on est au bout du ruban
            head.setNext(new CelluleTape(null,head,null)); // à écrire ...
            last=head.getNext();
        } // tout est en place pour avancer
        head=head.getNext();
        this.avance(x-1); // solution réursive
    }
}
```

- void recule(int x) s'écrit symétriquement sans plus de difficultés

Les cas créateurs / destructeurs sont plus délicats

Ici seulement des créations

Fichier CelluleTape.java

```
public class CelluleTape {  
    private E content;  
    private CelluleTape next;  
    private CelluleTape previous;  
    public CelluleTape(E x){  
        content=x;  
        next=null;  
        previous=null;  
    }  
    public CelluleTape(E x, CelluleTape avant, CelluleTape après){  
        content=x;  
        previous=avant;  
        next=après;  
    }  
}
```

Bilan de l'étude de cas



- nous avons utilisé des cellules doublement chaînées
- ajouté les liens utiles pour optimiser le retour aux extrémités
- géré une politique particulière pour la création de cellules

Modèles plus exotiques

Listes Circulaires



- Pour les parcours, comparer les séquences à une cellule fixée comme repère

Exemple sur des cellules chaînées circulairement

Fichier CelluleCirculaire.java

```
public class CelluleCirculaire {
    private E content;
    private CelluleCirculaire next;
    ...
    public void affiche(){
        CelluleCirculaire aux=this;
        while (...) {
            System.out.println(aux.content);
            aux=aux.next; // Condition d'arrêt ? Ce ne sera pas null en tout cas
        }
    }
}
```

Exemple sur des cellules chaînées circulairement

Fichier CelluleCirculaire.java

```
public class CelluleCirculaire {  
    private E content;  
    private CelluleCirculaire next;  
    ...  
    public void affiche(){  
        CelluleCirculaire aux=this;  
        while (aux!=this) { // pb : c'est le cas dès le début  
            System.out.println(aux.content);  
            aux=aux.next;  
        }  
    }  
}
```

Exemple sur des cellules chaînées circulairement

Fichier CelluleCirculaire.java

```
public class CelluleCirculaire {  
    private E content;  
    private CelluleCirculaire next;  
    ...  
    public void affiche(){  
        CelluleCirculaire aux=this;  
        do {  
            System.out.println(aux.content);  
            aux=aux.next;  
        } while (aux!=this);  
    }  
}
```

- C'est maintenant qu'il faut fournir un travail. Vous avez encore un peu de temps !

Partiel

- Définitions de classes
- Méthodes statique/dynamique
- Modificateurs `static`, `final`, `private`, `public`
- Politique getters/setters, notion d'interface
- Listes simplement chaînées et opérations liées
- Récursion
- Autres formes de listes
- La semaine prochaine révisions en cours