

Tutorat

Le tutorat commencera la semaine du 13 février 2012 avec les créneaux et les salles suivants:

- Lun 12h30-14h30 (531C)
- Mar 12h30-14h30 (452C)
- Mer 12h30-14h30 (551C)
- Jeu 12h30-14h30 (551C)
- Ven 12h30-14h00 (436C)

- 1/ Rappels et **compléments Java (ICI)**
- 2/ Tableaux, boucles et invariants.
- 3/ Notions élémentaires de complexité.
- 4/ Récursion.
- 5/ Structures de données et introduction aux types abstraits de données.
- 6/ Quelques compléments Java.

Entièrement orienté objet

JAVA est un langage *entièrement orienté objet*.

Tout le code doit être inclus dans ce qu'on appelle des **classes**.

Entièrement orienté objet

JAVA est un langage *entièrement orienté objet*.

Tout le code doit être inclus dans ce qu'on appelle des **classes**.

Constructeurs

Lors de la création d'un objet (**instanciation d'une classe**), on fait appel à une **méthode générique** appelée **constructeur**. Cette méthode permet d'affecter des valeurs aux variables définies dans la classe et **obéit à des règles**.

- 1 Un constructeur doit toujours être une méthode **publique**.
- 2 Vous devez impérativement **donner le même nom** que votre classe au constructeur.
- 3 Un constructeur **ne retourne rien**.

Pour créer un objet, il faut tout d'abord déclarer une variable du type de la classe. Si vous avez une classe qui se nomme Test, votre variable sera de type Test.

```
public class Test
{
    public Test()
    {

    }
}
```

On peut maintenant créer un objet de type Test :

```
Test mon_objet = new Test();
```

Attribut

Un attribut permet de stocker des données à l'intérieur d'une classe.

Syntaxe:

```
[droits d'accès] [type de l'attribut]  
[nom de l'attribut] [éventuellement une définition]
```

Attribut

Un attribut permet de stocker des données à l'intérieur d'une classe.

Syntaxe:

```
[droits d'accès] [type de l'attribut]  
[nom de l'attribut] [éventuellement une définition]
```

Droits d'accès

Les droits d'accès à un attribut peuvent être de trois types :

- **private** : seules les méthodes de la classe dans laquelle est défini l'attribut peuvent y accéder et le modifier.
- **protected** : seules les méthodes de la classe et des éventuelles classes héritées peuvent accéder à l'attribut et le modifier.
- **public** : tout le monde peut accéder à l'attribut en question.

Généralités

Les méthodes (aussi appelées abusivement fonctions) vont vous permettre d'effectuer des traitements.

Un des principes de la POO

Responsabiliser les objets : ils doivent *s'assumer et communiquer* (entre eux et avec les autres objets).

Généralités

Les méthodes (aussi appelées abusivement fonctions) vont vous permettre d'effectuer des traitements.

Un des principes de la POO

Responsabiliser les objets : ils doivent *s'assumer et communiquer* (entre eux et avec les autres objets).

Une méthode se délimite comme une classe, c'est-à-dire par deux accolades. On est obligé d'y ajouter des parenthèses, (même si elles ne contiennent rien) sinon on y place les *paramètres*.

Généralités

Les méthodes (aussi appelées abusivement fonctions) vont vous permettre d'effectuer des traitements.

Un des principes de la POO

Responsabiliser les objets : ils doivent *s'assumer et communiquer* (entre eux et avec les autres objets).

Une méthode se délimite comme une classe, c'est-à-dire par deux accolades. On est obligé d'y ajouter des parenthèses, (même si elles ne contiennent rien) sinon on y place les *paramètres*.

Il existe différents *types de méthodes* que l'on peut globalement classer en trois familles :

- **Les accesseurs** : ces méthodes vont permettre de modifier un attribut privé d'une classe par une autre classe (il s'agit d'une méthode publique accessible aux autres classes).
- **Les modificateurs** : ces méthodes ne retournent rien et modifient la valeur d'un attribut privé. Il s'agit de méthodes publiques.
- **Le reste** : toutes les fonctions qui ne rentrent pas dans ces deux catégories.

Les droits d'accès des méthodes

Les méthodes disposent de d'accès qui limitent leur utilisation :

- **public** : la méthode est accessible de n'importe où.
- **private** : la méthode est accessible uniquement dans la classe où elle figure.
- **protected** : la méthode est accessible dans la classe où elle figure et dans les classes héritées.

Exemple

```
public class Test
{
    public Test()
    {
        MaMethode();
    }
    public void MaMethode()
    {
        System.out.println("Appel de la méthode Test");
    }
}
```

Cf. fichier complexe-cours2.java

...

- 1/ Rappels et compléments Java.
- 2/ Tableaux, boucles et invariants. **ICI**
- 3/ Notions élémentaires de complexité.
- 4/ Récursion.
- 5/ Structures de données et introduction aux types abstraits de données.
- 6/ Quelques compléments Java.

2/ Tableaux, boucles et invariants

- En java, les tableaux sont des objets particuliers
- Le type des éléments est appelé type de base du tableau
- Le nombre d'éléments gérés est un attribut fixé, appelé longueur du tableau.
- On accède aux différents éléments avec l'opérateur d'index [].

- Un tableau à une dimension : `int t1[]` ou `String[] t1`
- Un tableau à deux dimensions : `double t2[][]` ;
- et ainsi de suite...
- Les variables `t1` et `t2` représentent des variables références
- A ce stade, elles ne peuvent pas encore être utilisées.

- Une opération d'instanciation est nécessaire pour créer le tableau

```
t1 = new int[3] ;
```

```
t1[0] = 9 ;
```

```
t1[1] = 12 ;
```

```
/* t1[2] = 0 ; */
```

- Le premier élément d'un tableau à l'indice 0.

Lorsqu'on crée des tableaux avec des types non primitifs, il faut instancier les références.

```
String tabString = new String[4];  
tabString[0] = new String("Bonjour");  
tabString[1] = new String("tout");  
tabString[2] = new String("le");  
tabString[3] = new String("monde!");
```

On a vu les boucles (for, while, do ... while).

Une boucle est répétitive. Il faut toujours s'assurer que 2 problèmes soient résolus :

- 1 **La terminaison** : est-ce que la boucle se termine ?
- 2 **La preuve de programme** : la programme est-il correct ? est ce que le résultat obtenu à la fin de l'itération est celui attendu ?

Pour la preuve du programme, on utilise ce que l'on appelle un ***invariant de boucle*** qui est un prédicat qui ne change pas au cours de la boucle. C'est donc une proposition qui est vraie dans les 3 cas suivants :

- ➊ Avant l'exécution de la boucle (pré-condition)
- ➋ à chaque tour de boucle
- ➌ à la sortie (post-condition) de la boucle.

Une des méthodes pour les invariants consiste à déterminer ce qui reste inchangé à chaque itération dans la boucle.

Un exemple d'invariants

On a le code suivant:

```
static int somme2(int[] t, int r, int l) {  
    int s=0;  
    int i=r;  
    while(i<=l) {  
        s+=t[i]*t[i];  
        i+=1;  
    }  
    return s;  
}
```

Un exemple d'invariants

On a le code suivant:

```
static int somme2(int[] t, int r, int l) {  
    int s=0;  
    int i=r;  
    while(i<=l) {  
        s+=t[i]*t[i];  
        i+=1;  
    }  
    return s;  
}
```

Exo de compréhension

Donner un invariant de la boucle ci-dessous?

Un exemple d'invariants

On a le code suivant:

```
static int somme2(int[] t, int r, int l) {  
    int s=0;  
    int i=r;  
    while(i<=l) {  
        s+=t[i]*t[i];  
        i+=1;  
    }  
    return s;  
}
```

Exo de compréhension

Donner un invariant de la boucle ci-dessous?

Un invariant

Vérifier que

(Inv): ***s contient la somme des éléments du tableau de r jusqu'à i-1***

est un invariant montrant ainsi que l'algorithme calcule la somme des éléments de r à l du tableau t.

Il n'est pas toujours évident de prouver qu'une boucle se termine. L'exemple suivant est un prototype très connu dont on n'a toujours pas prouvé mathématiquement qu'il se termine.

```
public static boolean terminaison(int n) {  
    while(n!=1)  
        if ((n %2)==0) n=n/2;  
        else  
            n=3*n+1;  
    return true;  
}
```

Les algorithmes de **tris** de tableaux (par exemple trier les un tableau d'entiers par ordre [dé]croissant) représentent des algorithmes de base. Ils sont primordiaux en algorithmique (**théorie des algorithmes**). Nous allons voir plusieurs algo. de tris. Notamment, les plus “classiques”.

- Tri à bulles.
- Tri par insertion.
- Tri fusion.
- Tri ...

Le **tri à bulles** ou **tri par propagation** est un algorithme de tri qui consiste à faire remonter progressivement les plus grands éléments d'une liste (comme les bulles d'air remontent à la surface d'un liquide). Son code est donné ci-dessous:

```
public static void triBulle(int tableau[]) {  
    int longueur=tableau.length;  
    boolean permut;  
    do {  
        permut=false; //hypothèse : le tableau est trié  
        for(int i=0;i<longueur-1;i++) {  
            //Teste si 2 éléments successifs sont dans le bon ordre ou non  
            if(tableau[i]>tableau[i+1]) {  
                //s'ils ne le sont pas on échange leurs positions  
                int tmp=tableau[i]; tableau[i]=tableau[i+1];tableau[i+1]=tmp;  
                permut=true;  
            }  
        }  
    }  
    while(permut);  
}
```

1/ Faire **tourner** l'algorithme du tri à bulles sur un exemple

2/ Quels sont les **invariants** des boucles de l'algorithme?

Exemple d'exécution du tri à bulle

Prenons le tableau "5 1 4 2 8" et trions-la de manière croissante en utilisant le tri à bulles.

Première étape:

(**5** **1** 4 2 8) → (**1** **5** 4 2 8)

(1 **5** **4** 2 8) → (1 **4** **5** 2 8)

(1 4 **5** **2** 8) → (1 4 **2** **5** 8)

(1 4 2 **5** **8**) → (1 4 2 **5** **8**) Maintenant que ces deux éléments (5 et 8) sont triés (rangés), l'algorithme ne les intervertira plus.

Deuxième étape:

(**1** **4** 2 5 8) → (**1** **4** 2 5 8) On recommence les mêmes opérations.

(1 **4** **2** 5 8) → (1 **2** **4** 5 8)

(1 2 **4** **5** 8) → (1 2 **4** **5** 8)

(1 2 4 **5** **8**) → (1 2 4 **5** **8**)

La liste est triée, mais l'algorithme ne sait pas si le tri est fini. L'algorithme doit effectuer un passage sans aucune échange pour savoir si le travail est terminé. D'où

la troisième étape:

(**1** **2** 4 5 8) → (**1** **2** 4 5 8)

(1 **2** **4** 5 8) → (1 **2** **4** 5 8)

(1 2 **4** **5** 8) → (1 2 **4** **5** 8)

(1 2 4 **5** **8**) → (1 2 4 **5** **8**)

La règle est la suivante. **Si** l'on veut vérifier le code suivant

`pre programme post`

où `programme` est de la forme

`while (b) instructions`

alors on peut procéder de la manière suivante :

- trouver un invariant I pour la boucle `while (b) instructions`
- I doit en plus vérifier `pre`) I (la précondition doit être forte pour assurer I).
- I et **non b**) Post (à la sortie de la boucle, on doit assurer la postcondition
- vérifier que la boucle se termine.

Observation : si on veut trier un tableau t entre les indices 1 et r , on peut construire un algorithme basé sur la propriété suivante:

(P_1) : Le tableau est trié de 1 à $i - 1$.

En effet,

- Si $i = 1$ la propriété (P_1) est vraie.
- Si $i > r$ alors $[1, i - 1]$ contient $[1, r]$

Donc, si on arrive à construire une boucle telle que (P_1) soit un **invariant**, la règle implique qu'on aura un programme triant le tableau entre les indices 1 à r .

On va renforcer la propriété (P_1) avec la propriété suivante

(P_2) : pour tout a dans $[i, r - 1]$ et pour tout b dans $[1, i - 1]$ alors $t[b] \leq t[a]$.

On va renforcer la propriété (P_1) avec la propriété suivante

(P_2) : pour tout a dans $[i, r - 1]$ et pour tout b dans $[1, i - 1]$ alors $t[b] \leq t[a]$.

Pour avoir à la fois (P_1) et (P_2) vérifiées,

On va renforcer la propriété (P_1) avec la propriété suivante

(P_2) : pour tout a dans $[i, r - 1]$ et pour tout b dans $[1, i - 1]$ alors $t[b] \leq t[a]$.

Pour avoir à la fois (P_1) et (P_2) vérifiées, il suffit de déterminer l'élément **minimal** du tableau entre i et r et de l'échanger avec $t[i]$:

On va renforcer la propriété (P_1) avec la propriété suivante

(P_2) : pour tout a dans $[i, r - 1]$ et pour tout b dans $[1, i - 1]$ alors $t[b] \leq t[a]$.

Pour avoir à la fois (P_1) et (P_2) vérifiées, il suffit de déterminer l'élément **minimal** du tableau entre i et r et de l'échanger avec $t[i]$:

l'idée est d'avancer à chaque itération de i .

On a besoin de deux petits programmes :

On a besoin de deux petits programmes : $\text{Min}(t, i, j)$ qui donne l'indice du minimum du tableau t entre i et j ET de $\text{Echanger}(t, i, j)$ qui échange les éléments d'indice i et j du tableau.

Trivialement, soit $x = \text{Min}(t, i, j)$. Nous avons alors les deux propriétés :
(a) $x \in [i, j]$ et **(b)** $\forall k \in [i, j] \quad t[x] \leq t[k]$.

Et on a

On a besoin de deux petits programmes : `Min(t, i, j)` qui donne l'indice du minimum du tableau `t` entre `i` et `j` ET de `Echanger(t, i, j)` qui échange les éléments d'indice `i` et `j` du tableau.

Trivialement, soit $x = \text{Min}(t, i, j)$. Nous avons alors les deux propriétés :
(a) $x \in [i, j]$ et **(b)** $\exists k \in [i, j] \ t[x] = t[k]$.

Et on a

Un petit programme intuitif

```
int i = 1;
while (i < r) {
    int x = Min(t, i, r);
    Echanger(t, i, x);
    i = i + 1;
}
```

On peut vérifier que (

- Si le tableau t est bien défini pour les valeurs d'indices dans $[i,r]$ après l'instruction $i=i+1$, la propriété est vraie. Comme $[1,i-1]$ est vide alors $((P_1) \text{ et } (P_2))$ est vérifiée.
- Ensuite $((P_1) \text{ et } (P_2)) \text{ et } i > r$ entraîne $((P_1) \text{ et } i = r)$ et donc le tableau est bien **trié** pour les indices dans $[1,r]$.
- Et, la boucle termine toujours puisque on incrémente i .

- Si le tableau t est bien défini pour les valeurs d'indices dans $[i, r]$ après l'instruction $i=1$, la propriété est vraie. Comme $[1, i-1]$ est vide alors $((P_1) \text{ et } (P_2))$ est vérifiée.
- Ensuite $((P_1) \text{ et } (P_2)) \text{ et } i > r$ entraîne $((P_1) \text{ et } i = r)$ et donc le tableau est bien **trié** pour les indices dans $[1, r]$.
- Et, la boucle termine toujours puisque on incrémente i .

On écrit les codes de Min et de Echanger

Min

```
int x=i;
for(int k=i+1;k<=j;k++)
    if (t[k] < t[x]) x=k;
```

- Si le tableau t est bien défini pour les valeurs d'indices dans $[i, r]$ après l'instruction $i=1$, la propriété est vraie. Comme $[1, i-1]$ est vide alors $((P_1)$ et $(P_2))$ est vérifiée.
- Ensuite $((P_1)$ et $(P_2))$ et $i > r$ entraîne $((P_1)$ et $i = r$) et donc le tableau est bien **trié** pour les indices dans $[1, r]$.
- Et, la boucle termine toujours puisque on incrémente i .

On écrit les codes de Min et de Echanger

Min

```
int x=i;
for(int k=i+1;k<=j;k++)
    if (t[k] < t[x]) x=k;
```

Echanger

```
int tampon = t[i];
t[i] = t[x];
t[x] = tampon;
```

C'est le tri le plus intuitif. On remplace la boucle while par for et on a

triselection

```
public static void triselection(int[] tab,int l,int r) {  
    for(int i=l;i<=r;i++) {  
        int min=i;  
        for(int j=i+1;j<=r;j++)  
            if(tab[j]<tab[min]) min=j;  
        int tmp=tab[i];  
        tab[i]=tab[min];  
        tab[min]=tmp;  
    }  
}
```

C'est le tri le plus efficace sur les listes de petite taille (cf. paragraphe sur la notion de complexité). Il est le tri employé quand on range. Par exemple, quand on range les copies par ordre alphabétique, on prend une copie, on la met à sa place parmi les copies déjà dans l'ordre. Et on continue.

Le tri par insertion

C'est le tri le plus efficace sur les listes de petite taille (cf. paragraphe sur la notion de complexité). Il est le tri employé quand on range. Par exemple, quand on range les copies par ordre alphabétique, on prend une copie, on la met à sa place parmi les copies déjà dans l'ordre. Et on continue.

En fait, on maintient la propriété (P_1) en sachant que dans le tableau de 1 à $i-1$, les éléments ne sont pas forcément à leur place définitive. A chaque itération, on place $t[i]$ dans le tableau de 1 à i (ainsi le tableau reste trié à chaque incrément).

Tri par insertion

```
// on suppose ici que de 1 à l c'est déjà rangé
public static void triinsertion(int t[],int l,int r){
    int i;
    for(i=l;i<=r;i++){
        for(int j=i;j>l;j--){
            if(t[j]<t[j-1]) echanger(t,j-1,j);
        }
    }
}
```

C'est un tri élégant et compliqué. Il introduit aussi la récursivité (voir plus tard!). L'idée principale est la suivante : le résultat du tri est la fusion de deux (sous-)tableaux triés.

Les tris sont utilisés des milliers de fois tous les jours. Il est pertinent de savoir lequel de ces algorithmes est le plus adéquat et pour quelle(s) situation(s)!!!!

Les tris sont utilisés des milliers de fois tous les jours. Il est pertinent de savoir lequel de ces algorithmes est le plus adéquat et pour quelle(s) situation(s)!!!!

Pour étudier la complexité (cf. paragraphe complexité) des algorithmes, on distingue principalement 3 cas.

- Le cas **pire** (dans les situations les plus “catastrophiques” : difficiles parfois de distinguer ces situations!)
- En **moyenne** (nécessité d'une distribution en entrée : difficile la plupart du temps!)
- Dans le **meilleur** des cas.

On montre que dans tous les cas (pire/moyen/meilleur) :

On montre que dans tous les cas (pire/moyen/meilleur) :

Le tri par sélection exécute $\frac{n(n-1)}{2}$ itérations.

Complexité : tri par insertion

On peut améliorer le code précédent en arrêtant la boucle interne dès qu'on ne fait plus d'échange :

Tri par insertion

```
// on suppose ici que de 1 à l c'est déjà rangé
public static void triinsertion(int t[],int l,int r){
    int i;
    for(i=l;i<=r;i++){
        for(int j=i;j>l;j- -) {
            if(t[j]<t[j-1]) echanger(t,j-1,j);
            else break;
        }
    }
}
```

Complexité : tri par insertion

On peut améliorer le code précédent en arrêtant la boucle interne dès qu'on ne fait plus d'échange :

Tri par insertion

```
// on suppose ici que de 1 à l c'est déjà rangé
public static void triinsertion(int t[],int l,int r){
    int i;
    for(i=l;i<=r;i++){
        for(int j=i;j>l;j- -){
            if(t[j]<t[j-1]) echanger(t,j-1,j);
            else break;
        }
    }
}
```

On montre alors que dans le pire cas

le tri par insertion exécute $\frac{n(n-1)}{2}$ itérations et comparaisons. Dans le meilleur des cas (le tableau est trié), il effectue $n-1$ comparaisons. Dans le cas moyen, **intuitivement**, au moment d'insérer le i -ème élément, on est amené à faire la moitié de comparaisons pour faire $\frac{n(n-1)}{4}$ comparaisons.

- Le tri rapide (`quicksort`) dû à [\[HOARE 1961\]](#) .
- Le tri fusion.

- Le tri rapide (`quicksort`) dû à [\[HOARE 1961\]](#) .
- Le tri fusion.

Attention

Le but est ici de donner les idées générales des algorithmes de tri sur les tableaux. On verra un peu plus en profondeur la [récursion](#) plus tard. Idem pour l'analyse de [complexité](#) de ces deux derniers algorithmes.

Le tri rapide : les principes

Il est basé sur le paradigme **diviser pour régner**. L'idée consiste à placer un élément du tableau qu'on appellera **pivot** à sa place définitive de telle sorte que tous les éléments qui lui sont inférieurs (resp. supérieurs) sont mis à sa gauche (resp. à sa droite). Ce processus est répété **récurivement**. Techniquement, voici ce qu'on va faire

Idée de base

```
trirapide(tableau t, entier premier, entier dernier)
début
    si premier < dernier alors
        pivot = choixpivot(t,premier,dernier)
        pivot = partitionner(t,premier,dernier,pivot)
        trirapide(t,premier,pivot-1)
        trirapide(t,pivot+1,dernier)
    finsi
fin
```

```

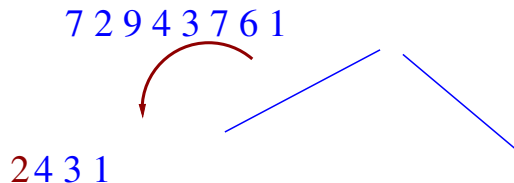
public static void quicksort(int [] tableau, int début, int fin) {
    if (début < fin) {
        int indicePivot = partition(tableau, début, fin);
        quicksort(tableau, début, indicePivot-1);
        quicksort(tableau, indicePivot+1, fin);
    }
}

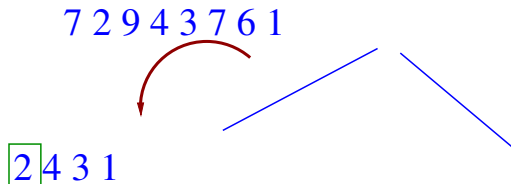
public static int partition (int [] t, int début, int fin) {
    int valeurPivot = t[début];
    int d = début+1;
    int f = fin;
    while (d < f) {
        while(d < f && t[f] >= valeurPivot) f--;
        while(d < f && t[d] <= valeurPivot) d++;
        int temp = t[d];
        t[d]= t[f];
        t[f] = temp;
    }
    if (t[d] > valeurPivot) d--;
    t[début] = t[d];
    t[d] = valeurPivot;
    return d;
}

```

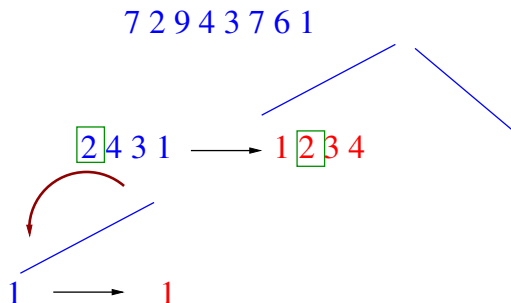
7 2 9 4 3 7 6 1

Exemple d'exécution

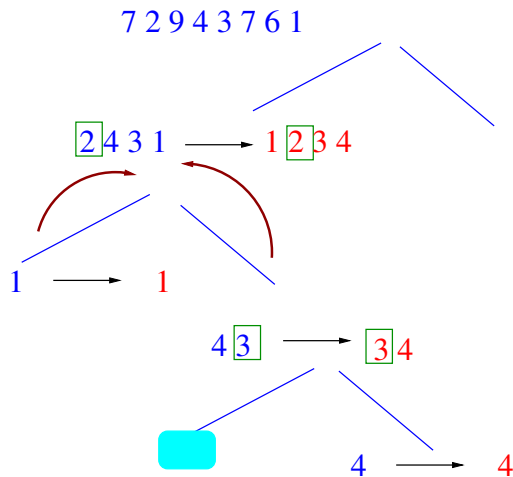




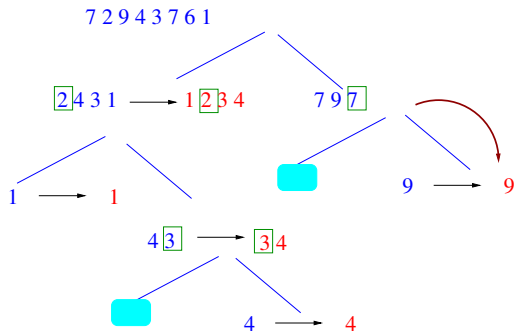
Exemple d'exécution



Exemple d'exécution



Exemple d'exécution



Exemple d'exécution

