Memo IP2

Version du 25 mars 2023

Chapitre 1

Rappels de cours

1.1 Java : les bases

1.1.1 `break` et `continue`

Le mot clé `break` permet d'arrêter une boucle. Toute instruction dans cette boucle qui se situe après ne sera pas utilisée.

```
for (int i = 0; i < 5; i++) {
  if (i == 2)
    break;
  System.out.println(i);
}
```

va afficher 0 puis 1.

À l'inverse le mot clé `continue` indique de passer au tour de boucle suivant.

```
for (int i = 0; i < 5; i++) {
  if (i == 2)
    continue;
  System.out.println(i);
}
```

va afficher les nombres de 0 à 4 sauf 2.

1.1.2 `return`

Le mot clé `return` permet de renvoyer une valeur.

Remarque 1. *Dans le cas d'une méthode `void` on peut utiliser `return;` pour sortir de la fonction.*

1.1.3 Incrémentation

La différence entre `i++` et `++i` est la valeur retournée.

```
int i = 2;
int j = i++; // i vaut 3 et j vaut 2
int k = 2;
int l = ++i; // k et l valent 3
```

Remarque 2. *Beaucoup de gens choisissent `++i` lorsqu'ils n'utilisent pas la valeur de retour. C'est parce que cette version est théoriquement la plus rapide*[1].

1. En pratique les compilateurs savent très bien comment traiter cela

1.2 Les classes

Une classe permet de regrouper plusieurs éléments (attributs).

```
public class Personne {
  // Attribut prenom de type String
  String prenom;
  // Attribut age de type entier
  int age;
}
```

Pour créer un objet, on a besoin d'un constructeur. Il est de la forme `Nom_De_Classe(arg1, arg2,...){}`. Par exemple,

```
public class Personne {
  ...

  public Personne(String p, int a) {
    this.prenom = p;
    this.age = a;
  }
}
```

Remarque 3. *Par défaut il existe un constructeur sans argument (`Personne()` ici) qui attribue une valeur par défaut aux attributs (0 aux entiers, `null` aux `String`,...). Cependant si on ajoute un constructeur à la main, alors ce constructeur par défaut est supprimé.*

Pour construire un objet, on utilise l'opérateur `new` :

```
public class Test {
  public static void main(String[] args) {
    Personne p1 = new Personne("Dark", 2);
    Personne p2 = new Personne("Vador", 3);
  }
}
```

Définition 1 (Instance d'une classe)**.** *Dans l'exemple précédent, chaque objet créé (`p1` et `p2`) sont des instances de la classe `Personne`.*

1.3 Les attributs *static*

Définition 2 (Attribut *static*)**.** *Un attribut d'une classe est static s'il est partagé entre toutes les instances d'une classe.*

```
public class Personne {
  String prenom;
  int age;
  static int nb_humains; // Un compteur du nombre d'humains sur Terre

  public Personne(String p, int a) {
    this.prenom = p;
    this.age = a;
    nb_humains++; // Un humain est créé, donc le nombre d'humains augmente
  }
}

public class Test {
  public static void main(String[] args) {
    Personne p1 = new Personne("Dark", 2);
    // Personne.nb_humains vaut 1
    Personne p2 = new Personne("Vador", 3);
    // Personne.nb_humains vaut 2
  }
}
```

Remarque 4. *Dans l'exemple précédent,* `p1.nb_humains` *donne le bon résultat. Cependant, ce n'est pas correct :*
- *d'un point du vue logique :* `nb_humains` *n'est pas liée à* `p1` *mais à la classe* `Personne` *;*
- *cela peut créer une ambiguïté :*

```
public interface A {
  public static int val;
}

public interface B {
  public static int val;
}

class C implements A, B {
}

public class Test {
  public static void main(String[] args) {
  C x = new C();
  System.out.println(x.val); // Error : The field x.val is ambiguous
  System.out.println(C.val); // Error : The field C.val is ambiguous
  System.out.println(A.val); // OK
  System.out.println(B.val); // OK
}
```

1.4 Les méthodes *static*

Définition 3 (Méthode *static*)**.** *Une méthode est static si elle n'est liée à aucune instance.*

En pratique cela signifie qu'elle :
- ne peut pas utiliser `this` ;
- doit avoir en argument l'objet [1] ;
- doit être appelée par `Nom_De_Classe.Nom_De_Methode`

```
public class Personne {
  ...

  public static void affiche(Personne p) {
    System.out.println(p.prenom + " (" + p.age + ")");
  }
}

public class Test {
  public static void main(String[] args) {
    Personne p1 = new Personne("Dark", 2);
    Personne.affiche(p1);
  }
}
```

1.5 Les méthodes non *static*

Les méthodes non *static* sont liées à une instance (`this`).

```
public class Personne {
  ...

  public void affiche() {
    System.out.println(this.prenom + " (" + this.age + ")");
  }
}
```

1. Si le but est d'accéder à des attributs non *static* de l'objet, s'il s'agit simplement d'un accès à une variable *static*, alors il n'y a pas d'objet à passer

```java

public class Test {
  public static void main(String[] args) {
    Personne p1 = new Personne("Dark", 2);
    p1.affiche();
  }
}
```

Remarque 5. *Dans le cas non static, il n'y a pas besoin de préciser que c'est la méthode `affiche` de la classe `Personne` car on sait que `p1` est une `Personne`.*

Remarque 6. *`this` ne peut jamais être `null`.*

1.6 Attribut *final*

Définition 4 (Attribut *final*)**.** *Un attribut est final si on ne peut plus changer sa valeur une fois qu'elle a été donnée.*

```java
public class Personne {
  String prenom;
  final String nom;
  int age;

  public Personne(String prenom, String nom, int age) {
    this.prenom = prenom;
    // On peut lui donner une valeur
    this.nom = nom;
    this.age = age;
  }

  public void changeNom(String nouveauNom) {
    // Ne fonctionne pas car nom est final
    this.nom = nouveauNom;
  }
}
```

1.7 Algorithmes utiles

1.7.1 Recherche du plus grand élément

Le but ici est de trouver la personne avec l'attribut `age` le plus élevé dans un tableau. Le principe est :
- de créer une variable `resultat` qui vaut `null` au départ
- de parcourir toutes les cases et regarder si la valeur est plus grande que le résultat actuel

Remarque 7. *Le tableau peut contenir `null`, accéder à un attribut de `null` est une erreur !*

```java
public class Personne {
  ...

  public static Personne plusAgee(Personne[] ps) {
    Personne resultat = null;
    for (int i = 0; i < ps.length; i++) {
      Personne actuelle = ps[i];
      if (actuelle == null)
        continue; // continue permet de passer au tour de boucle suivant directement
      if (resultat == null // Le cas où on n'a pas encore trouvé quelqu'un
        || resultat.age < actuelle.age) { // Cas où on a trouvé quelqu'un plus vieux
        resultat = actuelle;
      }
    }
    return resultat;
  }
}
```

Exercice 1. *Appliquez cet algorithme à la main sur le tableau* `{Personne("Dark", 3), null, Personne("Dark", 6)}`.

1.7.2 Ajout d'un identifiant

Le but ici est d'avoir un identifiant unique pour chaque instance d'une classe. Pour cela, on va avoir un compteur `static` qui va permettre de connaître le plus grand identifiant déjà créé.

```
1  public class Personne {
2    // Le premier identifiant sera 1
3    private static int cmpt = 1;
4    int id;
5
6    public Personne() {
7      this.id = cmpt;
8      Personne.cmpt++;
9    }
10 }
```

Remarque 8. *La valeur* `cmpt` *doit être privée et ne doit pas pouvoir être modifiée en dehors de la classe (donc pas de setter).*

Exercice 2. *On suppose qu'il existe un getter* `getCmpt` *qui renvoie la valeur de* `cmpt`*. Que va afficher le code suivant ?*

```
1  public static void main(String[] args) {
2    System.out.println(Personne.getCmpt());
3    Personne p1 = new Personne();
4    System.out.println(Personne.getCmpt());
5    Personne p2 = new Personne();
6    System.out.println(p1.id);
7    System.out.println(Personne.getCmpt());
8  }
```

1.8 Les listes simplement chaînées

Durant le cours, vous avez vu une manière d'implémenter des listes [2].

Avec cette implémentation, il y a deux classes :
- Une classe qui représente la liste (`Liste`)
- Une classe qui représente un élément de la liste (`Cellule`)

La première contient un pointeur vers le premier élément de la liste alors que la seconde contient la valeur et un pointeur vers la cellule suivante. Elles ont donc la forme suivante (pour stocker un objet de type `Animal`) :

Liste.java

```
1  public class Liste {
2    // Pointeur vers le premier élément de la liste.
3    // Il vaut null si la liste est vide.
4    private Cellule premier;
5  }
```

Cellule.java

```
1  public class Cellule {
2    // Pointeur vers l'élément suivant de la liste.
3    // Il vaut null si on est sur le dernier élément.
4    private Cellule suivante;
5    // Valeur contenue dans la Cellule.
6    private Animal valeur;
7  }
```

2. Il s'agit de listes chaînées, voir `https://fr.wikipedia.org/wiki/Liste_cha%C3%AEn%C3%A9e` pour plus de détails

Pour faciliter votre code, je vous recommande de faire deux constructeurs de cellule :

— Un qui ne prend que la valeur (l'`Animal` ici) ;
— Un qui prend la valeur et la cellule suivante.

Cellule.java

```java
public class Cellule {
  ...

  public Cellule(Animal a, Cellule suivante) {
    this.valeur = a;
    this.suivante = suivante;
  }

  public Cellule(Animal a) {
    this(a, null);
  }
}
```

La classe `Liste` prend généralement un unique constructeur qui crée une liste vide :

Liste.java

```java
public class Liste {
  ...

  public Liste() {
    this.premier = null;
  }
}
```

1.8.1 Ajout d'une valeur au début de la liste

La première méthode décrite est l'ajout de valeur au début de la liste. Elle consiste à

1. Regarder quel est le premier élément `x` de la liste ;
2. Créer une nouvelle `Cellule c` qui contient la valeur et a pour suivante `x` ;
3. Changer première (de la classe `Liste`) en `c`.

Liste.java

```java
public class Liste {
  ...

  public void ajouteEnTete(Animal a) {
    // Etape 1
    Cellule actuellePremiere = this.premier;
    // Etape 2
    Cellule nouvellePremiere = new Cellule(a, actuellePremiere);
    // Etape 3
    this.premiere = nouvellePremiere;
  }
}
```

Exercice 3. *Regarder ce qu'il se passe lorsque la liste et vide et lorsque la liste contient déjà deux éléments.*

1.8.2 Test de vacuité

Pour tester si une liste est vide, il suffit de tester si le premier élément est `null` :

Liste.java

```
public class Liste {
  ...

  public void estVide() {
    return this.premiere == null;
  }
}
```

1.8.3 Méthodes itératives

Dans le cas des méthodes itératives, le principe est généralement de tester si le premier élément est différent de `null` dans la classe `Liste`. Si c'est le cas, on appelle une méthode de la classe `Cellule`. Dans cette méthode de la classe `Cellule`, on va avoir une variable `tmp` de type `Cellule` qui va représenter la `Cellule` que l'on est en train de traiter.

Description

Notre but ici est d'afficher tous les `Animal` de la liste. Dans le cas où la liste est vide, on affichera "Liste vide".

On suppose que dans la classe `Animal` il y a une méthode `description` qui retourne une chaîne de caractères le décrivant.

Liste.java

```
public class Liste {
  ...

  public void description() {
    // Etape 1 : tester si la liste est vide
    if (this.premiere == null)
      System.out.println("Liste vide");
    else
      // Si la liste n'est pas vide, on demande
      // à la première Cellule de faire la description.
      this.premiere.description();
  }
}
```

Cellule.java

```
public class Cellule {
  ...

  public void description() {
    // Etape 1 : Création d'une variable qui va nous
    // permettre de parcourir la liste.
    Cellule tmp = this;
    // Etape 2 : Avancer tant que cette variable n'est pas
    // null.
    while (tmp != null) {
      // Etape 3 : Travail sur la Cellule actuelle.
      System.out.println(tmp.valeur.description());
      // Etape 4 : On passe à la Cellule suivante.
      tmp = tmp.suivant;
    }
  }
}
```

Exercice 4. *Regarder ce qu'il se passe avec une liste*
- *vide*
- *avec un élément*
- *avec deux éléments*

Taille d'une liste

Pour connaître la taille d'une liste, on va :
- Regarder si elle est vide ;
- Si ce n'est pas le cas, on va demander la taille à la première `Cellule`.

Liste.java

```
1  public class Liste {
2    ...
3
4    public int taille() {
5      // Etape 1 : tester si la liste est vide
6      if (this.premiere == null)
7        return 0;
8      else
9        // Si la liste n'est pas vide, on demande
10       // à la première Cellule la taille.
11       return this.premiere.taille();
12   }
13 }
```

Cellule.java

```
1  public class Cellule {
2    ...
3
4    public int taille() {
5      // Etape 1 : Création d'une variable qui va nous
6      // permettre de parcourir la Liste.
7      Cellule tmp = this;
8      // Le nombre de Cellule.
9      int resultat = 0;
10     // Etape 2 : Avancer tant que cette variable n'est pas
11     // null.
12     while (tmp != null) {
13       // Etape 3 : Travail sur la Cellule actuelle.
14       resultat++;
15       // Etape 4 : On passe à la Cellule suivante.
16       tmp = tmp.suivant;
17     }
18   }
19 }
```

Ajout à la fin

Pour ajouter un élément à la fin d'une liste on va :
- Aller jusqu'à la dernière `Cellule` ;
- Ajouter un nouveau `suivante` à cette `Cellule`.

Liste.java

```
1  public class Liste {
2    ...
3
4    public void ajoutFin(Animal a) {
5      if (this.premiere == null) // Etape 1 : tester si la liste est vide
6        // Si c'est le cas, on crée une nouvelle Cellule
7        this.premiere = new Cellule(a);
```

```
    else
      // Si la liste n'est pas vide, on demande
      // à la première Cellule de faire l'ajout.
      this.premiere.ajouteFin(a);
  }
}
```

Cellule.java

```
public class Cellule {
  ...

  public void ajouteFin() {
    // Etape 1 : Création d'une variable qui va nous
    // permettre de parcourir la Liste.
    Cellule tmp = this;
    // Etape 2 : Avancer tant que cette variable a un
    // successeur.
    while (tmp.suivant != null) {
      tmp = tmp.suivant;
    }
    tmp.suivante = new Cellule(a);
  }
}
```

Exercice 5. *Regarder ce qu'il se passe lorsque l'on ajoute une valeur à une liste*

- *vide*
- *avec un élément*
- *avec deux éléments*

Ajout à une position donnée

Pour ajouter à une position `i` on va :

1. Aller jusqu'à la `Cellule i-1` ;
2. Ajouter une nouvelle `Cellule` entre les `Cellule i-1` et `i`.

Liste.java

```
public class Liste {
  ...

  public void ajoutPos(Animal a, int pos) {
    // Si la position est 0, on ajoute au début.
    if (pos == 0)
      this.premiere = new Cellule(a, this.premiere);
    else
      this.premiere.ajoutPos(a, pos);
  }
}
```

Cellule.java

```
public class Cellule {
  ...

  public void ajoutePos(Animal a, int pos) {// Remarque : Ici pos est toujours > 0
    // Etape 1 : Création d'une variable qui va nous permettre de parcourir la Liste.
    Cellule tmp = this;
    for (int i = 0; i < pos - 1; pos++){ // On avance jusqu'à la Cellule i-1.
      // Si on atteint la fin de la liste, on arrête la boucle
      if (tmp.suivante == null)
        break;
      tmp = tmp.suivante;
```

```
    }
    tmp.suivante = new Cellule(a, tmp.suivante);
  }
}
```

Exercice 6. *Regardez ce qu'il se passe pour ajouter une valeur à*
- *la première position (`pos = 0`) d'une liste quelconque ;*
- *la troisième position (`pos = 2`) d'une liste avec deux éléments ;*
- *la troisième position (`pos = 2`) d'une liste avec trois éléments ;*
- *la cinquième position (`pos = 4`) d'une liste avec trois éléments ;*

Suppression à une position donnée

Le principe pour supprimer une valeur à la position `i` est :

1. d'aller jusqu'à la position `i-1` ;
2. supprimer la `Cellule` suivante.

Liste.java

```
public class Liste {
  ...

  public void supprimePos(int pos) {
    // Si la position est 0, on supprime le premier.
    if (pos == 0)
      this.premiere = this.premiere.getSuivant();
    else
      this.premiere.supprimePos(pos);
  }
}
```

Cellule.java

```
public class Cellule {
  ...

  // Un getter pour la classe Cellule.
  public Cellule getSuivant() {
    return this.suivante;
  }

  public void SupprimePos(int pos) {
    // Remarque : Ici pos est toujours > 0

    // Etape 1 : Création d'une variable qui va nous
    // permettre de parcourir la Liste.
    Cellule tmp = this;
    // Etape 2 : On avance jusqu'à la Cellule pos-1.
    for (int i = 0; i < pos - 1; pos++){
      // Si on atteint la fin de la liste, on arrête
      if (tmp.suivante == null)
        return;
      tmp = tmp.suivante;
    }
    // Etape 3 : On supprime la Cellule.
    tmp.suivante = tmp.suivante.suivante;
  }
}
```

Égalité de deux listes

Pour tester l'égalité de deux listes, on va les parcourir en même temps (avec deux variables) et vérifier que les valeurs sont les mêmes.

Liste.java

```java
public class Liste {
  ...

  public boolean egal(Liste autre) {
    // Si les deux listes ne font pas la même taille,
    // elles ne peuvent pas être égales.
    if (this.taille() != autre.taille())
      return false;
    // Si les deux listes sont vides, c'est bon.
    if (this.estVide() && autre.estVide())
      return true;
    // Sinon, on doit les parcourir
    return this.premiere.egal(autre);
  }
}
```

Cellule.java

```java
public class Cellule {
  ...

  public boolean egal(Cellule autre) {
    Cellule thisTmp = this;
    Cellule autreTmp = autre;
    // Puisque les deux listes sont de même taille,
    // thisTmp != null est équivalent à
    // autreTmp != null
    while (thisTmp != null) {
      // Si les deux valeurs ne sont pas égales, alors
      // les listes ne sont pas égales.
      if (!thisTmp.valeur.egal(autreTmp.valeur))
        return false;
      thisTmp = thisTmp.suivante;
      autreTmp = autreTmp.suivante;
    }
    // On est arrivé à la fin des listes et on n'a
    // pas trouvé de différence.
    return true;
  }
}
```

1.8.4 La récursion

L'idée de la récursion est de procéder en deux étapes :

- Le traitement de la `Cellule` actuelle ;
- L'appel à la méthode actuelle sur la Cellule suivante.

Les méthodes `ajouteEnTete` et `estVide` restent les mêmes.

Description

Notre but ici est d'afficher tous les `Animal` de la liste.

Version "affichage" La première version va afficher les éléments pendant le parcours.

Liste.java

```java
public class Liste {
  ...

  public void description() {
    if (this.premiere == null)
```

```
      System.out.println("Liste vide");
    else
      this.premiere.description();
  }
}
```

Cellule.java

```
public class Cellule {
  ...

  public void description() {
    // Etape 1 : Traitement de la Cellule actuelle.
    System.out.println(this.valeur.description());
    // Etape 2 : Demander à la Cellule suivante de
    // travailler si elle existe.
    if (this.suivante != null)
      this.suivante.description();
  }
}
```

Version "String" Contrairement à la version "affichage" nous allons ici construire une chaîne de caractères qui sera renvoyée.

Liste.java

```
public class Liste {
  ...

  public String description() {
    if (this.premiere == null)
      return "Liste vide";
    else
      return this.premiere.description();
  }
}
```

Cellule.java

```
public class Cellule {
  ...

  public String description() {
    // Création de la variable qui contient le résultat
    String resultat = "";
    // Etape 1 : Traitement de la Cellule actuelle.
    // Le "\n" sert à ajouter un retour à la ligne.
    resultat += this.valeur.description() + "\n";
    // Etape 2 : Demander à la Cellule suivante de
    // travailler si elle existe.
    if (this.suivante != null)
      // On ajoute à résultat la description du reste
      // de la liste
      resultat += this.suivante.description();
    return resultat;
  }
}
```

Taille d'une liste

L'idée est que l'on va retourner :
- 1 si la `Cellule` n'a pas de successeur ;
- 1 + la taille du reste sinon.

Liste.java

```java
public class Liste {
  ...

  public int taille() {
    if (this.premiere == null)
      return 0;
    else
      return this.premiere.taille();
  }
}
```

Cellule.java

```java
public class Cellule {
  ...

  public int taille() {
    // Si on est sur le dernier élément de la liste, alors
    // la liste est de taille 1.
    if (this.suivante == null)
      return 1;
    else
      // On retourne 1 + this.suivante.taille();
      return 1 + this.suivante.taille();
  }
}
```

Exercice 7. *Essayez cette méthode sur une liste de taille*
- *0*
- *1*
- *3*

Ajout à la fin

L'idée est encore d'aller jusqu'à la dernière `Cellule` et de lui ajouter une nouvelle `Cellule` suivante.

Liste.java

```java
public class Liste {
  ...

  public void ajoutFin(Animal a) {
    if (this.premiere == null)
      this.premiere = new Cellule(a);
    else
      this.premiere.ajoutFin(a);
  }
}
```

Cellule.java

```java
public class Cellule {
  ...

  public void ajoutFin(Animal a) {
    if (this.suivante != null)
      this.suivante.ajoutFin(a);
```

```
7       else
8         this.suivante = new Cellule(a);
9     }
10  }
```

Ajout à une position donnée

Comme avant on va aller jusqu'à la `Cellule` `pos-1` et lui ajouter un nouveau successeur.

La récursion s'appuie sur l'idée suivante : Ajouter une valeur à la position `i` depuis la `Cellule` actuelle est la même chose qu'ajouter la valeur à la position `i-1` depuis la `Cellule` suivante.

Liste.java

```
1   public class Liste {
2     ...
3
4     public void ajoutPos(Animal a, int pos) {
5       if (pos == 0)
6         this.premiere = new Cellule(a, this.premiere);
7       else
8         this.premiere.ajoutPos(a);
9     }
10  }
```

Cellule.java

```
1   public class Cellule {
2     ...
3
4     public void ajoutPos(Animal a, int pos) {
5       // Ici pos est toujours >= 1
6       // Si je veux l'ajout à l'emplacement suivant
7       if (pos == 1)
8         this.suivante = new Cellule(a, this.suivante);
9       else if (this.suivante != null)
10        this.suivante.ajoutPos(a, pos - 1);
11      // Pour simplifier le code, si la position est trop
12      // grande je n'ajoute rien mais vous pouvez le gérer
13      // à la ligne 7
14    }
15  }
```

Exercice 8. *Essayez cette méthode sur :*
- *Une liste de taille 0 à la position 0 ;*
- *Une liste de taille 3 à la position 1.*
- *Une liste de taille 1 à la position 3.*

Égalité de deux listes

L'idée est de tester si les deux Cellule actuelles ont la même valeur. Si c'est le cas on fait l'appel récursif sur leur suivante.

Liste.java

```
1   public class Liste {
2     ...
3
4     public boolean egal(Liste autre) {
5       // Si les deux listes ne font pas la même taille,
6       // elles ne peuvent pas être égales.
7       if (this.taille() != autre.taille())
8         return false;
9       // Si les deux listes sont vides, c'est bon.
```

```
    if (this.estVide() && autre.estVide())
      return true;
    // Sinon, on doit les parcourir
    return this.premiere.egal(autre);
  }
}
```

Cellule.java

```
public class Cellule {
  ...

  public boolean egal(Cellule autre) {
    // On compare la Cellule actuelle de this
    // à l'actuelle de autre
    if (!this.valeur.egal(autre.valeur))
      return false;
    if (this.suivante != null) // et donc si autre.suivante != null
      return this.suivante.egal(autre.suivante);
    // S'il n'y a plus rien dans la liste, c'est fini
    return true;
  }
}
```

Suppression à une position donnée

Je propose ici une version sans *getter*. L'idée est que la méthode de la classe `Cellule` va toujours renvoyer la `Cellule` suivante de `this`. Ainsi dans la classe `Liste` on va récupérer le premier élément de la liste.

Liste.java

```
public class Liste {
  ...

  public void supprimePos(int pos) {
    if (pos >= 0 && pos < this.taille() && this.premier != null)
      this.premier = this.premier.supprimePos(pos);
  }
}
```

Cellule.java

```
public class Cellule {
  ...

  public void supprimePos(int pos) {
    // Cas où on est sur la Cellule à supprimer
    if (pos == 0) {
      return this.suivant;
    } else {
      // Puisque l'on a vérifié les bornes avant, this.suivant est != null
      this.suivant = this.suivant.supprimePos(pos - 1);
      return this;
    }
  }
}
```

Suppression des doublons dans une liste chaînée

Pour supprimer les `Cellule` en doublon, on va regarder si la `Liste` n'est pas vide. Si c'est le cas on va (dans la classe `Cellule`)

- Considérer la cellule actuelle (`supprimeDoublonsRec`) ;
- Supprimer toutes les `Cellule` après qui portent sa valeur (`supprimeVal`).

Liste.java

```java
public class Liste {
  ...

  public void supprimeDoublonsRec() {
    if (this.premier != null)
      this.premier.supprimeDoublonsRec();
  }
}
```

Cellule.java

```java
public class Cellule {
  ...

  public void supprimeVal(int val) {
    if (this.suivant != null) {
      if (this.suivant.val == val) {
        this.suivant = this.suivant;
        this.supprimeVal(val);
      }
      else
        this.suivant.supprimeVal(val);
    }
  }

  public void supprimeDoublonsRec() {
    this.supprimeVal(this.val);
    if (this.suivant != null)
      this.suivant.supprimeDoublonsRec();
  }
}
```

1.9 Listes doublement chaînées

Le deuxième type de liste vu en cours est la liste doublement chaînée (voir `https://en.wikipedia.org/wiki/Doubly_linked_list`).

Nous n'allons ici utiliser que des algorithmes récursifs.

Liste.java

```java
public class Liste {
  // Pointeur vers le premier élément de la liste.
  // Il vaut null si la liste est vide.
  private Cellule premier;
  // Pointeur vers le dernier élément de la liste.
  // Il vaut null si la liste est vide.
  private Cellule dernier;
}
```

Cellule.java

```java
public class Cellule {
  // Pointeur vers l'élément suivant de la liste.
  // Il vaut null si on est sur le dernier élément.
  private Cellule suivante;
  // Pointeur vers l'élément précédent de la liste.
  // Il vaut null si on est sur le premier élément.
  private Cellule precedent;
  // Valeur contenue dans la Cellule.
  private Animal valeur;
```

```
10
11    public Cellule(Animal a, Cellule precedent, Cellule suivant) {
12      this.precedent = precedent;
13      this.suivante = suivant;
14      this.valeur = a;
15    }
16  }
```

Elle ajoute à la classe `Liste` un pointeur vers la dernière `Cellule`. De plus une `Cellule` porte également un pointeur vers la `Cellule` précédente.

1.9.1 Ajout en tête de liste

Pour l'ajout en tête de liste, il faut penser à mettre à jour le dernier élément de la liste si elle était vide.

Liste.java

```
1  public class Liste {
2  ...
3    public void ajouteEnTete(Animal a) {
4      this.premier = new Cellule(a, null, this.premier);
5      if (this.dernier == null)
6        this.dernier = this.premier;
7    }
8  }
```

1.9.2 Ajout en fin de liste

La version du cours s'appuie sur un *setter* dans la classe `Liste` pour modifier le dernier élément de la liste.

Liste.java

```
1  public class Liste {
2  ...
3    public void ajouteFin(Animal a) {
4      // La liste est vide
5      if (this.premier == null)
6        this.premier = this.dernier = new Cellule(a, null, null);
7      else
8        Cellule c = new Cellule(a, this.dernier, null);
9        this.dernier.setSuivant(c);
10       this.dernier = c;
11   }
12 }
```

1.10 Listes doublement chaînées circulaires

Les listes doublement chaînées circulaires sont proches des listes doublement chaînées mais
- la liste n'a qu'un pointeur vers l'élément courant ;
- le précédent de la "dernière" cellule est la "première" (et inversement).

1.10.1 Ajout à la position courante

Ici on va faire ajouter une `Cellule` entre `courant` et son prédécesseur.

Liste.java

```
1  public class Liste {
2    ...
3
4    public void ajoutCourant(Animal a) {
5      if (this.courant = null)
```

```
6         this.courant = new Cellule(a);
7       else
8         this.courant = this.courant.ajoutCourant(a);
9     }
10  }
```

Cellule.java

```
1   public class Cellule {
2     ...
3
4     public Cellule ajoutCourant(Animal a) {
5       Cellule actuel_prec = this.prec;
6       Cellule c = new Cellule(a, actuel_prec, actuel);
7       actuel_prec.suivant = c;
8       actuel.precedent = c;
9       return c;
10    }
11  }
```

Chapitre 2

Compléments de TD

2.1 TD 5

Une correction du sujet du TD 5. Pour l'exercice 4 le code n'a pas été testé et est relativement complexe.

Cellule.java

```
1  public class Cellule {
2    private Employe emp;
3    private Cellule suivant;
4
5    public Cellule(Employe emp, Cellule suivant) {
6      this.emp = emp;
7      this.suivant = suivant;
8    }
9
10   public Cellule(Employe emp) {
11     this(emp, null);
12   }
13
14   public Employe getEmploye() {
15     return emp;
16   }
17
18   public Cellule getSuivant() {
19     return this.suivant;
20   }
21
22   public void affiche() {
23     // Affichage de l'employé dans la Cellule actuelle
24     System.out.println(this.emp.description());
25     if (suivant != null)
26       // Demande à la Cellule suivante de s'afficher
27       this.suivant.affiche();
28   }
29
30   public boolean appartient(String nom) {
31     if (this.emp.getNom().equals(nom))
32       return true;
33     else if (this.suivant != null)
34       return this.suivant.appartient(nom);
35     else
36       return false;
37   }
38
39   public void demission(String nom) {
```

```
40      // Dans cette méthode on va regarder si c'est le suivant qu'il faut
41      // supprimer. Si c'est le cas on va modifier this.suivant pour qu'il
42      // supprime l'employé suivant
43      if (this.suivant != null) {
44        if (this.suivant.emp.getNom().equals(nom))
45          this.suivant = this.suivant.suivant;
46        else
47          this.suivant.demission(nom);
48      }
49    }
50
51    public boolean augmente(String nom, int montant) {
52      // Si this est le bon employé alors on l'augmente et on indique que l'on
53      // a augmenté l'employé
54      if (this.emp.getNom().equals(nom)) {
55        this.emp.augmente(montant);
56        return true;
57        // Sinon s'il existe un suivant on retourne le résultat de l'appel récursif
58      } else if (this.suivant != null)
59        return this.suivant.augmente(nom, montant);
60      else
61        // S'il n'y a pas de suivant alors on n'a augmenté personne
62        return false;
63    }
64
65    public Cellule choixSalaire(int min, int max) {
66      // Cette implémentation inverse l'ordre de la liste. Ainsi pour une liste
67      // [1 2 3 4] avec min = 2, max = 3 le résultat contiendra [3 2]
68      // Une suite de nouvelle Cellule qui forment le résultat
69      Cellule resultat = null;
70      // La variable qui nous permet de parcourir la liste
71      Cellule tmp = this;
72      // On boucle tant que tmp n'est pas null
73      while (tmp != null) {
74        // Si l'Employe a un "bon" salaire...
75        if (tmp.emp.getSalaire() >= min
76            && tmp.emp.getSalaire() <= max)
77          // ... alors on ajoute cette Cellule au résultat
78          resultat = new Cellule(tmp.emp, resultat);
79        // On passe à la Cellule suivante
80        tmp = tmp.suivant;
81      }
82      return resultat;
83    }
84
85    public Cellule choixSalaire2(int min, int max) {
86      Cellule suivantResultat = null;
87      // Dans le cas où il existe un suivant on traite le reste de la liste
88      if (this.suivant != null)
89        suivantResultat = this.suivant.choixSalaire2(min, max);
90      // Si mon employé a un "bon" salaire alors je l'ajoute à la liste précédente
91      if (this.emp.getSalaire() >= min
92          && this.emp.getSalaire() <= max)
93        return new Cellule(this.emp, suivantResultat);
94      // Sinon je retourne juste le résultat de l'appel récursif
95      else
96        return suivantResultat;
97    }
98
99    public boolean croissante() {
100     // S'il n'y a rien après c'est bien croissant
101     if (this.suivant == null)
102       return true;
```

```
103       else
104         // La liste est croissante si je suis plus petit que le suivant dans la
105         // liste et que le reste de la liste est croissant.
106         return this.emp.getSalaire() <= this.suivant.emp.getSalaire()
107             && this.suivant.croissante();
108     }
109
110     public void ajoutTri(Employe e) {
111       // Dans cette méthode on est certain que this a un salaire < à celui
112       // de e.
113       // On va donc regarder si this.suivant a un salaire >= de e. Si c'est le
114       // cas alors on doit créer une Cellule entre this et this.suivant qui
115       // contient e.
116       // S'il n'y a pas de suivant, on l'ajoute juste après this
117       if (this.suivant == null)
118         this.suivant = new Cellule(e);
119       // Si le saiaire de this est inférieur à celui de l'employé alors on veut
120       // ajouter e plus loin dans la liste (=> appel récursif)
121       else if (this.suivant.emp.getSalaire() < e.getSalaire())
122         this.suivant.ajoutTri(e);
123       // On est dans le cas où this.salaire < e.salaire <= this.suivant.salaire
124       // donc on ajoute une Cellule entre this et this.suivant qui contient e.
125       else
126         this.suivant = new Cellule(e, this.suivant);
127     }
128
129     public void acquisition(Entreprise ent) {
130       // Ajout de l'Employe actuel à l'Entreprise
131       ent.ajoutTri(this.emp);
132       // Si l'Employe suivant existe on fait l'appel récursif
133       if (this.suivant != null)
134         this.suivant.acquisition(ent);
135     }
136
137     public void augmenteVersion2(String nom, int montant) {
138       // Etape 1 : Recherche de la Cellule pour laquelle le suivant porte le bon
139       // nom
140       Cellule tmp = this;
141       while (tmp.suivant != null && tmp.suivant.emp.getNom() != nom)
142         tmp = tmp.suivant;
143       // Cas où l'employé n'est pas dans la liste
144       if (tmp.suivant == null)
145         return;
146
147       // Correspond à pushIt mais je préfère le mettre ici
148       int nouveau_salaire = tmp.suivant.emp.getSalaire() + montant;
149       tmp.emp.setSalaire(nouveau_salaire);
150       while (tmp.suivant.suivant != null
151           && tmp.suivant.suivant.emp.getSalaire() < nouveau_salaire) {
152         Cellule x = tmp.suivant;
153         tmp.suivant = x.suivant;
154         x.suivant = x.suivant.suivant;
155         tmp.suivant.suivant = x;
156       }
157     }
158
159     public Cellule acquisitionVersion2(Cellule premier) {
160       // Le cas où on est arrivé au bout de la première entreprise : on copie
161       // les employés de this.
162       if (premier == null) {
163         Cellule res = null;
164         Cellule tmp = this;
165         while (tmp != null) {
```

```
166           res = new Cellule(emp, res);
167         }
168         return res;
169       }
170
171       if (this.emp.getSalaire() < premier.emp.getSalaire())
172       {
173         if (this.suivant != null)
174           return new Cellule(this.emp, this.suivant.acquisitionVersion2(premier));
175         else
176           return new Cellule(emp, premier);
177       }
178       else {
179         Cellule tmp = this.acquisitionVersion2(premier.suivant);
180         premier.suivant = tmp;
181         return tmp;
182       }
183
184     }
185   }
```

Employe.java

```
1   public class Employe {
2     private final String nom;
3     private int salaire;
4
5     public int getSalaire() {
6       return salaire;
7     }
8
9     public void setSalaire(int salaire) {
10      this.salaire = salaire;
11    }
12
13    public Employe(String nom, int salaire) {
14      this.nom = nom;
15      this.salaire = salaire;
16    }
17
18    public String description() {
19      return this.nom;
20    }
21
22    public String getNom() {
23      return this.nom;
24    }
25
26    public void augmente(int montant) {
27      this.salaire += montant;
28    }
29
30
31  }
```

Entreprise.java

```
1   public class Entreprise {
2     // Le sujet parle d'un nom mais puisqu'il n'est pas utilisé par la suite
3     // je le supprime.
4     private Cellule premier;
5
6     public Entreprise() {
```

```
7       this.premier = null;
8     }
9
10    private Entreprise(Cellule c) {
11      this.premier = c;
12    }
13
14    // Méthode récursive pour l'affichage des employés
15    public void affiche() {
16      if (this.premier != null)
17        this.premier.affiche();
18    }
19
20    public boolean appartient(String nom) {
21      if (this.premier == null)
22        return false;
23      return this.premier.appartient(nom);
24    }
25
26    public void ajout(Employe emp) {
27      if (!this.appartient(emp.getNom()))
28        this.premier = new Cellule(emp, this.premier);
29    }
30
31    public void demission(String nom) {
32      // Si on est dans une entreprise non-vide
33      if (this.premier != null) {
34        // Si c'est le premier employé qu'il faut supprimer
35        if (this.premier.getEmploye().getNom().equals(nom)) {
36          // Alors on dit que le nouveau premier est le 2e
37          this.premier = this.premier.getSuivant();
38        } else {
39          // Sinon on demande à la classe Cellule de le faire
40          this.premier.demission(nom);
41        }
42      }
43    }
44
45    public boolean augmente(String nom, int montant) {
46      // Verification des conditions
47      if (this.premier == null || montant <= 0)
48        return false;
49      // Demande à la classe Cellule de gérer ça.
50      return this.premier.augmente(nom, montant);
51    }
52
53    public Entreprise choixSalaire(int min, int max) {
54      // Dans cette méthode on va créer de nouvelles Cellules qui vont contenir
55      // des références vers l'Employe s'il a bien un salaire entre min et max.
56      if (this.premier != null)
57        return new Entreprise(this.premier.choixSalaire(min, max));
58      else
59        return new Entreprise(null);
60    }
61
62    public Entreprise choixSalaire2(int min, int max) {
63      if (this.premier != null)
64        return new Entreprise(this.premier.choixSalaire2(min, max));
65      else
66        return new Entreprise(null);
67    }
68
69    public boolean croissante() {
```

```
70      // Si la liste est vide elle est bien croissante.
71      if (this.premier == null)
72        return true;
73      // Sinon on demande à la classe Cellule
74      else
75        return this.premier.croissante();
76    }
77
78    public void ajoutTri(Employe e) {
79      // Si je dois ajouter au début de la liste (la liste est vide ou le salaire
80      // de e est inférieur à celui du premier)
81      if (this.premier == null
82          || this.premier.getEmploye().getSalaire() >= e.getSalaire())
83        this.ajout(e);
84      else
85        this.premier.ajoutTri(e);
86    }
87
88    public void augmenteTri(Employe e) {
89      this.demission(e.getNom());
90      this.ajoutTri(e);
91    }
92
93    public void acquisition(Entreprise ent) {
94      // L'idée est de créer dans la classe Cellule une méthode qui va parcourir
95      // les Cellule de ent et ajouter ses Employe à this (avec la méthode
96      // ajoutTri)
97      if (ent.premier != null) {
98        ent.premier.acquisition(this);
99      }
100   }
101
102   public void augmenteVersion2(String nom, int montant) {
103     // J'ai décidé de faire une version "void" pour simplifier le code mais
104     // il serait plus logique de renvoyer un Booléen comme dans la version
105     // précédente
106     if (this.premier != null) {
107       // Si c'est le premier on arnaque un peu : on le supprime de la liste
108       // et on fait ajout (il n'y a qu'un tour de boucle)
109         if (this.premier.getEmploye().getNom().equals(nom)) {
110           Employe e = this.premier.getEmploye();
111           this.premier = this.premier.getSuivant();
112           this.ajoutTri(e);
113         }
114         else
115           this.premier.augmenteVersion2(nom, montant);
116     }
117   }
118
119   public void acquisitionVersion2(Entreprise autre) {
120     // L'idée va être de parcourir les 2 entreprises en même temps.
121     // Voir https://fr.wikipedia.org/wiki/Tri_fusion#Optimisations_possibles
122     if (autre.premier != null) {
123       this.premier = autre.premier.acquisitionVersion2(this.premier);
124     }
125   }
126
127 }
```

Autres

Fonctions utiles

— `public static double sqrt(double a)`
— `public static double random()` renvoie une valeur $\in [0; 1[$
— `public static int round(float val)`

Exercice 9. *Que donne* `sqrt(-1.0)` *?*

Remarque 9. *Pour obtenir un entier aléatoire entre x et y, on fait* `(int) Math.random() * (y - x) + x;`

La vie plus facile sous VSCode

Puisque Atom est ~~atomisé~~ abandonné, je ne vais pas lister ses possibilités. Si vous souhaitez passer à VSCode en douceur, il existe l'extension ms-vscode.atom-keybindings qui permet de conserver vos raccourcis clavier.

Remarque 10. *Certains des raccourcis seront différents pour vous mais cette liste a principalement pour but de vous montrer ce qu'il est possible de faire avec un IDE bien configuré pour travailler plus rapidement.*

Remarque 11. *L'utilisation de certains raccourcis peut vous faire perdre l'habitude d'écrire les "vraies" morceaux de code (*`sysout` *par exemple).*

Remarque 12. *On suppose que le pack d'extensions vscjava.vscode-java-pack est installé et activé.*

— Mettre le document en forme ctrl+⇧+I
— Mettre la sélection en forme ctrl+K puis ctrl+F
— Dupliquer la sélection (pas de raccourcis mais c'est trop bien !)
— `sysout` + →| pour avoir `System.out.println()`
— `main` + →| pour avoir `public static void main(String[] args){}`
— Pour les autres, voir `https://github.com/redhat-developer/vscode-java/blob/master/snippets/java.json`, et tapez le contenu des lignes *prefix*
— Au dessus de votre fonction `main`, il y a un bouton *Run*
— Il y a également un bouton *Debug* qui permet d'exécuter votre programme mais en pouvant étudier le contenu des variables (il faut ajouter les endroits où suspendre le programme, voir `https://code.visualstudio.com/docs/editor/debugging#_breakpoints`). Cela évite d'afficher des valeurs partout !