Programmation Orienté Objet (POO)

Présenté par : M. Bouderbala

Promotion : 2ème Année LMD Informatique / Semestre N°4

Etablissement : Université de Relizane

Année Universitaire : 2021/2022

1. Classes abstraite (1)

- Une classe abstraite est une classe qui ne peut pas être instanciée. Elle ne peut servir que de classe de base pour être dérivée.

  abstract class A {……}

- Dans une classe abstraite on trouve des méthode habituelles et des champs (attributs) mais aussi des **méthodes dites abstraites.** C'est-à-dire, des méthodes dont on fournit uniquement la **signature** et le **type de la valeur de retour.**

```
abstract class A
{ public void f(){….}
public abstract void g (int n);
}
```

1. Classes abstraite (2)

- Comme nous l'avons vu dans un des chapitres précédents, les classes peuvent être considérées comme étant des moules (des modèles, des plans, ...) permettant de fabriquer des objets.

- Donc les classes abstraites peuvent être considérées comme étant des moules incomplets (des modèles partiels, des plans non terminés, ...) qui ne peuvent pas être utilisés tels quels pour créer des objets mais qui peuvent être utilisés pour fabriquer d'autres plans plus précis (représentés par des sous classes) qui seront complétés et qui permettront, eux, de créer des Objets (des instances des sous classes).

2. Classes abstraite: Les règles (1)

- Une classe qui comporte une ou plusieurs méthodes abstraites est abstraite, même si on n'indique pas le mot clé **abstract** à sa déclaration. Et de ce fait, elle ne sera pas **instanciable** (on ne peut pas créer d'objet en utilisant l'opérateur new).
- Une méthode abstraite doit obligatoirement être déclarée publique (puisqu'elle est destinée à être redéfinie dans une classe dérivée).
- Une classe dérivé d'une classe abstraite ne peut être instanciée (c.a.d devenire concrète) que si elle redéfinit chaque méthode abstraite de sa classe parente et qu'elle fournit une implémentation (un corps) pour chacune des méthodes abstrait

2. Classes abstraite: Les règles (2)

- Si une classe dérivé d'une classe abstraite n'implémente pas toutes les méthodes abstraites dont elle hérite, cette classe est elle-même abstraite (et ne peut donc pas être instanciée).
- Une classe dérivée d'une classe abstract n'est pas obligée de redéfinir toutes les méthodes abstraites de sa classe de base et peut même n'en redéfinir aucune et restera abstraite elle même.
- Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et / ou contenir des méthodes abstraites.
- Les méthodes déclarées avec l'un des modificateurs **static, private ou final** ne peuvent pas être abstraites étant donné qu'elles ne peuvent pas être redéfinies dans une sous classe.

2. Classes abstraite: Les règles (3)

- Une classe déclarée final ne peut pas contenir de méthodes abstraites car elle ne peut pas être dérivé.

```
public final class ClasseFinal {
    ...
}
```

- Une classe peut être déclarée abstract même si elle ne possède pas réellement de méthode abstraite.

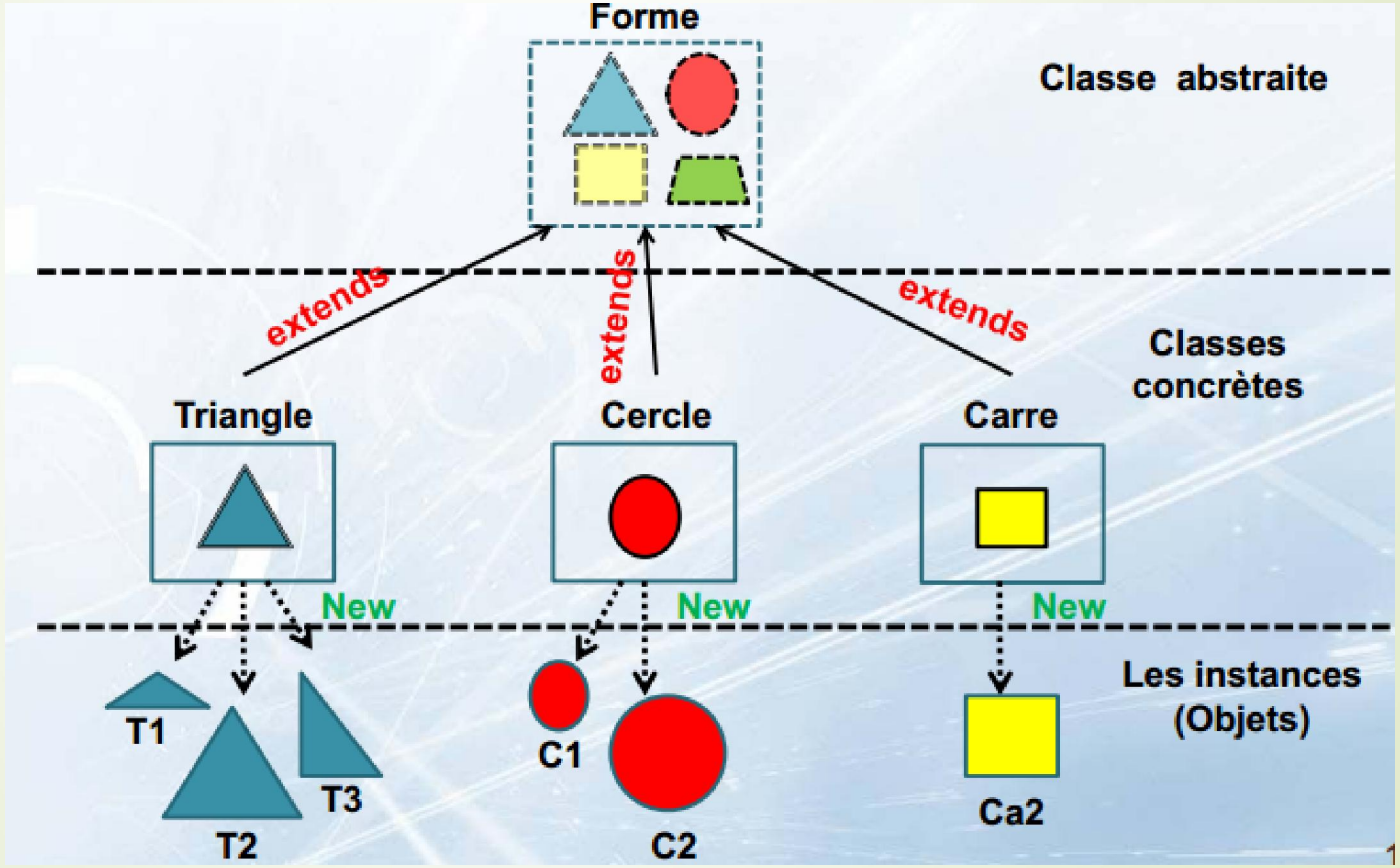
  Cela signifie que son implémentation est incomplète en certains points (p. ex. le corps de certaines méthodes doit être complété en fonction du contexte) et qu'elle est destinée à jouer le rôle de classe parente pour une ou plusieurs sous classes qui achèveront l'implémentation. Même si elle ne possède pas de méthodes abstraites, une telle classe ne peut pas être instanciée (erreur à la compilation)

3. Exemple de classe abstraite (1)

- On souhaite disposer d'une hiérarchie de classes permettant de manipuler des formes géométriques. On veut qu'il soit toujours possible d'étendre la hiérarchie en dérivant de nouvelles classes mais on souhaite pouvoir imposer que ces dernières disposent toujours des méthodes suivantes :
  - void rotation (double angle)
  - double surface()
  - double périmètre()

Quel solution proposez vous?

3. Exemple classe abstraite (2)



3. Exemple de classe abstraite (3)

- Il suffit d'appliquer les règles de définition d'une classe abstraite. On y place les en-têtes des méthodes qu'on souhaite voir redéfinies dans les classes dérivées, en leur associant le mot clé *abstract* :

```
abstract public class Forme // on peut enlevr le mot clé abstract
{
abstract public void rotation (double angle) ;
abstract public double surface() ;
abstract public double périmetre() ;
…}
```

3. Exemple de classe abstraite (4)

- Les classes ascendantes de la hiérarchie de Forme seront alors simplement définies comme classes dérivées de *Forme et elles devront définir les méthodes affiche et rotation,* par exemple :

```
class Cercle extends Forme
{
private Point centre;
private double rayon;

public double périmètre() { ..... }
public void rotation (double angle) { ..... }
public double surface() {……..}

}
```

3. Utilité des classes abstraites

- En pratique, les classes abstraites permettent de définir des fonctionnalités (des comportements) que les classes dérivés devront impérativement implémenter même si la classe abstraite n'est pas en mesure de fournir elle même une implémentation pour ces méthodes.
- Les utilisateurs des sous classes d'une classe abstraite sont donc assurés de trouver toutes les méthodes définies dans la classe abstraite dans chacune des sous classes concrètes.
- Les classes abstraites constituent donc une sorte de contrat (spécification contraignante) qui garantit que certaines méthodes seront disponibles dans les sous classes et qui oblige les programmeurs à les implémenter dans toutes les sous classes concrètes.
- Les classes abstraites facilitent la conception orientée objet car elles permettent de placer dans une classe toutes les fonctionnalités dont doivent disposer toutes ses descendantes.

Les interfaces

4. Les interfaces (1)

- On vient de voir comment une classe abstraite permettait de définir dans une classe de base des fonctionnalités communes à toutes ses descendantes, tout en leur imposant de redéfinir certaines méthodes.
- Si l'on considère une classe abstraite n'implantant aucune méthode et aucun champ (hormis des constantes), on aboutit à la notion d'interface.
- Comme une classe et une classe abstraite, une interface permet de définir un nouveau type (référence).

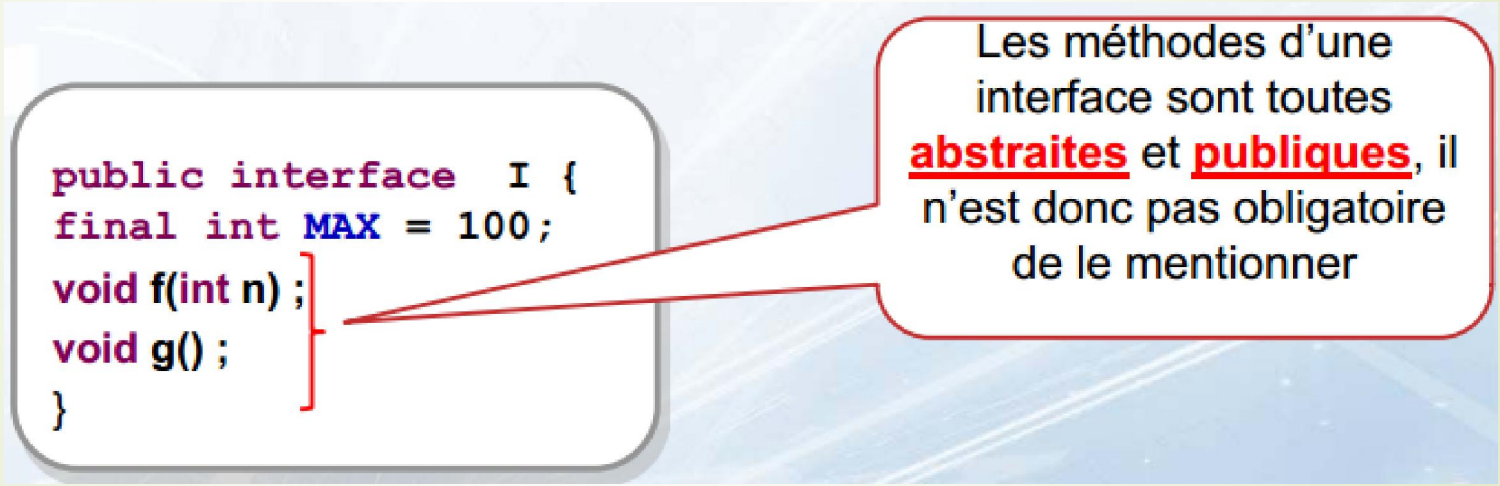
- En effet, une interface définit les en-têtes d'un certain nombre de méthodes ainsi que de constantes fonctionnalités dont doivent disposer toutes ses descendantes

4. Les interfaces (2)

- Une interface constitue essentiellement une spécification qui laisse de côté les détails d'implémentation.
- L'interface définit ainsi une sorte de contrat que les classes qui l'implémenteront s'engagent à respecter.
- Les interfaces jouent un rôle important dans la phase de conception (API) d'applications, de librairies ou de systèmes basés sur des ensembles d'objets qui communiquent.
- La notion d'interface est plus riche que celle de classe abstraite:
  - Une classe peut implémenter plusieurs interfaces (alors qu'une classe ne peut dériver que d'une seule classe abstraite)
  - La notion d'interface se superpose à celle de dérivation et ne la remplace pas.
  - Les interfaces peuvent se dériver
  - On peut utiliser des variables de type interface

4. Les interfaces (3)

- La définition d'une interface est proche de celle d'une classe abstraite. Il suffit de remplacer les mots clés **abstract class** par le mot clé **interface**:



4. Les interfaces (4)

Une classe peut implémenter une interface

```
class A implements I {
// A doit redéfinir les méthodes de I sinon
erreur à la compilation
}
```

Une classe peut implémenter plusieurs interfaces

```
public interface  I1 {
void f() ;
}
public interface  I2 {
void g() ;
}
class A implements I1,I2 {
// A doit redéfinir les méthodes f et g
}
```

4. Les interfaces (5)

- On ne peut pas créer des objets de type interface. On peut définir des références. Elles feront référence à une instance d'une classe qui implante l'interface

```
public interface I { …. }
….
I i;          //i est une référence à un objet d'une classe implémentant
              //l'interface I

class A implements I {….}
….
I i = new A(…); // ok
```

5. Interface et classe dérivée

- Le mot clé **implements** est une garantie de la part d'une classe d'implémenter toutes les méthodes proposées dans une interface. Elle est totalement indépendante de l'héritage. Donc une classe dérivée peut implémenter une ou plusieurs interfaces.

Exemple 1:

```
interface I
{
 void f(int n);
 void g();
}
class A {…..}
class B extends A implements I
{// les méthodes f et g doivent
   être soit déjà définies dans A
   soit définies dans B
}
```

Exemple 2:

```
interface I1{….}
interface I2{….}
class A implements I1{….}
class B extends A implements I2
   {…}
```

6. Dérivation d'une interface (1)

- On peut définir **une interface** comme étant **dérivée d'une autre interface** (mais pas d'une classe) en utilisant le mot clé *extends*. La dérivation d'interfaces revient simplement à concaténer les déclarations et n'est pas aussi riche que celles des classes.

```
interface I1
{
static final MAXI = 100;
void f (int n);
}
interface I2 extends I1
{
static final MINI = 10;
void g();
}
```

```
interface I2
{
static final MINI = 10;
static final MAXI = 100;
void f (int n);
void g();
}
```

6. Dérivation d'une interface (2)

- Une interface peut dériver de plusieurs interfaces.
- L'héritage multiple est autorisé pour les interfaces

```
interface I1
{ void f();
}
interface I2
{ void f1();
}
interface I3 extends I1,I2
{ void f2();
}
```

7. Conflits de noms (1)

- Des conflits de noms (collision) peuvent se produire lorsqu'une classe implémente plusieurs interfaces comportant des noms de méthodes ou de constantes identiques. Il faut distinguer plusieurs situations :
- Plusieurs méthodes portent des signatures identiques :
  - Pas de problème, la classe doit implémenter cette méthode
- Mêmes noms de méthodes mais profils de paramètres différents
  - Implémentation de deux ou plusieurs méthodes surchargées
- Mêmes noms de méthodes, profils de paramètres identiques, mais types des valeurs de retour différents :
  - Pas possible d'implémenter les deux interfaces (cela provoquera une erreur à la compilation).
- Noms de constantes identiques dans plusieurs interfaces :
  - Doivent être accédées en utilisant la notation qualifiée (Interface.Constante)

7. Conflits de noms (2)

Exemple 1

```
interface I1
{void f (int n);
void g();
}
interface I2
{void f(float x);
void g();
}
class A implements I1,I2
{/* Pour satisfaire I1 et I2, A doit définir
  deux méthodes f: void f(int) et void
  f(float), mais une seule méthode g: void
  g()*/}
```

7. Conflits de noms (3)

Exemple 2

```
interface I1
{void f (int n);
void g();
}
interface I2
{void f(float x);
int g();
}
class A implements I1,I2
{/* Pour satisfaire I1 et I2, A doit contenir à la
   fois une méthode void g() et une méthode int g(),
   ce qui est impossible d'après les règles de
   redéfinition. Une classe ne peut implémenter les
   interfaces I1 et I2 telles qu'elles sont définies
   dans cet exemple*/
```

8. Exemple d'interface (1)

- Supposons que nous voulions que les classes dérivées de la classe Forme disposent toutes d'une méthode afficher(), permettant d'imprimer les formes géométriques.
- Il serait naturellement possible d'ajouter une méthode abstraite afficher() à la classe Forme et ainsi chacune des sous classes concrètes devrait implémenter cette méthode.
- Avec cette solution, le problème serait résolu pour toutes les sous classes de Forme, mais si d'autres classes (qui n'héritent pas de Forme) souhaitaient également disposer des fonctions d'impression, elles devraient à nouveau déclarer des méthodes abstraites d'impression dans leur arborescence.
- L'utilisation d'une interface permet de résoudre ce problème de manière plus élégante en découplant la fonctionnalité d'impression de la hiérarchie des classes

8. Exemple d'interface (2)

- Au lieu d'ajouter une méthode abstraite afficher() dans la classe Forme nous créons une interface nommée Affichage qui définit la méthode abstraite afficher():

```
public interface  Affichage {
void afficher() ; }

abstract public class Forme implements Affichage {
  abstract public void rotation (double angle) ;
  abstract public double surface() ;
  abstract public double périmetre() ;
}
public class Cercle extends Forme {
  private Point centre;
  private double rayon;
  public double périmètre() { ..... }
  public void rotation (double angle) { ..... }
  public double surface() {..... ...}
  public void afficher(){.....}
```

8. Exemple de l'interface comparable (1)

- Une classe implémente l'interface *comaprable* si ses objets peuvent être ordonnés selon un ordre particulier.
- Par exemple la classe String implémente Comparable parce que les chaines de caractères peuvent être ordonnées selon l'ordre alphabétiques.
- Les classes numériques comme Integer et Double implémentent Comparable parce que les nombres peuvent être ordonnés selon l'ordre numérique .

8. Exemple de l'interface comparable (2)

- L'interface Comparable consiste en une seule méthode (et pas de constantes) int compareTo(T obj)qui compare l'objet à un objet de type T .
  interface Comparable {
  int compareTo(Object o);
  }
- A.compareTo(B)retourne un entier négatif si l'objet A est plus petit que B, zéro si les deux objets sont égaux et un entier positif si l'objet A est plus grand que l'objet B.
- Exemple:
  "Bonjour ".compareTo ("Bonsoir ") renvoie un entier négatif.

8. Différences entre interface et héritage

- Une interface fournit un contrat à respecter sous forme d'en-tête de méthodes auxquelles la classe implémentant l'interface s'engage à fournir un corps (qui peut être vide ou contenir des instructions).
- Des classes différentes peuvent implémenter différemment une même interface alors que les classes dérivées d'une même classe de base partagent la même implémentation des méthodes héritées
- Même si Java ne dispose pas de l'héritage multiple, ce dernier peut être remplacé par l'utilisation d'interfaces, avec l'obligation d'implémenter les méthodes correspondantes.