IUT d’Orsay - DUT Informatique - Jean-Claude MARTIN – JEAN-CLAUDE.MARTIN@U-PSUD.FR

# Programmation Orientée Objet (POO)

# COURS 6 : Interfaces

Définition

* Une interface est une spécification complètement abstraite :   
  ……………………………………………………………………………………………………………………………………………………..

……………………………………………………………………………………………………………………………………………………..

* Une interface ne comporte que des constantes et des méthodes abstraites
* ATTENTION : le terme "interface" n'a rien à voir ici avec les "interfaces graphiques" (que nous verrons plus tard dans le cours)
* Interface = spécification = partie visible par les développeurs qui vont utiliser la notion définie par l’interface
* Par opposition à Implémentation = choix de programmation (attributs, corps et instructions des méthodes)

Définition en Java : le mot clé *interface*

// DEFINI PAR LE RESPONSABLE DU STAGIAIRE

**interface Stockable {**

**// Definition de constantes**

**// Definition de prototypes des methodes**

**// (donc non implementees)**

**}**

Utilisation en Java

* Une classe qui implémente une interface doit définir le corps (les instructions) de toutes les méthodes dont le prototype figure dans l’interface
* Le mot-clef *implements* permet de rendre une classe conforme à une interface particulière (ou à plusieurs interfaces).
* L'interface spécifie ce à quoi la classe ressemble / son rôle.   
  L’implémentation spécifie comment la classe fonctionne, comment elle remplit son rôle.

// DEFINI PAR LE RESPONSABLE DU STAGIAIRE

**class** Entrepot {

// METHODE QUI PREND EN PARAMETRE UNE INSTANCE D'UNE CLASSE

// QUI IMPLEMENTE L'INTERFACE STOCKABLE

**void** ajouterProduit (Stockable s) {

**double** valeur = 0 ;

// ...

valeur = valeur + **s.getValeur()**;   
=> getValeur() est une méthode

**……………………………………………………………………………..**

**CAR ELLE A :**

// ...

}

}

// PROGRAMME PAR LE STAGIAIRE :

// DOIT IMPLEMENTER ………………………………………………………………………………………………….

**public** **class** Bouteille **……………………………………………………………………...** {

**double** valeurUnitaire ;

**float** volume ;

**public** **double** getValeur () {

**return** valeurUnitaire \* volume ;

}

**public** **void** commander (**int** quantite) {

System.*out*.println ("Commande de "+ quantite + " bouteilles");

}

**public** **static** **void** main (String args[]) {

Entrepot e = **new** Entrepot();

Bouteille b = **new** Bouteille ();

e.ajouterProduit(b);

}

}

**public class Table implements Stockable {**

**private int nbPieds ;**

**…**

**public double getValeur () {**

1er intérêt : séparer spécification et implémentation

* Le but est le même que pour les classes abstraites mais à l'extrême : aucune implémentation n'est fournie
* Une interface ne peut pas définir
  + D’implémentation / instructions de méthodes
  + Des attributs
* Une interface peut définir
  + Des constantes
  + Des prototypes de méthodes / des méthodes abstraites
* Interface = contrat entre le codeur de la classe et les développeurs
* Interface = ce que fait la classe (par opposition à l’implémentation = comment elle le fait)

2ème intérêt : permettre le polymorphisme avec des classes héritant d’autres classes

Exemple : un simulateur de zoo

* Hiérarchie de classes Animal, Félin, Canin, Lion, Chat, Tigre, Hippopotame, Loup, Chien
* Ajouter des méthodes pour les animaux Apprivoisables afin de pouvoir appeler par exemple la méthode visiteVeterinaire (Apprivoisable a)   
  pour les instances des classes.……………………………………………………………
* On doit pouvoir appeler les méthodes suivantes sur les animaux apprivoisables (chaque classe doit pouvoir implémenter à sa façon ces méthodes):
  + void **adopter** (Personne proprietaire)
  + Date getProchaineDateVisiteVeterinaire ()
  + void vacciner ()

Solutions possibles :

* **Définir la méthode adopter() dans la classe Animal**...
  + Inconvénient(s) :

UN TIGRE N’EST PAS APPRIVOISABLE

BESOIN DE MODIFIER LES SOUS-CLASSES COMME CHIEN CHAT … CAR ELLES DEVRONT IMPLEMENTER DE MANIERE SPECIFIQUE CES METHODES

* **Définir la méthode adopter () comme méthode abstraite dans la classe Animal**...)
  + Inconvénient(s) :

ETRE OBLIGE D’AVOIR DES METHODES QUI NE FONT RIEN DANS LES CLASSES D’ANIMAUX NON APPRIVOISABLE   
(ET SURTOUT ANNONCER VIA LES ENTETES DE METHODES QU’ELLES LE FONT)

* **Définir la méthode adopter uniquement dans Chien et Chat**...
  + Inconvénient(s) :

LES METHODES apprivoisables NE FONT PAS L’OBJET D’UN CONTRAT : UN DEVELOPPEUR PEUT SE TROMPER ET CHANGER LE TYPE D’UN PARAMETRE PAR EXEMPLE : LE COMPILATEUR NE PEUT PAS VERIFIER QUE CELA CORRESPOND A UN CONTRAT CAR IL N’Y EN N’A PAS

On ne peut pas utiliser le polymorphisme : la méthode visiteVeterinaire doit être implémentée pour chaque classe de paramètre possible : chat, chien, … (car le compilateur ne nous laissera pas appeler une méthode visiteVeterinaire sur un Animal en général

* CE DONT ON A BESOIN :
  + Une manière d’avoir les comportements apprivoisable uniquement dans les classes d’animaux apprivoisables
  + Une manière de garantir que toutes les classes d’animaux apprivoisables définissent les méthodes comme il le faut (entête, nombre et type des paramètres, …)
  + Une manière de pouvoir bénéficier du polymorphisme : permettre d’appeler les méthodes spécifiques des classes apprivoisable sans à avoir à créer une version des méthodes pour chaque classe apprivoisable

Une solution : l’héritage multiple ?

* Schéma avec Apprivoisable, Animal, …

HIERARCHIE DES CLASSES + INTERFACE

* Problème de l’héritage multiple : il peut y avoir ambiguïté lors de l’appel à une méthode qui figure dans Apprivoisable et dans Animal
* Est-ce que l’héritage multiple est possible en Java : NON
* La solution en Java :
* Définition de Apprivoisable en tant que INTERFACE
  + Implémentation des 3 méthodes d’Apprivoisable uniquement dans les classes qui en ont besoin : CHAT et CHIEN
  + Le polymorphisme est possible grace à l’interface :
    - Chat chat = new Chat ();
    - visiteVeterinaire (chat);
    - Chien chien = new Chien ();
    - visiteVeterinaire (chien);
    - car les classes Chat et Chien IMPLEMENTENT L’INTERFACE APPRIVOISABLE

Exemple d’implémentation Java

**import** java.util.Date;

//----------------------------------------------

// 1) HIERARCHIE DE CLASSES

//----------------------------------------------

**class** Animal { }

**class** Felin **extends** Animal { }

**class** Tigre **extends** Felin { }

//----------------------------------------------

// 2) DEFINITION DE L'INTERFACE

//----------------------------------------------

**interface** Apprivoisable {

Date getDateVisiteVeterinaire ();

**void** adopter ();

}

//----------------------------------------------

// 3) DEFINITION D'UNE CLASSE (ou plusieurs classes) QUI \*IMPLEMENTE\* L'INTERFACE

//----------------------------------------------

**public** **class** Chat **extends** Felin **implements** Apprivoisable {

**public** Date getDateVisiteVeterinaire() {

**return** **new** ~~Date~~(2103, 12, 20); // on retourne une date qui est spécifique aux chats

}

**public** **void** adopter () {

System.*out*.println ("Adoption d'un chat");

}

**public** **static** **void** main (String args []) {

Chat a = **new** Chat ();

}

}

// POLYMORPHISME !!!

// Zoo qui comporte une animalerie avec CERTAINS animaux adoptables par des familles

//----------------------------------------------

//4) ON PEUT UTILISER DES METHODES QUI PRENNENT EN PARAMETRE L'INTERFACE Apprivoisable

// (plus exactement une instance d'une classe qui implémente l'interface Apprivoisable)

//----------------------------------------------

**class** Zoo {

// A l'exécution, la référence animal pourra contenir l'adresse

// d'un objet instance d'une classe qui implémente l'interface

// Apprivoisable

Apprivoisable animal ;

Apprivoisable animaux []; // Idem avec un tableau de références

Zoo () {

animal = **new** Chat ();

// J'ai le droit car Chat est une classe

// qui implémente l'interface Apprivoisable

animal.adopter ();

}

**void** visiteVeterinaire (Apprivoisable a) {

//...

System.*out*.println ("Prochaine visite " + a.getDateVisiteVeterinaire());

//...

}

}

Héritage “multiple” et interfaces

* Une classe peut-elle hériter de plusieurs classes en Java ?**..………………….**
* Une classe C2 peut-elle étendre une classe C1 et implémenter une interface ? OUI**………**
* Une classe peut-elle implémenter plusieurs interfaces ? **……………………**
* Une interface peut-elle étendre plusieurs interfaces ?**..……………………**

# Video

<https://www.youtube.com/watch?v=fX1xNMBTMfg>



Exemple : utiliser une interface du JDK

Il existe des classes en Java implémentant des algorithmes de tri : par ex la classe Arrays

**import** java.util.Arrays;

**public** **class** TestArrays {

**public** **static** **void** afficher (**int** t []) {

System.*out*.println ();

**for** (**int** i=0; i < t.length; i++)

System.*out*.print (t[i] + " ");

}

**public** **static** **void** main (String args []) {

**int** t [] = {3, 6, 1, -10, 5};

*afficher* (t);

Arrays.*sort* (t);

*afficher* (t);

Résultat :

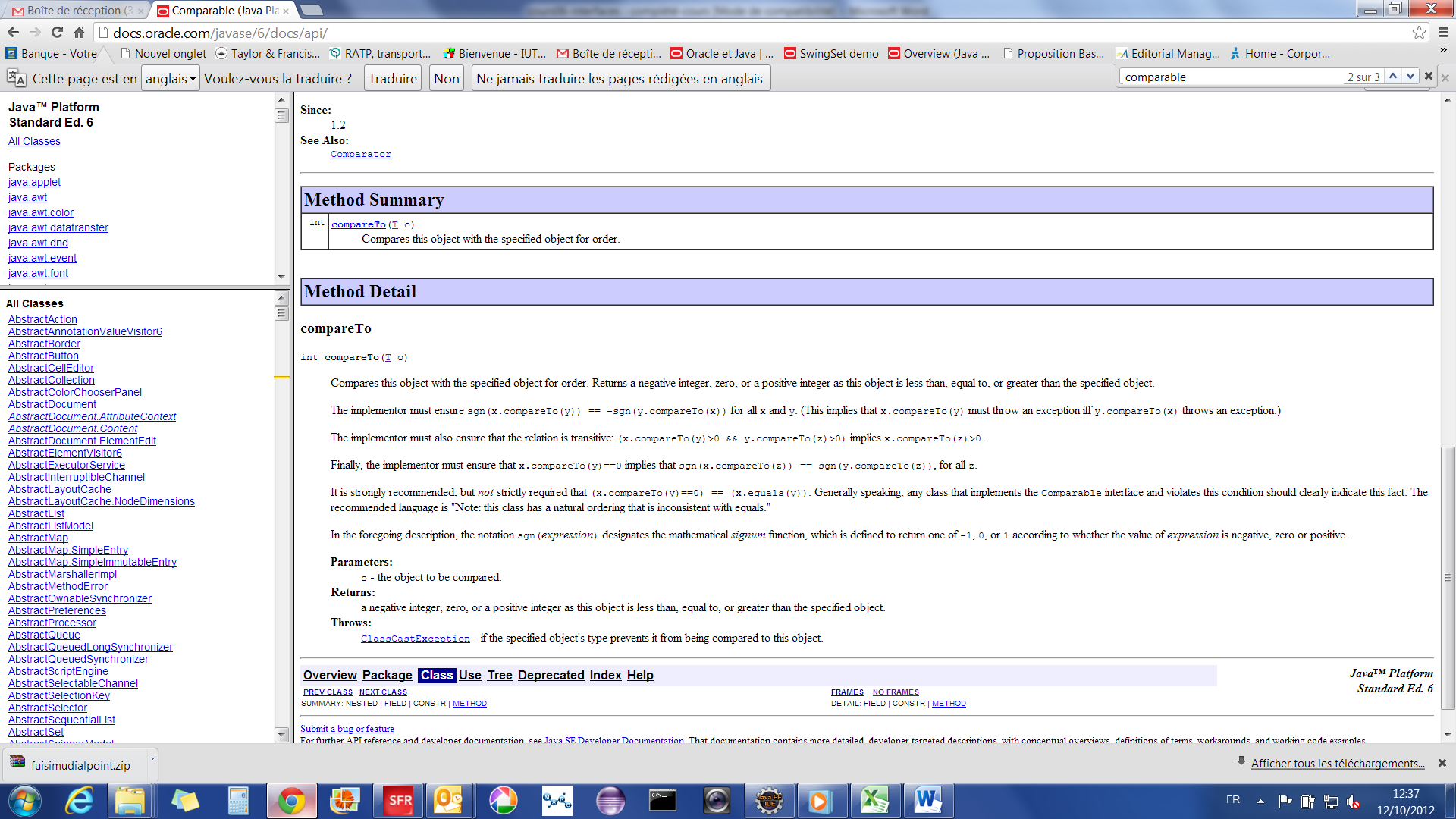
**3 6 1 -10 5**

**-10 1 3 5 6**

Ce serait bien que vous puissiez utiliser ces algorithmes avec vos propres classes : par exemple la classe Personne.

Le compilateur doit être sûr de pouvoir comparer deux objets. La classe des objets que vous voulez pouvoir trier doit implémenter l’interface *Comparable*.

La seule méthode de cette interface est *compareTo(Object autre)* qui renvoie un entier positif si l'objet est supérieur à l'autre, 0 s'ils sont égaux, et négatif sinon.



**import** java.util.\*;

**public** **class** Personne **…………………………………...…………………………………….**{

String prenom ;

String nom ;

**int** anneeNaissance ;

// Tester cette classe

**public** **static** **void** main (String args []) {

Personne p0 = **new** Personne ("Durand", "Albert", 1978);

Personne p1 = **new** Personne ("Acap", "Paul", 1975);

Personne p2 = **new** Personne ("Durand", "Albert", 1973);

Personne tp [] = **new** Personne [3];

tp [0] = p0 ;

tp [1] = p1 ;

tp [2] = p2 ;

System.*out*.println ("AVANT LE TRI :");

*afficher* (tp);

System.*out*.println ("TRI EN COURS ...");

Arrays.sort (tp) ;

System.*out*.println ("APRES LE TRI :");

*afficher* (tp);

}

// Constructeur

Personne (String nom, String prenom, **int** anneeNaissance) {

**this**.nom = nom ;

**this**.prenom = prenom ;

**this**.anneeNaissance = anneeNaissance ;

}

**// Pouvoir comparer deux personnes (ON CHOISIT SELON LEUR ANNEE DE NAISSANCE)**

// Retourner une version chaine de caracteres

**public** String toString () {

**return** nom.toUpperCase() + " " + prenom + " (" + anneeNaissance + ")";

}

// Afficher un tableau de personnes

**static** **void** afficher (Personne t[]) {

System.*out*.println ();

**for** (**int** i=0; i < t.length; i++)

System.*out*.println (t[i] + " ");

}

Résultat :

**AVANT LE TRI :**

**DURAND Albert (1973)**

**ACAP Paul (1975)**

**DURAND Albert (1973)**

**TRI EN COURS ...**

**APRES LE TRI :**

**DURAND Albert (1973)**

**DURAND Albert (1973)**

**ACAP Paul (1975)**

Questions :

Changer le critère de comparaison : trier sur le nom

(indice : aller voir la javadoc de String !)

Est-ce que Personne pourrait aussi hériter d'une classe ?

**………………………………………………………………………………………**

**………………………………………………………………………………………**

Quelle méthode est polymorphe ?

**………………………………………………………………………………………**

**………………………………………………………………………………………**

A quoi oblige le compilateur si l’on met implements Comparable ?

**………………………………………………………………………………………**

**………………………………………………………………………………………**

Qu’est-ce qui ne fonctionne pas si on n’indique pas implements COMPARABLE ?

**………………………………………………………………………………………**

**………………………………………………………………………………………**

Quelle méthode est appelée lors du tri dans la méthode Arrays.sort() ?

**………………………………………………………………………………………**

**………………………………………………………………………………………**

Pour aller plus loin

INTERFACES ET POLYMORPHISME

* Cf .PDF interfaces-polymorphisme-extrait-livre-DELECHAMP-LAUGIE

<https://drive.google.com/file/d/0B0eSWckcr0ijeXh3RU9YUS00UENFYlJnOXRieTBWb09TSzdF/view?usp=sharing>

PATRON DE CONCEPTION “STRATEGIE”

* Très bon exemple montrant comment le développeur peut éviter d’avoir à déplacer la définition d’une méthode d’une classe à l’autre durant un processus de conception itérative

<https://openclassrooms.com/courses/apprenez-a-programmer-en-java/les-classes-abstraites-et-les-interfaces#/id/r-21972>

* Définition du pattern “stratégie”

<https://fr.m.wikipedia.org/wiki/Strat%C3%A9gie_(patron_de_conception)>

En [génie logiciel](https://fr.m.wikipedia.org/wiki/G%C3%A9nie_logiciel), le patron stratégie est un [patron de conception](https://fr.m.wikipedia.org/wiki/Patron_de_conception) (design pattern) de type comportemental grâce auquel des algorithmes peuvent être sélectionnés à la volée au cours du temps d'exécution selon certaines conditions.

Le patron de conception stratégie est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application. Le patron stratégie est prévu pour fournir le moyen de définir une famille d'algorithmes, encapsuler chacun d'eux en tant qu'objet, et les rendre interchangeables. Ce patron laisse les algorithmes changer indépendamment des clients qui les emploient.

PASSAGE DE METHODES EN PARAMETRES

Contrairement à de nombreux langages de programmation, Java n’autorise pas de paramètres qui soient des méthodes. Les interfaces peuvent servir à simuler ce fonctionnement :

interface X{

void f(){

}

}

class C{

. . .

public void m(X x){

. . .

x.f() ;

}

}

class maClasse implements X{

. . .

public void f(){

. . .

}

}

C c = new C() ;

maClasse mc = new MaClasse() ;

c.m(mc) // la méthode m va appeler la méthode f

// définie dans maClasse.