Plan du cours

1/ Rappels et compléments Java.

2/ Tableaux, boucles et invariants.

3/ Notions élémentaires de complexité.

4/ Récursion. **ICI**

5/ Structures de données et introduction aux types abstraits de données. **ICI**

6/ Quelques compléments Java.

4/ Récursion.

1. Introduction & exemples.
2. Algorithmes récursifs.
3. Les différents types de récursivité.
4. Récursif ou itératif.
5. Complexités de nos exemples.

Tableaux

On a vu les **`tableaux`** . Ils ont comme avantages et inconvénients :

- L'insertion dans un tableau d'un élément peut entraîner un déplacement de tous les éléments du tableau (donc O(N) opérations pour un tableau de taille N).
- L'accès aux éléments se fait en temps constant O(1).
- Un tableau est une structure **statique** avec une taille fixe : (un objet tableau doit avoir une taille définie en java).

Listes chaînées

Définition récursive

Une liste chaînée est soit `vide` soit un élément suivi d'une liste chaînée.

Les `listes chaînées` permettent d'éviter les problèmes (insertion $O(N)$ et taille statique) rencontrés avec les tableaux et on aura avec ces structures :

des insertions et des suppressions en temps constant $O(1)$

Par contre, l'accès à un élément se fait en temps proportionnel à la taille de la liste (pas d'accès direct)!

Liste en JAVA

La définition peut s'écrire en JAVA et une liste est soit `null` soit un `Noeud`. Un `Noeud` est composé d'un `Item` suivi d'une référence vers un `Noeud` :

,

```
class Noeud {
    Object item;
    Noeud suivant;
    public Noeud(Object o) {
        item=o;
        suivant=null;
    }
    public Noeud(Object o, Noeud n) {
        item=o;
        suivant=n;
    }
}
```

Remarque

Une liste vide est représentée par `null` et pour accéder il faut tester au préalable que la liste est différente de null. Aussi parfois, il peut être intéressant de considérer qu'une liste contient au moins un noeud, dans ce cas la valeur de l'item de ce noeud n'est pas pris en compte et le premier élément la liste (si elle est non vide) est l'item du noeud suivant.

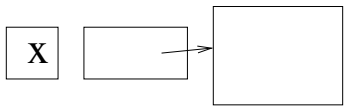
La classe Object

Une liste (comme un tableau) peut être définie pour n'importe quelle classe. Plutôt que de redéfinir des listes pour chaque classe, on peut vouloir définir des listes (de **manière générique**) pouvant contenir n'importe quel objet java.

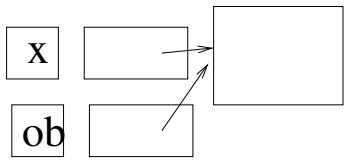
La classe Object

Une liste (comme un tableau) peut être définie pour n'importe quelle classe. Plutôt que de redéfinir des listes pour chaque classe, on peut vouloir définir des listes (de **manière générique**) pouvant contenir n'importe quel objet java.
Pour cela, on peut utiliser la classe `Object` (on a déjà rencontré cette classe à propos de la méthode `toString` qui permet de représenté comme chaîne de caractère n'importe quel objet) ; cette classe permet est dans une certaine mesure universelle elle peut correspondre à n'importe quel objet.

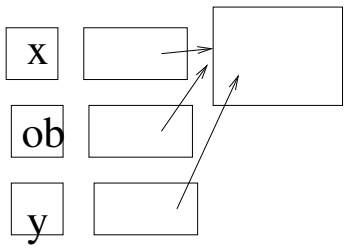
Type x = new Type();



Object ob = x;



Type y = (Type) ob;



Attention: forcer le type est obligatoire!

Forçage de type

Dans l'exemple précédent, le forçage de type : `(Type)` est nécessaire.
Java est un langage typé qui vérifie la compatibilité entre les types à la compilation.
Au moment de la compilation, le compilateur vérifie le typage, pour cela à partir des déclarations des variables et des types il est capable de déterminer le type de n'importe quelle expression et de vérifier que le typage est correct.
Dans l'exemple précédent, le compilateur sait que x est du type Type et que ob est du type Object. Cela signifie qu'il sait que x contient une référence sur un objet de type Type et x une référence de type Object. Comme Object est compatible avec n'importe quel type, x peut être une référence sur un objet de n'importe quel type.

Forçage de type

Dans l'exemple précédent, le forçage de type : `(Type)` est nécessaire.
Java est un langage typé qui vérifie la compatibilité entre les types à la compilation.
Au moment de la compilation, le compilateur vérifie le typage, pour cela à partir des déclarations des variables et des types il est capable de déterminer le type de n'importe quelle expression et de vérifier que le typage est correct.
Dans l'exemple précédent, le compilateur sait que x est du type Type et que ob est du type Object. Cela signifie qu'il sait que x contient une référence sur un objet de type Type et x une référence de type Object. Comme Object est compatible avec n'importe quel type, x peut être une référence sur un objet de n'importe quel type.
Il n'y a pas de problème ici parce que tout ce qui est possible avec un Object est possible avec n'importe quel objet (tous les objets ont, au moins, les propriétés des Object). Aussi ob=x ne pose pas de problème : x est une référence sur un objet Type et ob (déclaré comme variable de type Object) va référencer cet objet.

Forçage de type

Dans l'exemple précédent, le forçage de type : `(Type)` est nécessaire.
Java est un langage typé qui vérifie la compatibilité entre les types à la compilation.
Au moment de la compilation, le compilateur vérifie le typage, pour cela à partir des déclarations des variables et des types il est capable de déterminer le type de n'importe quelle expression et de vérifier que le typage est correct.
Dans l'exemple précédent, le compilateur sait que x est du type Type et que ob est du type Object. Cela signifie qu'il sait que x contient une référence sur un objet de type Type et x une référence de type Object. Comme Object est compatible avec n'importe quel type, x peut être une référence sur un objet de n'importe quel type.
Il n'y a pas de problème ici parce que tout ce qui est possible avec un Object est possible avec n'importe quel objet (tous les objets ont, au moins, les propriétés des Object). Aussi ob=x ne pose pas de problème : x est une référence sur un objet Type et ob (déclaré comme variable de type Object) va référencer cet objet.

Attention

Par contre, y=ob pose un problème : pour le compilateur, ob est une référence sur un Object (le compilateur ne regardant que les déclarations ne sait pas qu'ici, en fait, ob est une référence sur un Type) alors que y doit être une référence sur un Type. Le compilateur ne peut pas accepter (comme telle) cette affectation car tout Object ne peut pas être considéré comme un Type.