Java et OOP
Classes abstraites en interfaces

Processus d'abstraction

- On décide que les hommes et les baleines ont des choses en commun, qu'on met dans une classe `Mammifere` et qu'on considère comme ancêtre commun.
- Ce processus est un processus d'*abstraction* : on a abstrait des caractéristiques communes des deux classes pour les mettre dans une classe plus générale.
- La même chose se passe avec les classes `Mammifere` et `Poisson` – classe plus abstraite `Animal`.
- Plus on déduit des caractéristiques communes, plus on remonte dans la hiérarchie des classes, plus elles deviennent abstraites...
- On peut arriver à un certain moment ou la classe est tellement abstraite qu'on n'aurait jamais besoin de définir des objets de cette classe.
- Exemple : classe `Object` – c'est trop générale pour pouvoir se servir d'objets de type `Object` dans la programmation de tous les jours !
- Par contre, la **classe** `Object` est bien utile par les caractéristiques qu'elle définit et qu'on n'a pas besoin de définir nous-même dans chaque autre classe !

Méthodes et classes abstraites

- Une méthode abstraite spécifie le prototype d'une méthode que tous les héritiers doivent implémenter.
- C'est le contraire d'une méthode finale !

```
abstract class Animal{
    abstract void mange(Object quoi);
}
```

- Il n'y a pas de code pour une méthode abstraite ! On ne peut pas définir de façon générale comment un animal mange !
- Une classe qui possède une méthode abstraite doit elle aussi être déclarée abstraite.
- Il ne peut pas exister d'objet de classe abstraite. Donc on ne peut pas déclarer d'objet de classe `Animal`.
- On peut déclarer une classe `abstract` même si elle n'a pas de méthode abstraite :

```
abstract class Personne {
 String nom, prenom;
 void affiche(){ System.out.println("nom = " + prenom + " " + nom);}
}
```

But des classes abstraites

- Structurer l'ensemble des classes.
- Faciliter la concéption des classes.
- Améliorer la clarté, la lisibilité des classes.
- Créer des modèles plus rigoureux pour développer des applications spécifiques.
- Factoriser, modulariser le code.
- Permettre le polymorphisme.

Example

Essayons de voir avec l'hiérarchie des personnes/étudiants/professeurs :

- On n'utilisera plus des `Personnes` – on suppose qu'on range que des `Etudiants` ou des `Professeurs` dans nos listes hétérogènes.
- Donc, la classe `Personne` devient abstraite.
- Il s'agit juste de déclarer la classe comme abstraite, les méthodes gardent leurs implémentations.
- **Effets** :
  - On ne peut plus **créer** des objets de type `Personne` !
    Donc, pas de `new Personne("Qqn","Bizarre")`.
  - On ne peut plus cloner des Personnes !
  - Mais on peut déclarer des références de type `Personne` :
    `Personne[] p = new Personne[3] ;` est permis !

Interfaces – introduction

- L'héritage multiple est intérdit en Java.
- Mais, parfois, on a besoin d'hériter plus d'un type dans notre nouvelle classe :
  - Dans notre exemple d'hiérarchie de `Personne`s, on veut rajouter des fonctionnalités d'affichage formaté.
  - Ces fonctionnalités seraient aussi utiles pour l'hiérarchie `Animal/Mammifere/Poisson` etc.
  - Il s'agit bien des fonctionnalités à déclarer comme **abstraites** : on ne peut pas supposer qu'on affiche de la même manière les animaux et les personnes !
- Ça serait contraire au principe de la programmation orientée objet de devoir déclarer ces fonctionnalités dans chaque classe qui les implémente !
- **Solution** : interfaces !

Interfaces

- En quelque sorte, ces sont des *classes abstraites pures*.
- Ne déclarent que :
  - des méthodes **abstraites publiques**.
  - des **attributs publics, statiques et finaux**.
- Ces sont vraiment des *interfaces*, car elles permettent de définir des "protôcoles de communication" entre classes :

"Voilà à quoi une classe doit s'attendre si elle veut obtenir/échanger de l'information avec les classes qui m'implémentent"

```
interface Affichable{
 public abstract void affiche();
}
```

Implémenter une interface

- Une classe peut "hériter" une interface – mot-clé `implements` :
  `class Animal implements Affichable {...}`
- Rappel : les méthodes définies dans une interface sont abstraites !
- Donc dans la classe `Animal`, on doit écrire du code pour la méthode `affiche()` !
- Par ailleurs, une classe abstraite peut elle aussi "implémenter" une interface :
  `abstract class Personne implements Affichable {...}` .
- Dans ce cas il n'est pas nécessaire d'écrire du code pour `affiche()` dans `Personne` – cette classe est elle-aussi **abstraite** !
- On peut implémenter plusieurs intérfaces !
- En même temps, on peut hériter une classe !

```
class Etudiant extends Personne
      implements Affiche_ds_fenetre, Affiche_en_mode_text{
   .....
}
```

- Donc `Etudiant` implémente trois interfaces : `Affichable`, `Affiche_ds_fenetre`, `Affiche_en_mode_text`.

Interfaces qui se héritent

- Les interfaces peuvent hériter des autres interfaces ! Il suffit de penser qu'une interface est une classe de type spécial :

```
interface Ne_Sert_A_Rien extends Affichable{
   public static final String NOM = new String("un mot");
}
```

Les attributs étant statiques, ils doivent être initialisés.

- Même plus, une interface peut hériter de plusieurs autres interfaces !

```
interface X {}
interface Y {}
interface Z extends X,Y{ }
```

- Il n'y a pas de membres privés ou protégés !

Héritage et ambiguité

♦ Faire attention aux ambiguités !

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }
class C2 implements I1, I2 {
  public void f() {}                  // f de I1 redéfinie
  public int f(int i) { return 1; } // f de I2 redéfinie
}
class C3 extends C implements I2 {  // f de C non-redéfinie
  public int f(int i) { return 1; } // f de I2 redéfinie
}
class C4 extends C implements I3 {  // f héritée de C et de I3 avec
  public int f() { return 1; }       // la même signature
}
class C5 extends C implements I1 {} // ERREUR ! deux f avec la même signature
interface I4 extends I1, I3 {}       // ERREUR ! pareil
```

Interfaces et transtypage

- Rappel du principe : une interface est un type spécial de classe.
- Donc des références de type interface peuvent être déclarées !
- Alors on peut faire du transtypage implicite (upcasting) et explicite (downcasting) :

```
Affichable x = new Etudiant("Jean","X","deug");
```

Exemple d'utilisation

- On veut créer un algorithme de tri qui puisse être utilisé par tout type d'objet.
- Bien sûr, on ne peut pas trier des objets qui ne sont pas comparables !
- C'est exactement là qu'on utilise une interface :

```
interface Comparable{
    public abstract int compareTo(Object o);
}
```

- Cette interface existe dans l'API Java ! Vous n'avez pas besoin de la redéclarer !
- Il faut toujours penser qu'on l'appele avec un objet – dans toute implémentation il y aura un `this` !
- Une implémentation attendue devrait retourner :
  - 0 si `this` et `o` ont la même valeur.
  - -1 si `this` est "plus petit" que `o`.
  - +1 si `this` est "plus grand" que `o`.

Exemple continué

- L'algorithme est un des algorithmes connu (tri par bulles, quicksort...) :

```
public class Tri{
  public static void triBulles(Comparable[] t){
    for(int l=t.length-1; l>0; l--){
        for(int i=0; i<l; i++)
            if (t[i].compareTo(t[i+1]) > 0){
                Comparable tmp = t[i];
                t[i] = t[i+1];
                t[i+1] = tmp;
            }
    }
  }
}
```

Implémentation de l'interface

- Maintenant on veut trier des dates :

```
class Date implements Comparable{
   private int jour, mois, annee;
   public int compareTo(Object o){
       Date d = (Date)o;
       if (annee < o.annee) return -1;
       if (annee > o.annee) return 1;
       if (mois < o.mois) return -1;
       if (mois > o.mois) return 1;
       if (jour < o.jour) return -1;
       if (jour > o.jour) return 1;
       return 0;
   }
}
```

Trier des dates

```
class Main{
   public static void main(String[] argv){
       Date[] mes_dates = new Date[1000];
       // initialisation
       Tri.triBulles(mes_dates);
   }
}
```

Interfaces en tant que paramètres

- On peut utiliser une interface comme paramètre de type fonction du C :

```
interface FctUneVar{
  double valeur(double);
}
class Exp implements FctUneVar{
   double valeur(double x){ return Math.exp(x);}
}
class Integration{
   public static double integrer(FctUneVar f, double debut,
                                 double fin, double pas){
       double s = 0., x = deb;
       long n = (long)((fin - deb)/pas + 1;
       for (long k=0; k<n; k++, x+=pas) s+= f.valeur(x);
       return s/n;
   }
}
double intExp = Integration.integrer(new Exp(),0.,1.,0.001)
```

Interface ou classe abstraite ?

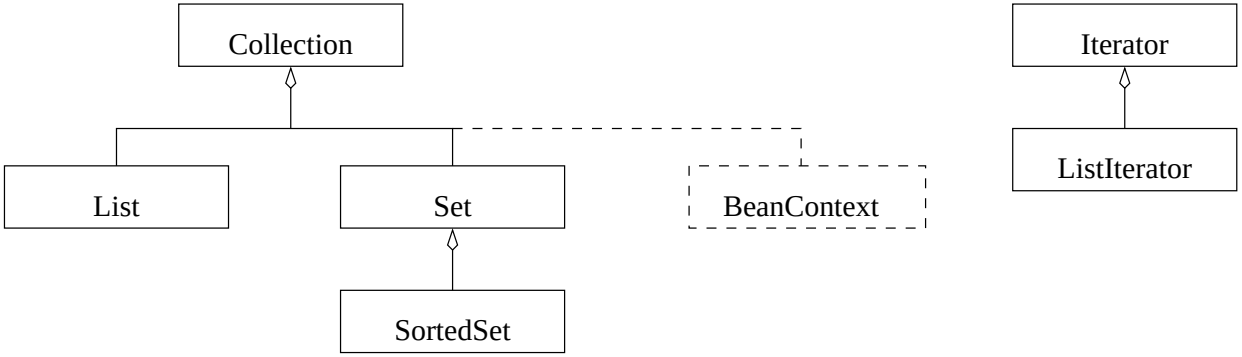
- Une interface représente un ensemble de fonctionnalités qu'une classe implémentant l'interface doit fournir.
- Une classe abstraite représente le "squelette" d'un objet qu'il est possible de créer.
- Une classe abstraite permet de spécifier certaines fonctionnalités.
- Une interface ne peut pas contenir d'attribut non-statique.
- Il est préferable d'utiliser les interfaces que les classes abstraites.
- Mais parfois les classes abstraites sont utiles !

On préfère une classe abstraite à une interface si on veut encapsuler des constantes :

```
abstract class ConstantesPhysiques{
    public static final double AVOGADRO = 6.02214e23;
    public static final double MASSE_ELECTRON = 9.109e-31;
}
```

Quelques interfaces utiles

Paquetage `java.util`



- `Collection` – permet de manipuler toutes les collection de façon unifiée :
  - Taille : `int size()`
  - Test si vide : `boolean isEmpty()`
  - Ajout : `boolean add(Object elmt)`
  - Suppression : `boolean remove(Object elmt)`
  - Recherche : `boolean contains(Object elmt)`
  - parcours : `Iterator iterator()` – on va voir à la suite...
  - d'autres – jeter un GROS coup d'oeil dans l'API Java !
- L'interface `Collection` ne précise pas
  - Si les élements apparaissent plusieurs fois ou non,
  - Si les élements sont stockés suivant un ordre

Quelques interfaces utiles

- `Set extends Collection` – les mêmes méthodes que `Collection`, mais le but est différent : sert implémenter des ensembles, donc chaque élement doit apparaître une seule fois.
- `List extends Collection` – en plus de méthodes de `Collection`, des fonctionnalités pour l'accès par position des élements (et un Iterateur spécifique) :
  - `Object get(int index)`
  - `Object set(int index, Object elmt)`
  - `void add(int index, Object elmt)`
  - `void indexOf(Object elmt)`
  - `void lastIndexOf(Object elmt)`
  - `ListIterator listIterator()`

Interfaces `Iterator`

- Permet d'implémenter une méthode de parcours de n'importe quelle collection.
- Méthodes déclarées :
  - `boolean hasNext()` – renvoie `true` si l'objet courant n'est pas le dernier
  - `Object next()` – élement suivant l'élement courant.
  - `void remove()` – enlève le dernier objet renvoyé par l'itérateur.

```
class A{
   int identite;
   A(int i){ identite = i;}
   void id(){ System.out.println("identite : "+identite);}
}
class B{
 public static void main(String[] args) {
   List listeA = new ArrayList();     // classe implementant l'interface List
   for(int i = 0; i < 7; i++) listeA.add(new A(i));
   Iterator e = listeA.iterator();
   while(e.hasNext()) ((A)e.next()).id();
 }
}
```