Chapitre 1

L'héritage et l'interface

L'héritage des classes
La surcharge des méthodes à travers l'héritage
Les classes génériques
Les collections polymorphes
Création et utilisation des interfaces
Les traitements génériques grâce aux interfaces

1. Introduction

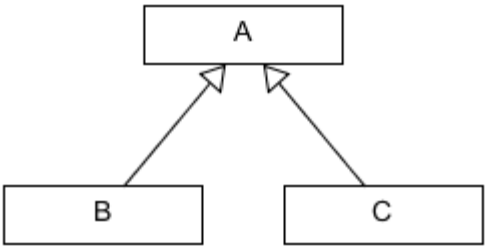
Le concept d'héritage est le cœur de la programmation objet. Le langage JAVA est un langage orienté objet. Il utilise donc l'héritage et les concepts qui en découlent dans toutes ces APIs.

Ainsi, il est très difficile d'utiliser les classes des API prédéfinies Java sans maitriser les concepts associés à l'héritage.

Ces concepts que nous allons apprendre sont :

- **l'héritage** des classes qui entraîne un héritage des attributs, et des méthodes
- les **classes abstraites** qui permettent "historiquement" le polymorphisme et la généricité des traitements
- **l'interface** qui remplace efficacement mais partiellement les classes abstraites et permet aussi l'implémentation de traitement générique
- le **polymorphisme** des collections (une application de la généricité)
- la réalisation de traitement **générique**

L'héritage est une relation qui lit les classes entre elles.
Cette relation peut se représenter sous une forme graphique.
Symbole : 



On dit que les classes B et C héritent de la classe A.
On dit que les classes B et C sont des **classes dérivées** de A.

On dit que B "**est un**" A.
On dit que C "**est un**" A.

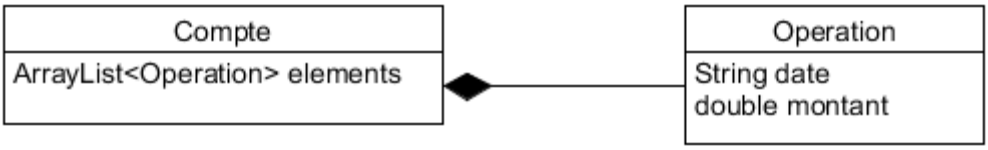
En java :

```
public class B extends A {…}

public class C extends A {….}
```

Cette notation est celle utilisée dans les diagrammes de classes du formalisme UML.

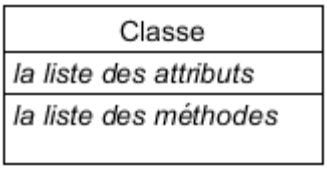
Nous en profiterons aussi pour utiliser les notations UML suivantes :

- la relation de composition :

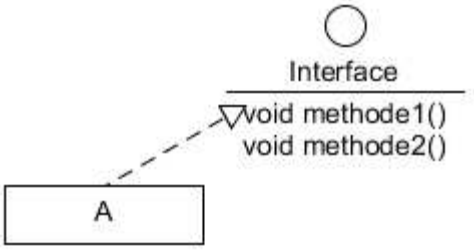


- la relation d'agrégation :

- la représentation d'une classe



- l'implémentation d'une interface



En java :

```
public interface Interface
{
    public void methode1();
    public void methode2();
}
```

```
public class A implements Interface {…}
```

Ajouter un exemple de code

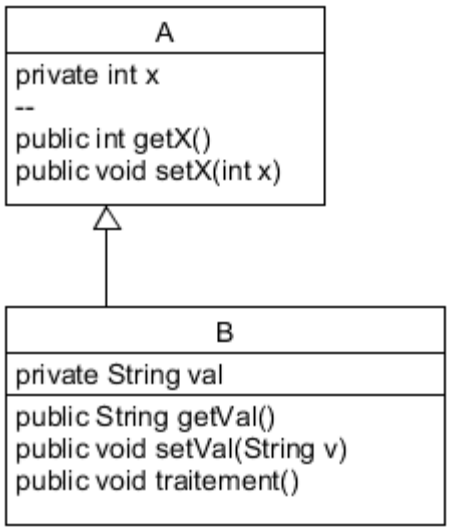
Ajouter un exemple de héritage/généricité / polymorphisme

2. Définition théorique de l'héritage

B hérite de A signifie que :

- B hérite des **attributs** de A et que
- B hérite des **méthodes** de A.

Soit l'héritage suivant :



L'héritage des attributs signifie que si on fait dans du code l'instruction :

```
B b = new B()
```

alors cela signifie que b pointe vers une zone mémoire contenant la réunion de tous les attributs de B et de A.

Par défaut le constructeur de B appelle le constructeur A().

On dit que B hérite des attributs de A. Mais, attention, si les attributs de A sont private alors les méthodes de B ne pourront pas utilisées directement ces attributs.

Généralement, pour que B puissent utilisée directement les attributs de A, on les déclarent **protected** (voir plus loin).

Pour du code qui utilise la classe B :

```
B b = new B()
String s = b.getVal()
b.setVal("TOTO")
```

L'héritage des méthodes signifie qu'on peut écrire le code ::

```
int a = b.getX()
b.setX(12)
```

Dans le code de la méthode **traitement** de B on peut écrire :

```
public void traitement()
{
     val = "TOTO";  // Normal
     setX(100);       // méthode de A (la méthode doit être public)
     int v = getX();  // méthode de A (la méthode doit être public)
}
```

Si l'attribut x de A est **public** alors on peut écrire

```
B b = new B();
b.x = 100;
```

```
public void traitement()
{
     val = "TOTO";  // Normal
     x = 100;
}
```

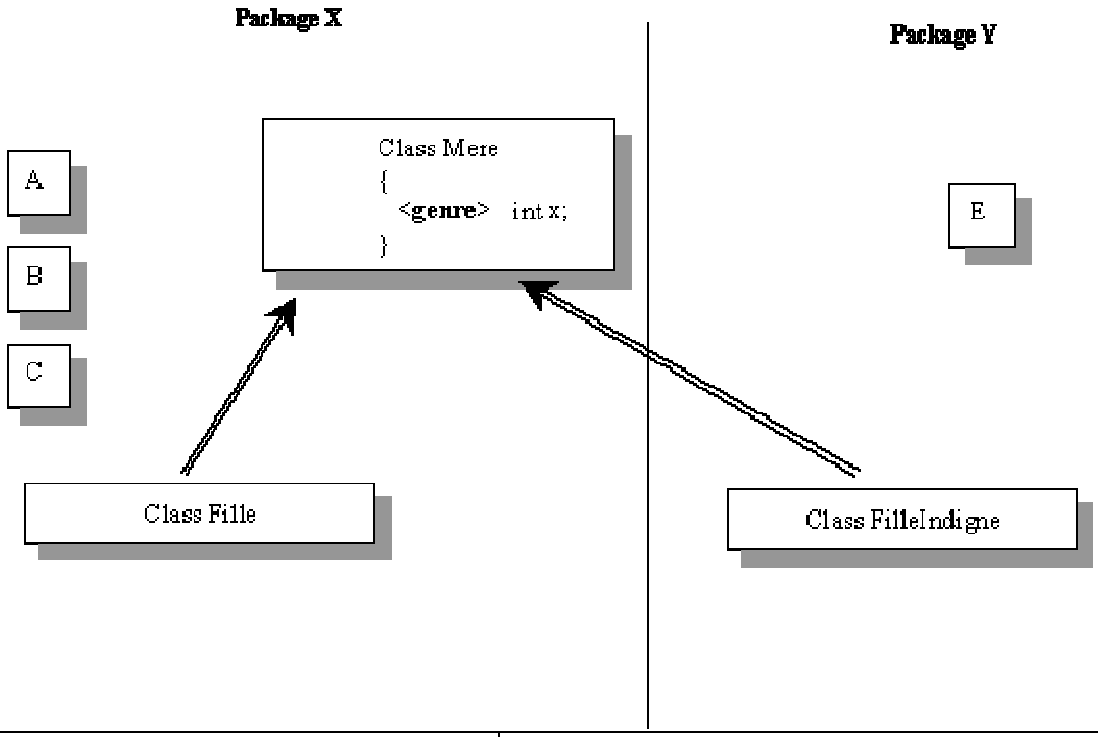
Si l'attribut x de A est **protected** alors on peut aussi écrire :

```
Dans une classe C et si C appartient au même package que B
public C {
    public void trt()
    {
       B b =new B();
       b.x = 100;
    }
}
```

Et aussi, même si B n'appartient pas au même package que A :

```
public void traitement()
{
     val = "TOTO";  // Normal
     x = 100;
}
```

Rappel de la visibilité d'un attribut en Java :



genre	x est visible dans les classes
private	Mere
public	Toutes
protected	Mere, Fille, FilleIndigne, A, B, C
default *(vide)*	Mere, Fille, A, B, C

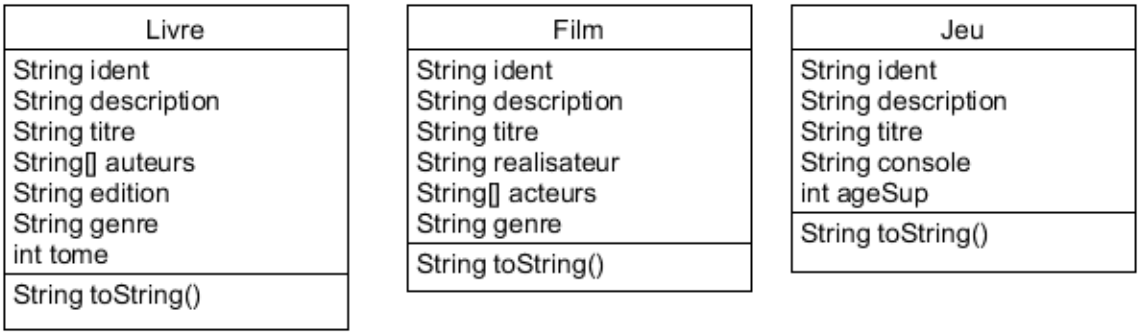
3. La factorisation des attributs

3.1. Introduction

Le premier usage de l'héritage est de factoriser les attributs qui sont communs dans différentes classes.

On appelle cela le principe de généralisation.

Prenons un exemple de 3 classes suivantes :



Toutes ces classes définissent les getteurs, setteur et le constructeur permettant d'initialiser tous les attributs de la classe :

```
  public Livre(String ident, String description, String titre, String[]
auteurs, String edition, String genre, int tome)
  {
     this.ident = ident;
     this.description = description;
     this.titre = titre;
     this.auteurs = auteurs;
     this.edition = edition;
     this.genre = genre;
     this.tome = tome;
  }

  // Constructeur simplifié
  public Livre(String ident, String description, String titre,
       String auteur)
  {
     this.ident = ident;
     this.description = description;
     this.titre = titre;
     auteurs = new String[1];
     auteurs[0] = auteur;
     this.edition = "";
     this.genre = "";
     this.tome = 0;
  }

  // Constructeur par défaut sans paramètre
  public Livre()
  {
     this.ident = "IDENT INCONNUE";
     this.description = "";
     this.titre = "SANS TITRE";
     auteurs = new String[1];
     auteurs[0] = "AUTEUR INCONNU";
     this.edition = "";
     this.genre = "";
     this.tome = 0;

  }
```

```
  public Film(String ident, String description, String titre, String
realisateur, String[] acteurs, String genre)
  {
```

```
    this.ident = ident;
    this.description = description;
    this.titre = titre;
    this.realisateur = realisateur;
    this.acteurs = acteurs;
    this.genre = genre;
  }
```

```
  public Jeu(String ident,String description, String titre, String console,
int ageSup)
  {
    this.ident = ident;
    this.description = description;
    this.titre = titre;
    this.console = console;
    this.ageSup = ageSup;
  }
```

Toutes ces classes définissent aussi la méthode :
String toString() qui affiche tous les attributs de l'objet.

Dans la classe Livre :

```
  public String toString()
  {
    String a=""; for(String s:auteurs)a=a+s;
    return
      ident+" "+description+" "+titre+" "+a+" "+edition+" "+genre+
      " "+tome;
  }
```

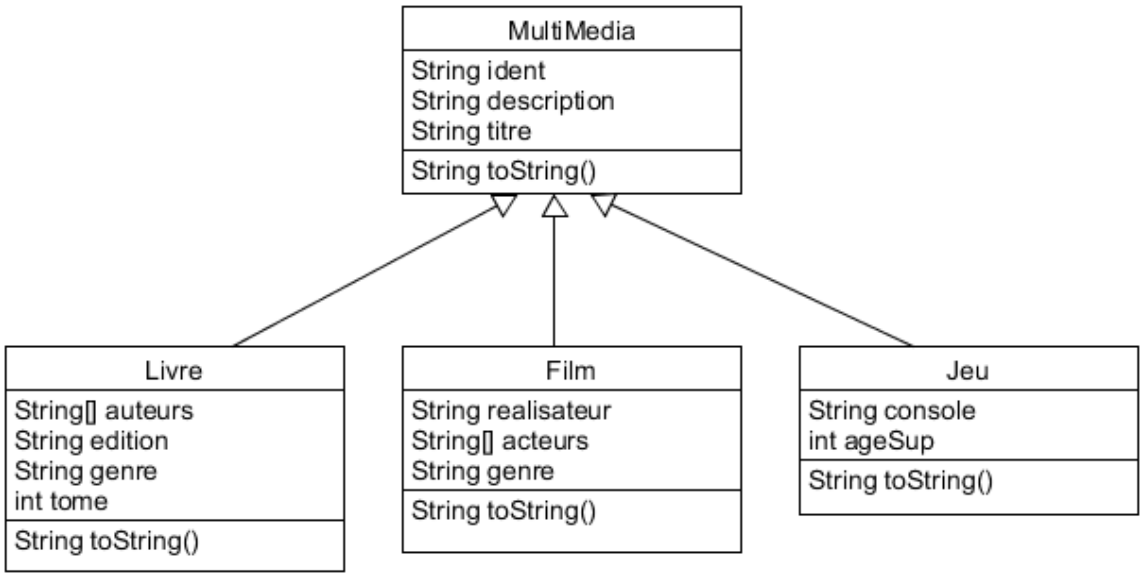
Dans la classe Film :

```
  public String toString()
  {
    String a=""; for(String s:acteurs)a=a+s;
    return
      ident+" "+description+" "+titre+" "+realisateur+" " +
       a+" "+genre;
  }
```

Dans la classe Jeu :

```
  public String toString()
  {
    return
      ident+" "+description+" "+titre+" "+console+" "+ageSup;
  }
```

3.2. Factorisation par héritage



La nouvelle classe MultiMedia contient les attributs communs.
Les getteurs et les setteurs des attributs se répartissent dans chacune des classes.

On crée le constructeur de la classe MultiMedia :

```
public MultiMedia(String ident, String description, String titre)
{
   this.ident = ident;
   this.description = description;
   this.titre = titre;
}

public MultiMedia()
{
   this.ident = "INCONNU";
   this.description = "RIEN";
   this.titre = "PAS de TITRE";

}
```

3.3. Mais que deviennent les constructeurs des autres classes ?

```
  public Livre(String ident,String description, String titre, String[]
auteurs, String edition, String genre, int tome)
  {
     super(ident,description,titre);
     this.auteurs = auteurs;
     this.edition = edition;
     this.genre = genre;
     this.tome = tome;
  }

  public Livre(String ident,String description, String titre,
    String auteur)
  {
     super(ident,description,titre);
     auteurs = new String[1];
     auteurs[0] = auteur;
     this.edition = "";
     this.genre = "";
     this.tome = 0;
  }

  public Livre(String auteur)
  {
     //super();  // PAR DEFAUT JAVA APPEL LE CONSTRUCTEUR Super
     auteurs = new String[1];
     auteurs[0] = "AUTEUR INCONNU";
     this.edition = "";
     this.genre = "";
     this.tome = 0;
  }
```

```
  public Film(String ident,String description, String titre, String
realisateur, String[] acteurs, String genre)
  {
     super(ident,description,titre);
     this.realisateur = realisateur;
     this.acteurs = acteurs;
     this.genre = genre;
  }
```

```
  public Jeu(String ident,String description, String titre, String console,
int ageSup)
  {
     super(ident,description,titre);
     this.console = console;
     this.ageSup = ageSup;
  }
```

La règle :
On doit appeler dans un constructeur le constructeur de la classe dont on hérite.

En JAVA, le mot **super** correspond à **l'objet hérité** (à l'opposé de this)

Attention :

Si on n'utilise pas l'appel explicite au constructeur de la classe héritée alors par défaut l'instruction **super()** est utilisé par Java (voir exemple avec Media() plus haut)

3 cas :

- si le constructeur père sans paramètre n'a pas été défini et qu'il existe un constructeur avec des paramètres alors il y aura une erreur de compilation.
- si le constructeur père sans paramètre est défini alors ce sera ce constructeur qui sera appelé
- si aucun constructeur n'est défini dans la classe père alors ce sera le constructeur par défaut de Java qui sera utilisé.

La ligne contenant l'appel à super d'un constructeur est OBLIGATOIREMENT la première ligne du code de la méthode.

3.4. Et que deviennent les méthodes toString ?

Dans la classe MultiMedia :

```
public String toString()
{
   return
     ident + " " + description + " " + titre;
}
```

Dans la classe Livre :

```
public String toString()
{
   String a=""; for(String s:auteurs)a=a+s;
   return
     super.toString() + " "+a+" "+edition+" "+genre+
     " "+tome;
}
```

Dans la classe Film :

```
public String toString()
{
   String a=""; for(String s:acteurs)a=a+s;
   return
     super.toString() +" "+realisateur+" " +
      a+" "+genre;
}
```

Dans la classe Jeu :

```
public String toString()
{
   return
     super.toString() + " "+console+" "+ageSup;
}
```

On voit ici l'avantage de l'héritage :
si on ajoute un nouvel attribut dans la classe Multimedia alors il suffit de modifier la méthode toString de la classe MultiMedia et tout naturellement les autres méthodes toString par "héritage" utiliseront la nouvelle version de la classe.

Mais si on ajoute un attribut, l'impact sur les constructeurs reste important puisqu'il faut ajouter ce nouvel attribut en paramètre de tous les constructeurs des classes Livre, Film et Jeu, et impacter tous les appels à super de la classe MultiMedia.
Sauf si cet attribut est un attribut avec une valeur par défaut. Ensuite un peut utiliser le setteur de cet attribut pour renseigner la valeur si besoin.
Exemple : si on ajoute la propriété de prêt d'un multi-média, on peut ajouter un attribut **String emprunteur** à la classe MultiMedia qui par défaut sera initialisé à null (pas emprunté) et on ajoute la méthode **public void emprunter(String emprunteur){this.emprunteur = emprunteur;}**

Tout le code ci-dessus et dans un exemple du site.
Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple13_Heritage**

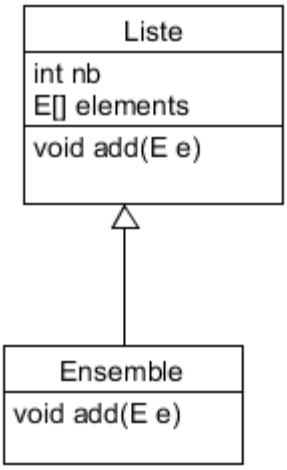
Ainsi, on a vu que le principe de généralisation algorithmique d'un traitement se fait par :
- centraliser le traitement dans la classe père, ou
- répartir le traitement dans toutes les classes et utiliser le principe de l'héritage pour reconstituer la généralité du traitement.

3.5. La surcharge des méthodes

On peut remarquer que la méthode toString de la classe Livre (et Film, Jeu) surcharge la méthode toString de la méthode MultiMedia.
On appelle cela le mécanisme de surcharge.

On peut donc re-écrire complètement une méthode dont on hérite.
On appelle cela le principe de spécialisation.

Exemple de la classe Ensemble qui hérite de la classe Liste et qui surcharge la méthode add afin de ne pas ajouter 2 fois le même élément.



Faire l'exercice :

Voir sur le site http://jacques.laforgue.free.fr **Exercice16_EnsembleString**

4. La classe Object de Java

En Java, toutes les classes héritent de la classe prédéfinie: **Object**. (package **lang**)

La classe Object est composée de méthodes prédéfinies que chaque classe peut **surcharger** afin que certains traitement prédéfinis de java appelle la méthode de la classe plutôt que celle de la classe Object qui généralement ne fait rien ou très peu (On appelle cela la généricité, voir plus loin).

Les méthodes les plus souvent surchargées sont : **clone**, **equals** et **toString**.

Les méthodes pratiques sont : **getClass** qui retourne la classe d'appartenance de tout objet.

Tout objet en Java peut être utilisé comme un verrou (programmation parallèle et asynchrone) : **wait**, **notify**, **notifyAll**.

La méthode **finalize** est le destructeur complémentaire.

La méthode **hashCode** est utilisée pour les tables de hashing et permet d'avoir un identifiant unique de tout objet.

5. Les collections polymorphes

5.1. Définition

Le polymorphisme est la possibilité d'utiliser des instances de classes différentes dans un tableau ou une collection Java :

```
collection.add( new MultiMedia())
collection.add( new Livre(…))
collection.add( new Film(…))
collection.add( new  Jeu(…))
```

Mais de quel type doit être les éléments de la collection :

```
ArrayList< ???? > collection;
```

Il faut pouvoir remplacer le type **????** par un élément du langage qui traduise que cela peut être soit un **MultiMedia**, soit un **Livre**, soit un **Film** ou soit un **Jeu**.

La solution : la classe **MultiMedia,** car le lien d'héritage traduit la relation "**est un**".
Un Livre est un MultiMedia, un Film est un MultiMedia, un Jeu est un MultiMedia.

Cela a du sens si on réalise des traitements sur l'ensemble des éléments de la collection.
Ces traitements utilisent des méthodes "communes" aux différentes classes des éléments de la collection.

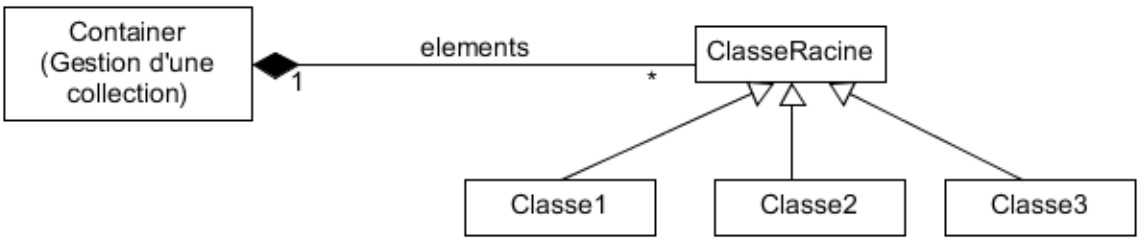
Ces méthodes "commune" sont soit :

- des méthodes appartenant à une classe commune et héritante (exemple la méthode toString de la classe MulitiMedia)
- des méthodes abstraites appartenant à une classe abstraite commune et héritante (voir plus loin)
- des méthodes appartenant à une interface dont la référence est connue des traitements (paramètre ou attribut) (voir plus loin)

Cela permet une propriété essentielle dans la conception des programmes informatiques : l' évolutivité.

Car une fois que ces traitements sont écrits, il est inutile de les modifier quand on ajoute de nouvelles classes qui appartiennent au même héritage.

Cela se représente ainsi :



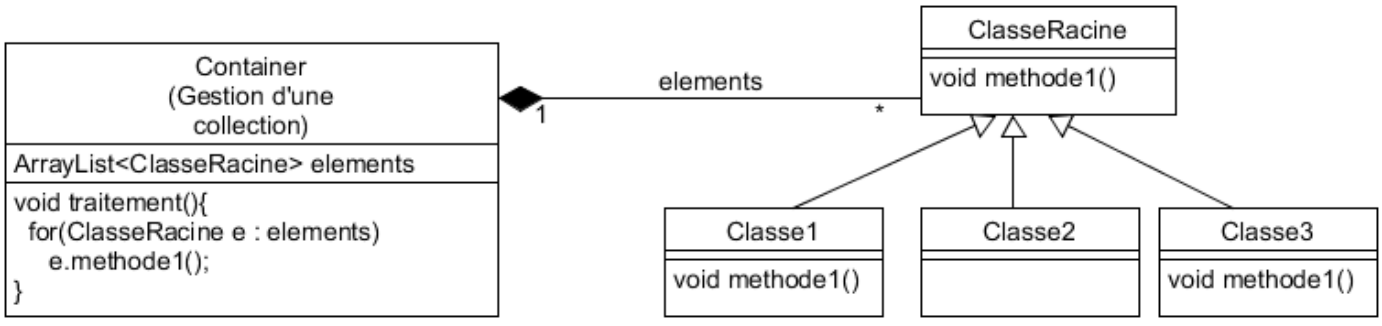
avec dans du code de Container

```
elemnents.add( new Classe1() )
elemnents.add( new Classe2() )
elemnents.add( new Classe3() )
```

Le compilateur est intraitable : un traitement de la classe Container ne peut utiliser que des méthodes qui ont une définition dans la classe ClasseRacine.

Si ces méthodes sont surchargées par les classes dérivées alors ce sont celles-ci qui seront appelées.

Exemple théorique :



avec

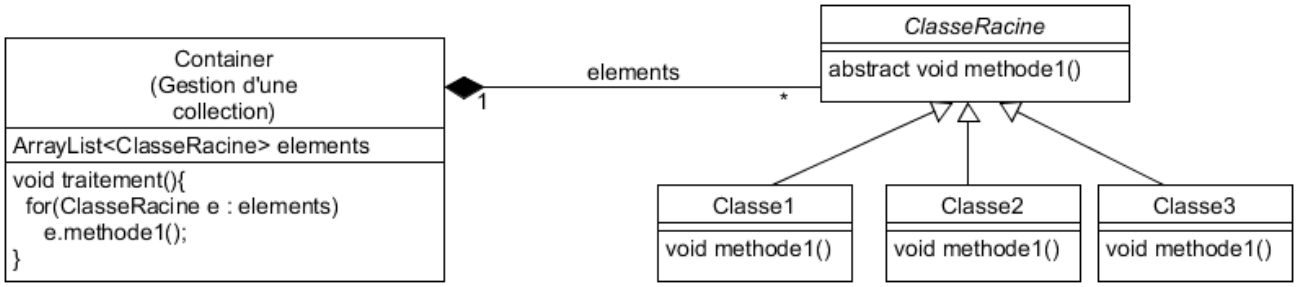
```
elemnents.add( new Classe1() )
elemnents.add( new Classe2() )
elemnents.add( new Classe3() )
```

L'exécution de traitement fait successivement :

- l'appel de la méthode methode1 de la classe Classe1
- l'appel de la méthode methode1 de la classe ClasseRacine
- l'appel de la méthode methode1 de la classe Classe3

Remarque : Dans certains langage objet comme le C++, il faut que la classe ClasseRacine soit une classe abstraite et la méthode methode1 de la classe ClasseRacine est abstraite (ou virtual en C++).

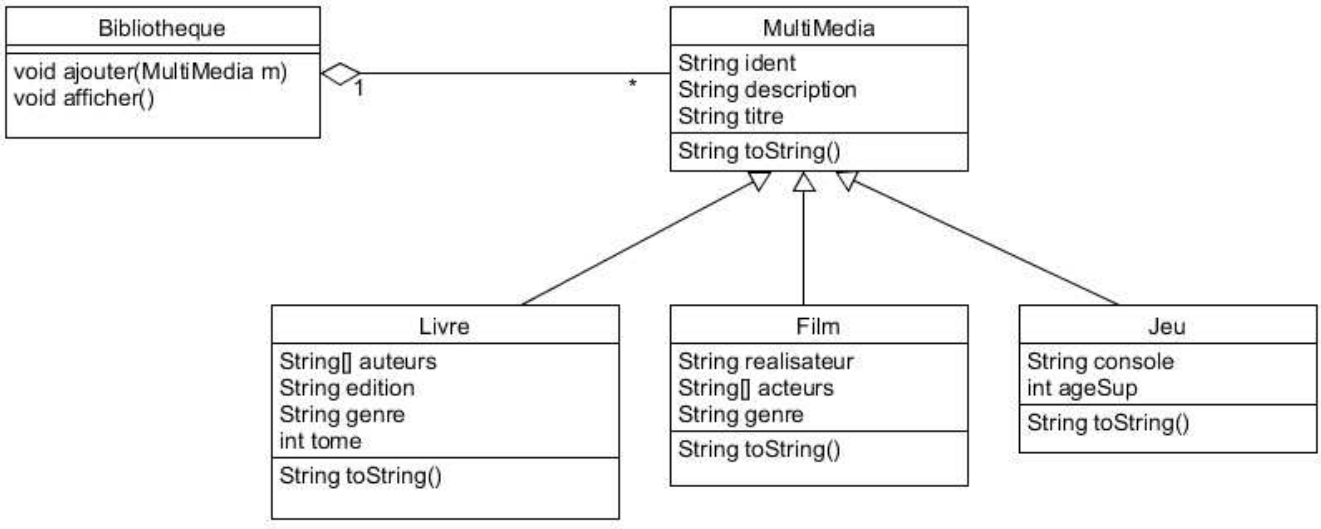
On dit que Java est un langage à liaison dynamique (explication en cours).



Ici la classe ClasseRacine est abstraite (voir plus loin l'utilisation des classes abstraites).

5.2. Exemple

En continuant l'exemple précédent sur les Livre, Film et Jeu :




```
  public class Exemple13
  {

      public static void main(String... args)
      {
          Livre l1 = new Livre("2012-02-0001",
                               "Livre de SF, appartenant au cycle de
Trantor",
                               "Face aux feux du soleil",
```

```
                                "Isaac Asimov"
                                );
        System.out.println(l1);

        Film f1 = new Film("2012-02-0002",
                           "Film","Les visiteurs 3","Jean-Marie Poire",
                           "Comedie")
            .addActeur("Jean Reno")
            .addActeur("Christian Clavier");
        System.out.println(f1);

        Jeu j1 = new Jeu("2012-02-0003",
                         "Jeu de voiture",
                         "Karting III",
                         "PC",
                         11);
        System.out.println(j1);

        Bibliotheque biblio = new Bibliotheque();
        biblio.ajouter(l1);
        biblio.ajouter(f1);
        biblio.ajouter(j1);

        biblio.afficher();
    }
}
```

```
public class Bibliotheque
{
    private ArrayList<MultiMedia> elements;

    public Bibliotheque()
    {
        elements = new ArrayList<MultiMedia>();
    }

    public void ajouter(MultiMedia media)
    {
        elements.add(media);
    }

    public void afficher()
    {
        System.out.println("------ AFFICHAGE DE LA BIBLIOTHEQUE ----");
        for(MultiMedia m:elements)
            {
                System.out.println("-----------------");
                System.out.println(m.toString());
            }
    }
}
```

Résultat de l'exécution :

```
------ AFFICHAGE DE LA BIBLIOTHEQUE ----
-----------------
ident           : 2012-02-0001
description     : Livre de SF, appartenant au cycle de Trantor
titre           : Face aux feux du soleil
auteur          : Isaac Asimov
```

```
edition         :
genre           :
tome            : 0

-----------------
ident           : 2012-02-0002
description     : Film
titre           : Les visiteurs 3
realisateur     : Jean-Marie Poire
acteur          : Jean Reno
acteur          : Christian Clavier
genre           : Comedie

-----------------
ident           : 2012-02-0003
description     : Jeu de voiture
titre           : Karting III
console         : PC
ageSup          : 11
```

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple13_Heritage**

6. Les classes abstraites

6.1. Définition

Une classe abstraite est une classe qui est déclarée abstraite.
Une classe qui contient au moins une **méthode abstraite** doit être déclarée abstraite.

Une méthode abstraite est une méthode qui n'a pas de code !!! (si si !)

Cela implique qu'une classe abstraite ne peut pas être instanciée, mais cela ne veut pas dire que la classe n'a pas de constructeur.
Cela implique que ce constructeur ne peut être appelé que par les classes dérivées (via super).

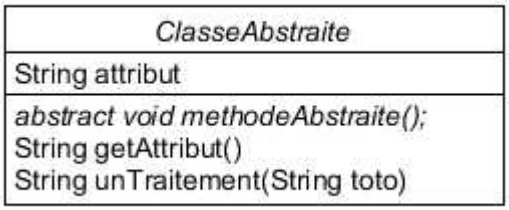
Une classe abstraite peut donc contenir des attributs réels et des méthodes réelles (toutes les méthodes ne son pas abstraites).

En Java, la syntaxe est la suivante :

```
abstract public class ClasseAbstraite
{
     private String attribut;

     public ClasseAbstraite(String a)
     {
       attribut = a;
     }
```

```
        abstract public void methodeAbstraite();

        public String getAttribut(){return attribut;}

        public String unTraitement(String toto)
        {
            return "exemple d'une methode relle " + toto;
        }

}
```

Comment on représente une classe abstraite en UML :

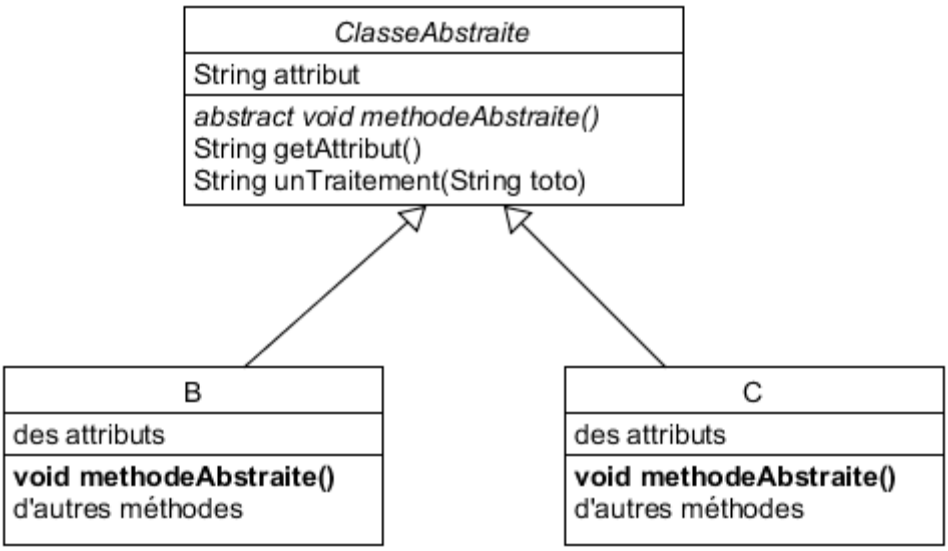


Le nom de la classe est en italique mais il est préférable de mettre dans le nom de la classe Abstract (quand on écrit sur une feuille ce n'est pas facile de faire du gras, de l'italique).
La méthode abstraite est en italique mais il vaut mieux mettre le mot abstract devant.

Pourquoi créer des classes abstraites ?
Pour obliger la classe qui hérite de la classe abstraite **d'implémenter** cette méthode. Mais on le verra plus loin, plus que ça.

On dit que l'on implémente les méthodes abstraites (au lieu de dire que l'on "sucharge").
Et qui implémente ?
La classe dérivée bien sur.

On obtient le schéma suivant :



Voilà pour la théorie, voyons des exemples

6.1. Un traitement générique

On a vu qu'un traitement "classique" en informatique peut prendre en entrée des données, que l'on a appelées les paramètres de la méthode.

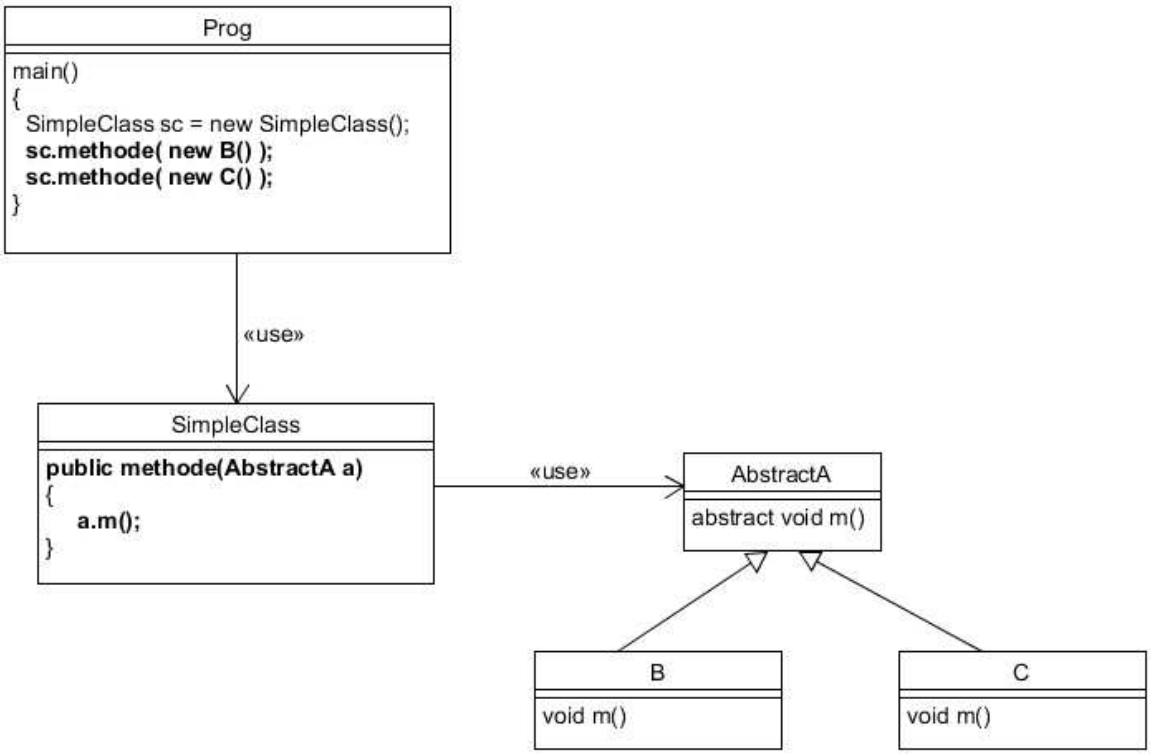
Définition: Un traitement générique est un traitement informatique qui prend (aussi) en entrée des traitements.

En terme objet cela se traduit en : "une méthode qui prend en entrée des méthodes".

Comment **décrire** le passage en paramètre de ces méthodes ?
Réponse : la classe abstraite.

Comment **passer** en paramètre des méthodes réelles?
Réponse : passer en paramètre un objet dont la classe hérite de la classe abstraite, et qui a donc implémenté les méthodes.

On peut représenter ce principe avec le diagramme suivant :



La méthode ici

```
methode(AbstractA a)
```

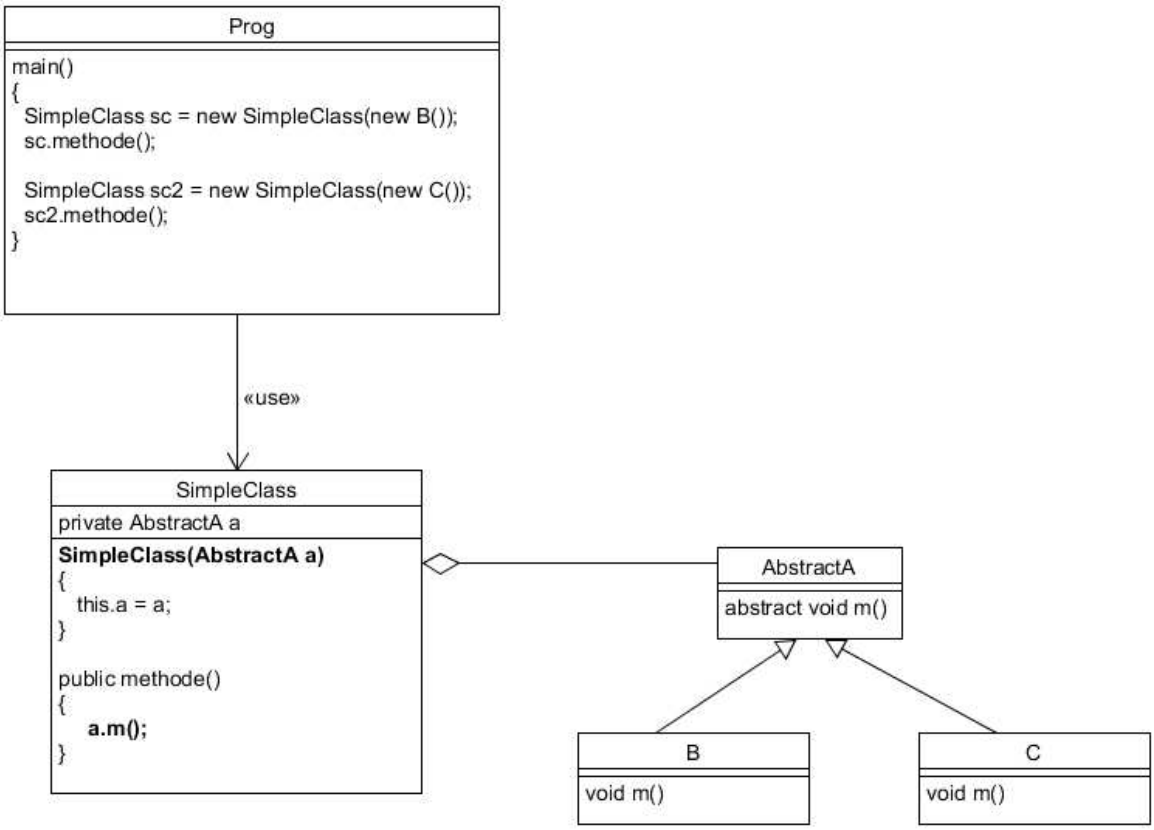
prend en entrée une classe abstraite.

On utilise la méthode :

```
sc.methode(new B());
```

en passant en paramètre de A, une instance de la classe B (voir plus loin "la compatibilité des classes').

On peut avoir aussi cette représentation :



6.2. Exemple 14 : la bibliotheque

On reprend l'exemple 13 en transformant la classe MultiMedia en classe abstraite.

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple14_Bibliotheque**

Dans cet exemple les méthodes de la classe Bibliotheque suivantes sont génériques :
getTousLesMedias
getCatalogue
getMediaEmpruntDepasse

Commentaires en cours.

6.3. Exercice 17 : la classe Musique

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exercice17_Musique**

6.4. La compatibilité des classes

Soit la clase B qui hérite de A.
Soit la méthode suivante :

```
public void m(A a)
{……..}
```

Alors, on peut écrire :

```
B b = new B();
m(b)
```

Explication :
Par contrôle du compilateur, la méthode m ne peut utiliser sur a que des méthodes de la classe A.
Donc, si je passe en paramètre un objet b, cet objet a hérité des méthodes de A, il peut donc être utilisé dans la méthode m.

A l'inverse :

Soit la méthode

```
m(B b)
```

Si je veux passer en paramètre un objet de type A alors il faut que l'on "force" le passage en paramètre en ajoutant une instruction de "**cast**" :

Exemple :

```
public class Collection
{
    ArrayList<A> elements;

    public Collection() {
        elements = new ArrayList<A>();  }

    public void ajouter(A a)    {
       elements.add(a);   }

    public A get(int i){elements.get(i);}
}
```

```
B b = new B();
Collection tab = new Collection();
tab.ajouter(b);  // Collection polymorphe

A a = tab.get(0);

et soit une methode qcq void m(B b)

m( (B) a );   //  Cast
```

```
```

Attention si tab.get(0) n'était pas un objet de type B alors erreur d'exécution (exception).

D'où l'utilisation de l'instruction **instance of** :

```
if (a instance of B)    m( (B) a);
```

Exemple de la méthode getLivres() qui retourne tous les livres de la bibliothèque

6.5. La methode getClass()

La classe Object contient la method : getClass().
Cette méthode retour la classe, un objet de type **Class**, qui est la classe d'instanciation de l'objet.

Dans la gestion des collections polymorphes, il est souvent courant de vouloir tester l'appartenance de l'objet en fonction du nom (String) de la classe.

Pour tout objet, il est alors possible de faire le test suivant :

```
if (o.getClass().getName().equals(nomDeLaClasse))
   ….
```

où
nomDeLaClasse est une String qui contient le nom de la classe à tester.
Attention, en Java, ce nom est fonction du package dans lequel la classe est accessible.
Exemple : "fr.cnam.projet.Livre"

Voir l'exemple 14 dans lequel la méthode getCatalogue(String typeMedia) de la Bibliotheque utilise cela pour parcourir la collection polymorphe et ne retient que les objets qui sont d'un type donné.

7. Initialisation d'une collection polymorphe

Une collection polymorphe étant composée, par définition, de types différents, comment faire le **new** de la bonne classe quand les informations proviennent d'un fichier ou une base de données ?

Peut-on rester générique ? La réponse est non (sans utiliser la réflexivité du langage Java).

Pour créer un objet dans une collection polymorphe, il faut savoir avant de créer l'objet, de quel type il est. Mais savoir de quel type il est, c'est avoir l'objet.
Pour contourner ce paradoxe, il faut faire deux choses :

- dans le fichier, mettre devant les informations de l'objet, une information qui précise le type de l'objet.
- écrire, explicitement (pas de généricité), le code qui teste cette information et réalise en fonction le bon new. Cela implique que quand on crée une nouvelle classe dérivée, il faut compléter ce code (non générique).

Comme nous le verrons dans le cours sur les entrées/sorties, il existe le principe de Serialization qui est une "solution" pour lire une collection polymorphe de manière générique car le mécanisme de sérialisation utilise la réflexivité du langage Java pour créer lui-même les objets.

Extrait de l'exemple 14 qui crée explicitement les objets de la collection polymorphe en fonction du type de l'objet écrit en en-tête de chaque ligne des informations de l'objet.

```
    private void initialiser()
    {
        String[] lignes = Terminal.lireFichierTexte(
                                    "../data/Biblio.txt");

        for(String ligne : lignes)
            {
                String[] type = ligne.split("[;]");

                if (type[0].equals("LIVRE"))
                    {
                        Livre l = new Livre();
                        l.decoder(ligne);
                        ajouter(l);
                    }

                if (type[0].equals("FILM"))
                    {
                        Film f = new Film();
                        f.decoder(ligne);
                        ajouter(f);
                    }

                if (type[0].equals("JEU"))
                    {
                        Jeu j = new Jeu();
                        j.decoder(ligne);
                        ajouter(j);
                    }
            }
    }
```

Dans une bonne programmation objet, le décodage de la ligne se fait dans la classe d'appartenance de chacun des types d'objet.

Il faut alors nécessaire de créer des constructeurs par défaut (vide) pour toutes les classes.

8. L'interface

8.1. Définition

L'interface est une "classe abstraite" sans attributs et dont toutes les méthodes sont abstraites (sauf une seule depuis Java 1.8).

Pourquoi alors utiliser les interfaces ?

Le langage JAVA ne permet pas l'héritage multiple[1] mais une classe peut "hériter" de plusieurs interfaces.
Ainsi pour ajouter de nouvelles propriétés à une classe qui hérite déjà d'une classe, la seule solution est que cette classe "hérite" d'une interface.

Pour une interface, on ne parle pas d' "héritage" mais d'**implémentation**.

On dit qu'une classe implémente une interface.

En JAVA cela s'écrit :

```
public class Exemple implements UneInterface
{
….
}
```

ou

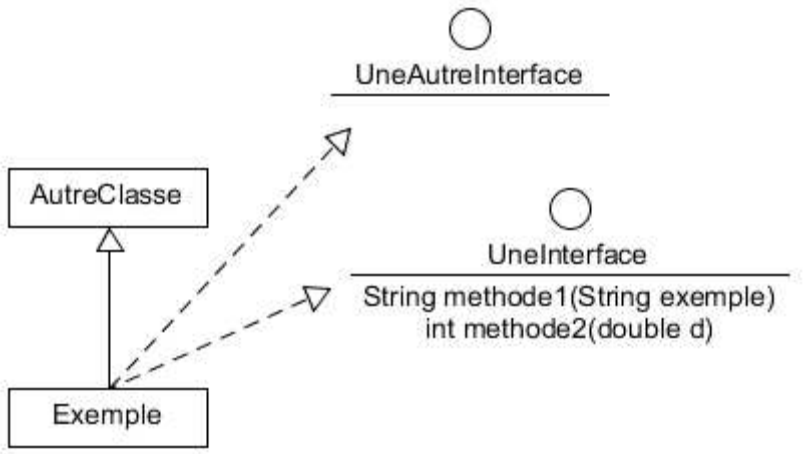
```
public class Exemple extends AutreClasse implements UneInterface,
UneAutreInterface
{
….
}
```

La déclaration de l'interface s'écrit :

```
public interface UneInterface
{
    public String methode1(String exemple);
    public int methode2(double d);
}
```

En notation UML cela s'écrit :

[1] Dans les langages objet, comme C++, qui ne possèdent pas les Interfaces, retenez que vous pourrez donc créer l'équivalent en créant une classe abstraite sans paramètres et comme ils permettent l'héritage multiple ils ont l'équivalent de l'interface de Java.



8.2. Utilisation d'une interface

Tout ce que l'on a précisé sur les classes abstraites s'applique donc aussi aux interfaces :

- une classe qui implémente une interface doit implémenter toutes les méthodes de l'interface
- une interface permet de créer une collection polymorphe
- une interface permet de créer un traitement générique.

Le premier exemple que l'on a vu de l'usage d'une interface est la classe Formulaire avec son interface FormulaireInt.

Nous pouvons maintenant décortiquer son code :

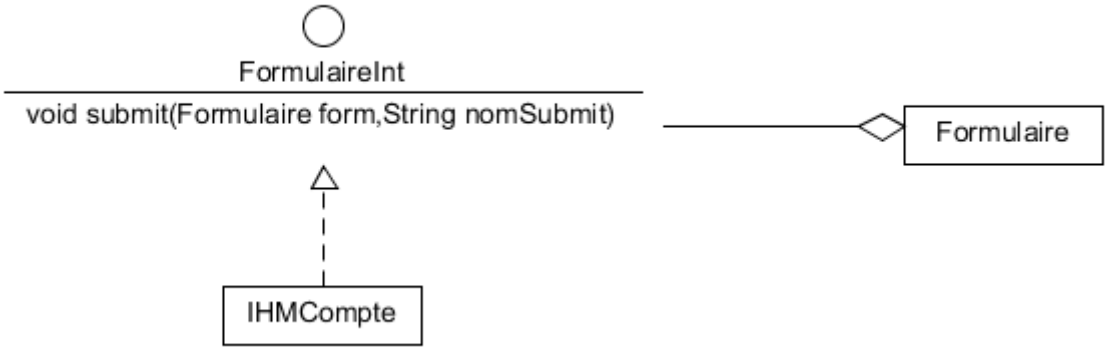
```
public class Formulaire
{
    private FormulaireInt    app;

    public Formulaire(String titre,
                      FormulaireInt app,
                      boolean synchrone,
                      int width,
                      int height,
                      boolean avecFrame)
    {
        initFormulaire(titre,app,synchrone,width,height,avecFrame);
    }

    public void initFormulaire(String titre,
                               FormulaireInt app,
                               boolean synchrone,
                               int width,
                               int height,
                               boolean avecFrame)
    {
        this.app             = app;
        […]
    }
```

```
[…]
      Button b;

      b.addActionListener(new SubmitListener(this,nom));

[…]



    // Classe d'action des boutons du formulaire
    class SubmitListener implements ActionListener
    {
        private Formulaire form;
        private String     nomSubmit;

        public SubmitListener(Formulaire form, String nomSubmit)
        {
            this.form      = form;
            this.nomSubmit = nomSubmit;
        }

        public void actionPerformed(ActionEvent e)
        {
            try{
                if (app!=null
                    app.submit(form,nomSubmit);
              […]
        }
    }

}
```

```
public interface FormulaireInt
{
    public void  submit(Formulaire form,String nom);
}
```

```
public class IHMCompte implements FormulaireInt
{
    private Compte     compte;  // Le compte
    private Formulaire form;    // Le formulaire

    // Constructeur
    //
    public IHMCompte(Compte compte)
    {
        this.compte = compte;

        // Creation du formulaire
        form = new Formulaire("Exemple",this,880,530);

  […]

    public void submit(Formulaire form,String nomSubmit)
    {

        // Affichage de tous les rendez-vous de l'agenda
        //
```

```
          if (nomSubmit.equals("AFF_TOUT_COMPTE"))
              {
                  String res = compte.listerToutesOperations();
                  form.setValeurChamp("RESULTATS",res);
              }
    […]

     }

 }
```



Le principe de base de l'utilisation d'une interface est donc le suivant :

Tout constructeur ou méthode qui prend en paramètre une interface accepte tout objet dont la classe d'appartenance implémente cette interface.

8.3. Propriétés d'une interface

En résumant et en complétant ce que l'on a vu jusqu'à maintenant :

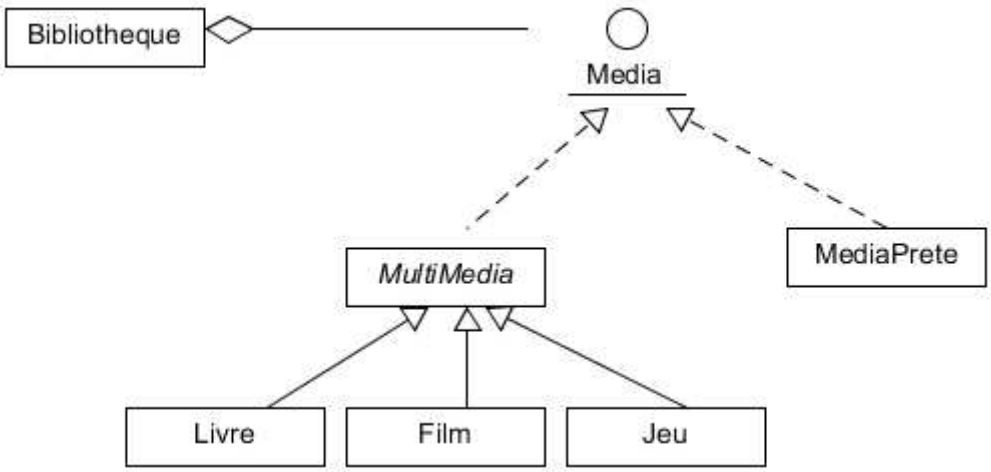
- Une classe peut implémenter plusieurs interfaces. Dans ce cas, la classe doit implémenter toutes les méthodes de toutes les interfaces
- Une interface peut hériter d'une interface. Dans ce cas, la classe doit implémenter toutes les méthodes de toutes les interfaces héritées
- Une interface peut contenir des attributs static.

8.4. Exemple 15 : Bibliotheque avec une interface

Nous continuons l'exemple précédent de la bibliothèque dans lequel :

- la bibliothèque gère une collection dont les éléments sont de type d'une interface **Media** (au lieu de la classe abstraite dans l'exemple précédent)
- on peut donc maintenant ajouté dans la collection des objets qui n'appartiennent pas à l'arborescence d'héritage de la classe abstraite **MultiMedia** .

On obtient la description synthétique suivante :



L'interface créée est la suivante :

```
public interface Media
{
    public Calendar getDateEmprunt();
    public void     setDateEmprunt(Calendar d);
    public int      getDureeEmprunt();
    public String   formatCourt();
    public void     decoder(String ligne);
}
```

On doit mettre dans cette interface toutes les méthodes des medias qui sont utilisées dans la classe Bibliotheque.

Commentaires en cours :

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple15_BibliothequeInterface**

9. Conclusion

Nous voyons que l'héritage et l'interface sont au cœur de l'architecture des classes d'une programmation orientée objet.
Ces concepts permettent une meilleure écriture du code et offrent une meilleure évolutivité du code.

Ainsi, comme nous le verrons plus loin, ces concepts sont très utilisés dans les classes prédéfinies du langage Java. Elles sont donc très importantes pour comprendre comment utiliser ces classes, et notamment pour les classes qui gèrent les collections :

- trier des collections polymorphes grâce aux interfaces **Comparable**, **Comparator**
- itérer des ensembles d'éléments grâce aux interfaces **Iterable** et **Iterator**
- implémenter de nouvelles collections grâces aux classes abstraites : **AbstractCollection**, **AbstractList**, **AbstractSet**, **AbstractMap**, …et grâce aux interfaces **Collection, List, Set, Map** …
- créer un mécanisme de notification à couplage faible : **Observer**
- … etc …

10. Exercice : Interface des vivants

Exercice 07 et Exercice 08 du site. Lisez les énoncés.

Commentaires en cours