



Introduction à l'Orienté Objet :

Présentation du paradigme Objet

Intérêt de l'orienté objet

Etapas création d'applications OObjet

Notions Fondamentales de la POO :

Les Classes en POO

Notion de Visibilité/accessibilité POO

Méthodes spéciales en POO

L'association des classes en POO

Notions Avancées de la POO :

L'héritage en POO

Le Polymorphisme en POO

Classes et méthodes statiques en POO

Classes et méthodes abstraites en POO

Les interfaces en POO

Les classes internes en POO

Auteur: Sarra KOUIDER

Conception et Programmation Orientées Objet

Programmation orienté Objet (POO) (Cours 3)



Introduction à l'Orienté Objet :

Présentation du paradigme Objet

Intérêt de l'orienté objet

Etapes création d'applications OObjet

Notions Fondamentales de la POO :

Les Classes en POO

Notion de Visibilité/accessibilité POO

Méthodes spéciales en POO

L'association des classes en POO

Notions Avancées de la POO :

L'héritage en POO

Le Polymorphisme en POO

Classes et méthodes statiques en POO

Classes et méthodes abstraites en POO

Les interfaces en POO

Les classes internes en POO

Auteur: Sarra KOUIDER

Conception et Programmation Orientées Objet

Introduction à l'orienté Objet

Que représente un objet dans le monde réel ?

L'Objet dans le monde réel

Un **Objet** est une toute chose **concrète perceptible** à l'être humain par la vue et le toucher.

Autrement défini : un **Objet** est toute chose définie par son **utilisation**, sa **valeur**, et ses **attributs**.

Un objet voiture



uneVoiture

Attributs :

couleur = bleue
poids = 979 kg
puissance = 12 CV
capacité carburant = 50 l
conducteur = Dupont
vitesse instantanée = 50 km/h

Opérations :

démarrer
déplacer
mettreEssence

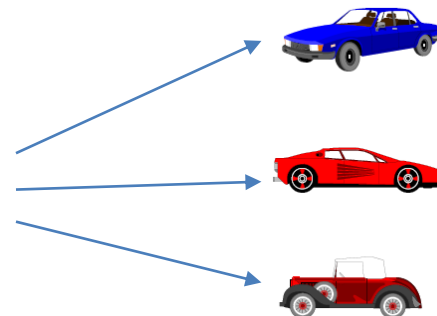
Le paradigme Object dans le monde numérique

L'Orientée Objet

- ❖ La notion d'orientée objet à pour concepts clés: **les Classes** et **les Objets**.
- ❖ La notion de **classe** représente le **type** de l'objet, alors que le mot **objet** est **l'instance** physiquement réalisée de la classe, qui n'a d'existence que pendant l'exécution du programme.
- ❖ Un **Programme Orienté Objet** est un **ensemble de classes** qui collaborent entre elles. Ces classes s'échangent des messages entre elles pour réaliser certaines tâches.

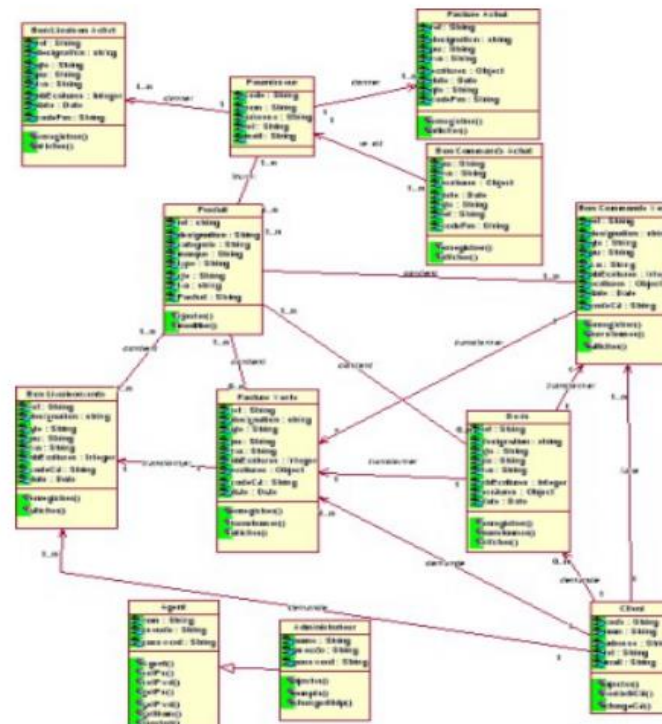
Voiture
- _marque: String - _couleur: String - _longueur: int - _immatriculation: String
+ Voiture(marque:String):void + get marque():String + get couleur():String + get longueur():int + get immatriculation():String + set couleur(nouvelleCouleur:String):void + set longueur(nouvelleLongueur:int):void + set immatriculation(nouvelleImmatriculation:String):void

Classe



Instances

Le paradigme Objet dans le monde numérique



exemple d'un POO représenté en UML

Pourquoi parler de la conception par objets ?

Succès des approches par objets (30 dernières années) :

- Décrire un système avec des représentations informatiques proches des entités du problème et de sa solution
- Avantages reconnus en termes de :
 - facilité du codage initial,
 - stabilité du logiciel construit car les objets manipulés sont plus stables que les fonctionnalités attendues,
 - aisance à réutiliser les artefacts existants et ...
 - à maintenir le logiciel, le corriger, le faire évoluer ;
- Fort développement dans les langages de conception, de programmation, les bases de données, les interfaces graphiques, les systèmes d'exploitation, etc.

Exemple de langages Orienté Objet : C++, .net, Java...

Caractéristiques majeures d'un modèle orienté objet

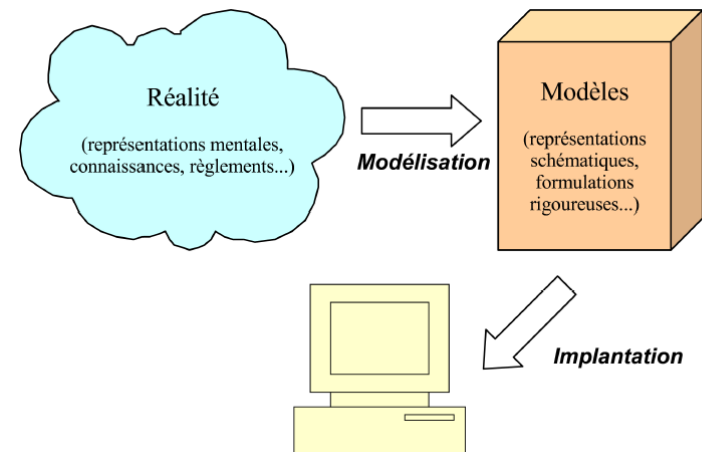
- **abstraction :**
Ressortir les caractéristiques externes essentielles d'une entité pour la distinguer des autres.
- **encapsulation :**
Cacher les détails qui ne font pas partie des caractéristiques essentielles d'une entité.
- **modularité :**
Décomposer un programme en un ensemble de modules cohérents et faiblement couplés pouvant être compilés séparément.
- **hiérarchisation :**
ranger ou ordonnancer les abstractions

Attention

si un modèle ne possède pas l'un de ces éléments, il n'est pas orienté objet.

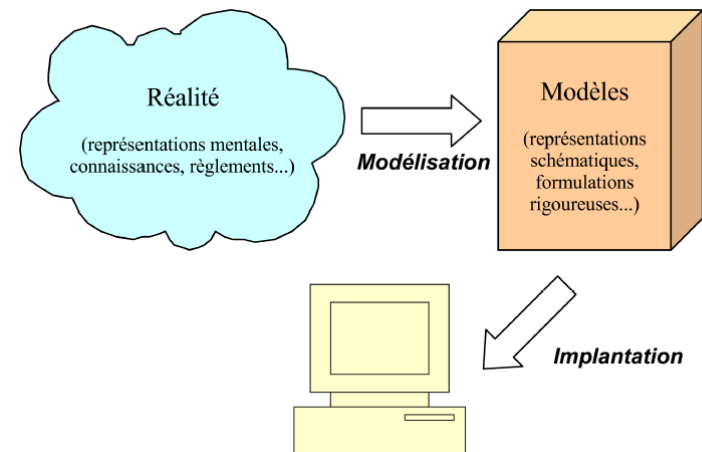
Etapas de création d'une application orientée objet

1. Utilisation d'un langage de modélisation pour la conception (UML)
2. Utilisation d'un langage de programmation pour l'implémentation (C#, Java,..)



Etapas de création d'une application orientée objet

1. Utilisation d'un langage de modélisation pour la conception (UML)
2. **Utilisation d'un langage de programmation pour l'implémentation (C#, Java,..)**





Introduction à l'Orienté Objet :

Présentation du paradigme Objet

Intérêt de l'orienté objet

Etapes création d'applications OObjet

Notions Fondamentales de la POO :

Les Classes en POO

Notion de Visibilité/accessibilité POO

Méthodes spéciales en POO

L'association des classes en POO

Notions Avancées de la POO :

L'héritage en POO

Le Polymorphisme en POO

Classes et méthodes statiques en POO

Classes et méthodes abstraites en POO

Les interfaces en POO

Les classes internes en POO

Auteur: Sarra KOUIDER

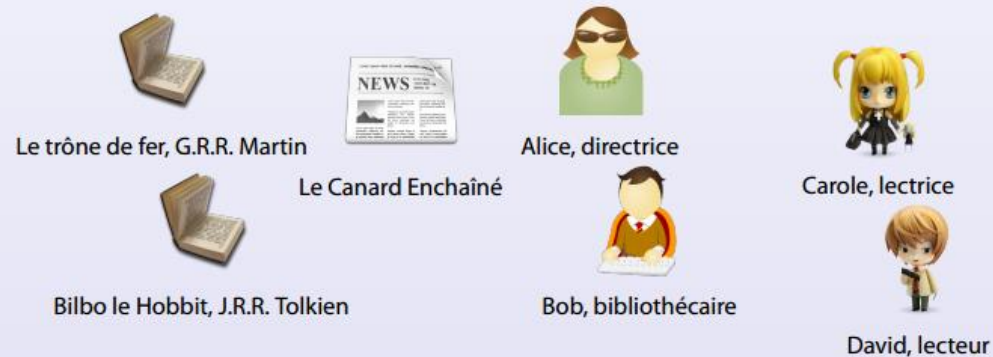
Conception et Programmation Orientées Objet

Notions Fondamentales de la POO

La notion de « Classe » en POO

- Approche procédurale : « *Que doit faire mon programme ?* »
- Approche orientée-objet : « *De quoi doit être composé mon programme ?* »

Cette composition est conséquence d'un choix de modélisation fait pendant la conception (comme en CSI)



La notion de « Classe » en POO

Des objets similaires peuvent être informatiquement décrits par une même abstraction : une **classe**

- même structure de données et méthodes de traitement
- valeurs différentes pour chaque objet

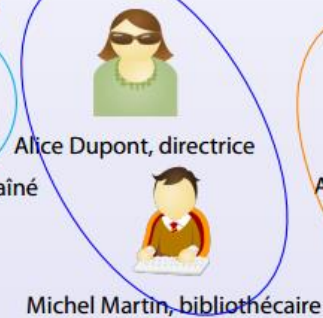
Classe Livre



Classe Journal



Classe Employé



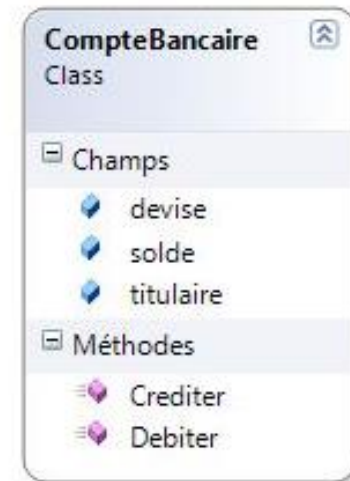
Classe Lecteur



La notion de « Classe » en POO

Une classe est composée de plusieurs **membres** dont chacun est soit

- ▶ un **attribut** : variable typée
- ▶ une **méthode** (ou opération) : ensemble d'instructions de traitement



Classe UML « CompteBancaire »

Déclaration d'une classe en C# et Java

Syntaxe

```
class <nom de la classe> {
    <contenu de la classe>
}
```

Code en C#	Code en Java
<pre>using System; namespace MaPremiereApplication { class Program { static void Main(string[] args) { Console.WriteLine("Hello World !!"); } } }</pre>	<pre>import java.io.*; Package MaPremiereApplication; public class Principale { public static void main(String[] args) { System.out.println("Hello world !!"); } }</pre>

Déclaration d'une classe en C# et Java

exemple : déclaration d'une classe « CompteBancaire »

Code en C#	Code en Java
<pre> public class CompteBancaire { public string titulaire; public double solde; public string devise; public void Crediter(double montant) { solde = solde + montant; } public void Debiter(double montant) { solde = solde - montant; } } </pre>	<pre> public class CompteBancaire { public string titulaire; public double solde; public string devise; public void Crediter(double montant) { solde = solde + montant; } public void Debiter(double montant) { solde = solde - montant; } } </pre>

même syntaxe

Instanciation et utilisation d'une classe en C# et Java

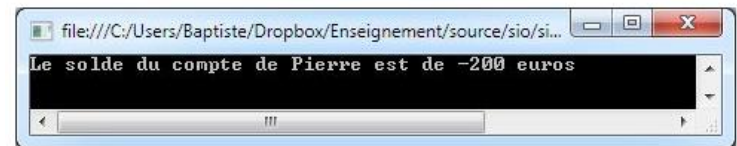
exemple : manipulation de la classe « CompteBancaire » (code en C#)

```
static void Main(string[] args)
{
    CompteBancaire comptePierre; // déclaration d'un nouvel objet
    comptePierre = new CompteBancaire(); // instanciation de cet objet

    // affectations de valeurs aux attributs
    comptePierre.titulaire = "Pierre";
    comptePierre.solde = 0;
    comptePierre.devise = "euros";

    // appels de méthodes
    comptePierre.Crediter(300);
    comptePierre.Debiter(500);
    string description = "Le solde du " + comptePierre.titulaire + " est de " + comptePierre.solde + " " +
        comptePierre.devise;

    Console.WriteLine(description);
}
```



Instanciation et utilisation d'une classe en C# et Java

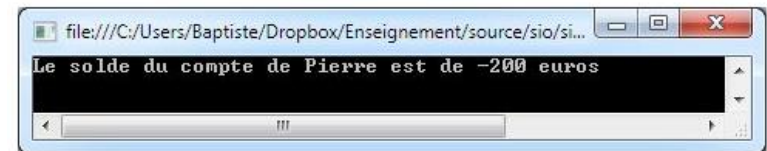
exemple : manipulation de la classe « CompteBancaire » (code en Java)

```
static void Main(string[] args)
{
    CompteBancaire comptePierre; // déclaration d'un nouvel objet
    comptePierre = new CompteBancaire(); // instanciation de cet objet

    // affectations de valeurs aux attributs
    comptePierre.titulaire = "Pierre";
    comptePierre.solde = 0;
    comptePierre.devise = "euros";

    // appels de méthodes
    comptePierre.Crediter(300);
    comptePierre.Debiter(500);
    string description = "Le solde du " + comptePierre.titulaire + " est de " + comptePierre.solde + " " +
        comptePierre.devise;

    System.out.println (description);
}
```



Notion de visibilité/accessibilité en C# et Java

- un membre privé (**private**) n'est visible que dans les instances directes de la classe où il est déclaré.
- un membre **sans modifieur** est visible uniquement dans les instances directes de la classe où il est déclaré et dans celles des classes du même paquetage.
- un membre protégé (**protected**) n'est visible que dans les instances, directes ou non, de la classe où il est déclaré (et donc aussi dans les instances des sous-classes) et dans les instances des classes du même paquetage.
- un membre public (**public**) est visible par n'importe quel objet.

Notion de visibilité/accessibilité en C# et Java

modifieur	classe	paquetage	sous-classes	autres classes
<code>private</code>	visible			
	visible	visible		
<code>protected</code>	visible	visible	visible	
<code>public</code>	visible	visible	visible	visible

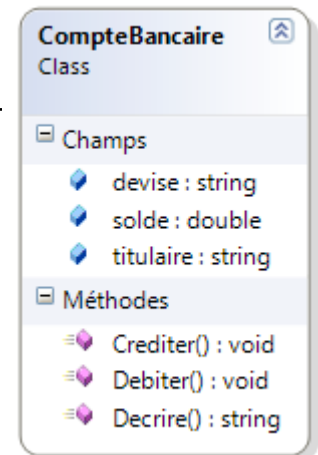
Visibilité des membres selon leurs modifieurs et le niveau d'encapsulation

Méthodes Spéciales : Constructeur

Déclaration d'une méthode constructeur (code en C# et Java)

```
public class CompteBancaire
{
    public string titulaire;
    public double solde;
    public string devise;

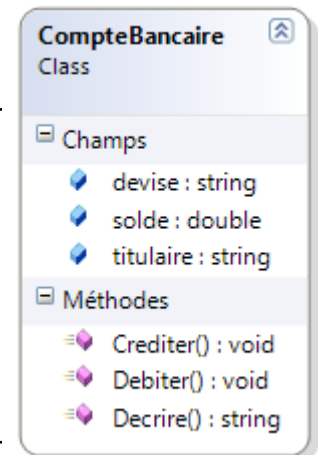
    // Le constructeur de classe
    public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)
    {
        this.titulaire = leTitulaire;
        this.solde = soldeInitial;
        devise = laDevise;
    }
}
```



Méthodes Spéciales : Constructeur

Déclaration d'une méthode constructeur (code en C# et Java)

```
// Le constructeur de classe
public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)
{
    this.titulaire = leTitulaire;
    this.solde = soldeInitial;
    devise = laDevise;
}
```



Attention

- ❖ Le nom du constructeur doit être **identique** au **nom de la classe**, et sa définition **ne comporte pas** le void
- ❖ Un constructeur par défaut (vide) est implicitement créé (pas besoin de l'écrire).
- ❖ Une classe peut disposer de plusieurs constructeurs initialisant différents attributs.

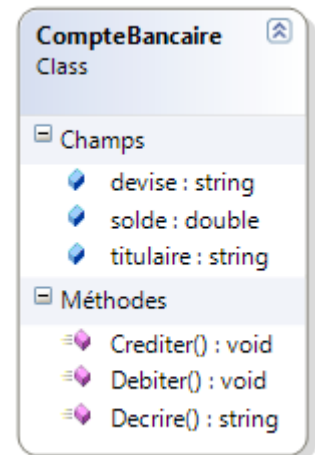
Méthodes Spéciales : Constructeur

Utilisation d'une méthode constructeur (code en C# et Java)

```
// déclaration et instanciation d'un nouvel objet en utilisant son constructeur
CompteBancaire comptePierre = new CompteBancaire("Pierre", 0, "euros");

// appels de méthodes
comptePierre.Crediter(300);
comptePierre.Debiter(500);
Console.WriteLine(comptePierre.Decrire()); //affichage en C#


/* Pour l'affichage en Java il faut utiliser la commande System.out.println (); */
```



Méthodes Spéciales : Destructeur

Déclaration d'une méthode destructeur (code en C#)

```
class CompteBancaire
{
    // destructor
    ~CompteBancaire()
    {
        // cleanup statements...
        Console.WriteLine("l'objet CompteBnacaire est nettoyé de la mémoire");
    }
}
```

CompteBancaire


Class

Champs

- devise : string
- solde : double
- titulaire : string

Méthodes

- Crediter() : void
- Debiter() : void
- Decrire() : string

Méthodes Spéciales : Destructeur

Déclaration d'une méthode destructeur (code en C#)

Attention

En C# , le destructeur appelle implicitement **Finalize()** sur la classe de base de l'objet.

```
protected override void Finalize()  
{  
    try  
    {  
        // Cleanup statements...  
    }  
    finally  
    {  
        base.Finalize();  
    }  
}
```


Méthodes Spéciales : Destructeur

Déclaration d'une méthode destructeur (code en Java)

```
class CompteBancaire
{
    // destructor
    public void finalize()
    {
        System.out.println("Objet CompteBancaire nettoyé de la mémoire");
    }
}
```

CompteBancaire
Class

Champs

- devise : string
- solde : double
- titulaire : string

Méthodes

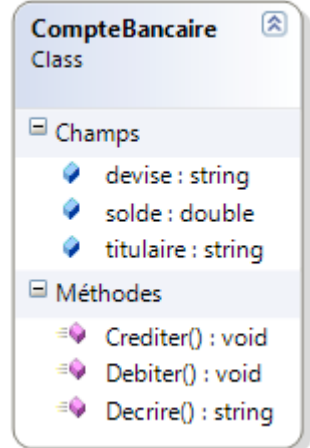
- Crediter() : void
- Debiter() : void
- Decrire() : string

Méthodes Spéciales : Destructeur

Utilisation d'une méthode destructeur (code en C# et Java)

Class Program

```
{
    static void Main(string[] args)
    {
        // déclaration et instanciation d'un nouvel objet en utilisant son constructeur
        CompteBancaire comptePierre = new CompteBancaire("Pierre", 0, "euros");
        Console.WriteLine(comptePierre.Decrire()); //affichage en C#
    }
}
```



Output :

Le compte de Pierre est a 0 euro
Pierre CompteBancaire nettoyé de la mémoire.

Méthodes Spéciales : Accesseurs

Les Accesseurs en POO

- ❖ L'**encapsulation** des **attributs** en POO est effectuée grâce à aux méthodes dites **Accesseurs** (**get()** et **set()**).
- ❖ L'**encapsulation** diminue **les risques** de toute mauvaise **manipulation** (accidentelle ou volontaire) des données internes.
- ❖ Possibilité de **modifier les détails internes** d'une classe **sans changer** son **comportement extérieur**.

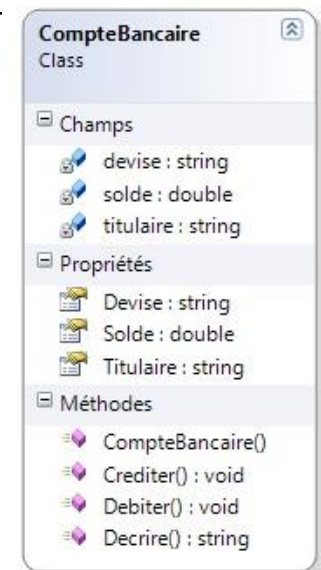
Méthodes Spéciales : Accesseurs

Ajout d'accesseurs en POO (code en C#)

```
public class CompteBancaire
{
    private string titulaire;
    private double solde;
    public string devise { get; set; }

    public string Titulaire
    {
        get { return titulaire; }
        set { titulaire = value; }
    }

    public double Solde
    {
        get { return solde; }
        set { solde = value; }
    }
}
```



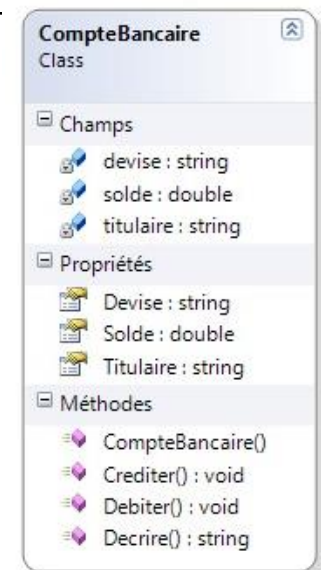
Méthodes Spéciales : Accesseurs

Attributs en lecture ou écriture seule (code en C#)

```
public class CompteBancaire
{
    private string titulaire;
    private double solde;
    public string devise { get; };

    public string Titulaire
    {
        get { return titulaire; }
    }

    public double Solde {
        set { solde = value; }
    }
}
```



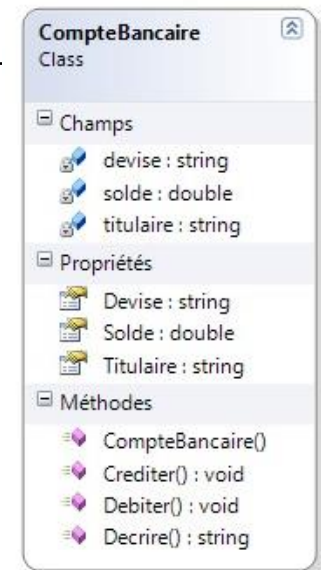
Méthodes Spéciales : Accesseurs

Ajout d'accesseurs en POO (code en Java)

```
public class CompteBancaire {
    private string titulaire;

    public string getTitulaire() {
        return titulaire;
    }

    public string setTitulaire (string value) {
        solde = value;
    }
}
```



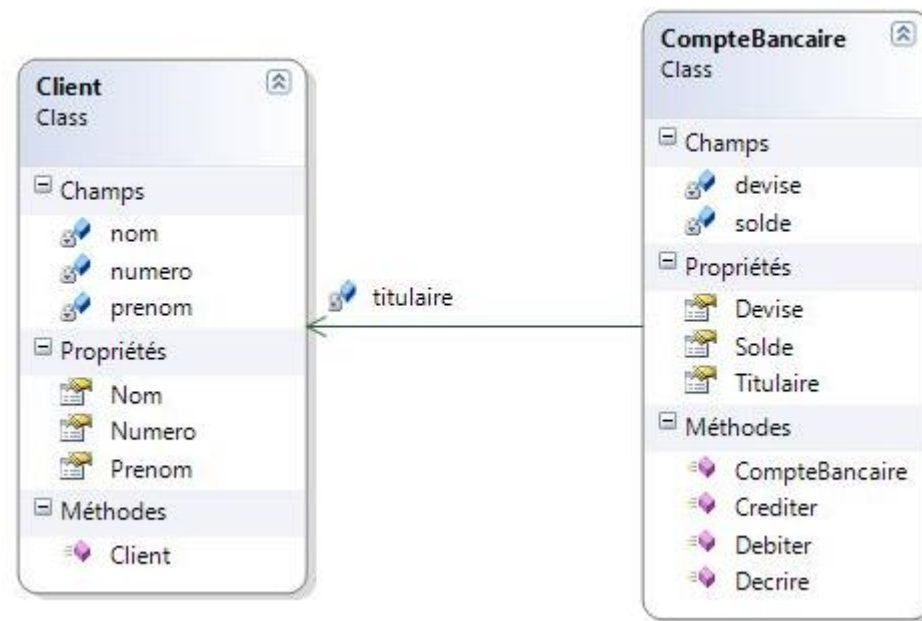
Association entre classes en POO

Le besoin de relation en POO

- ❖ La POO consiste à concevoir **une application** sous la forme de **briques logicielles** appelées des **objets**.
- ❖ Chaque **objet** joue un **rôle** précis et peut **communiquer avec les autres objets**.
- ❖ Les **interactions** entre les différents **objets** vont permettre à l'application de **réaliser les fonctionnalités** attendues.

Association entre classes en POO : **Association Simple**

Exemple d'une association de type simple



Association entre classes en POO : **Association Simple**

Association de type simple (code en C#)

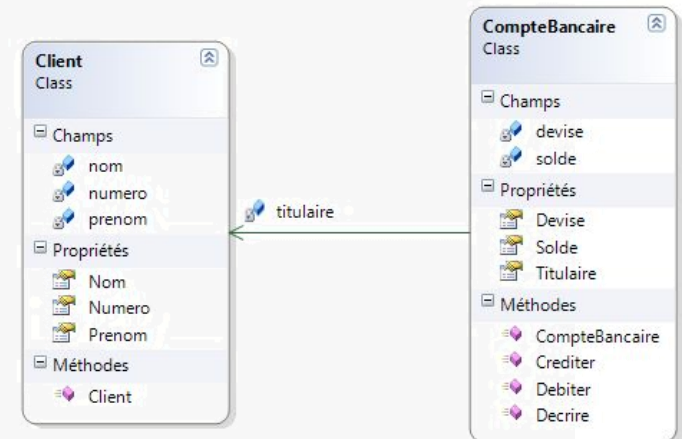
```
class CompteBancaire
{
    private Client titulaire; // type string => type Client

    // ...

    public CompteBancaire(Client leTitulaire, double soldeInitial, string laDevise)
    {
        titulaire = leTitulaire;
        solde = soldeInitial;
        devise = laDevise;
    }

    public Client Titulaire
    {
        get { return titulaire; }
    }

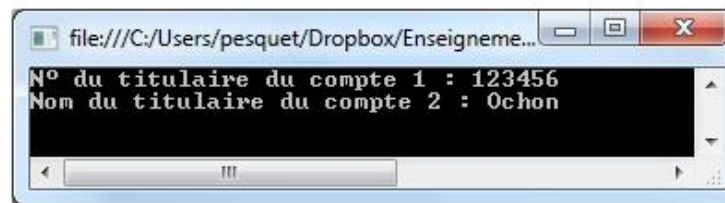
    // ...
}
```



Association entre classes en POO : **Association Simple**

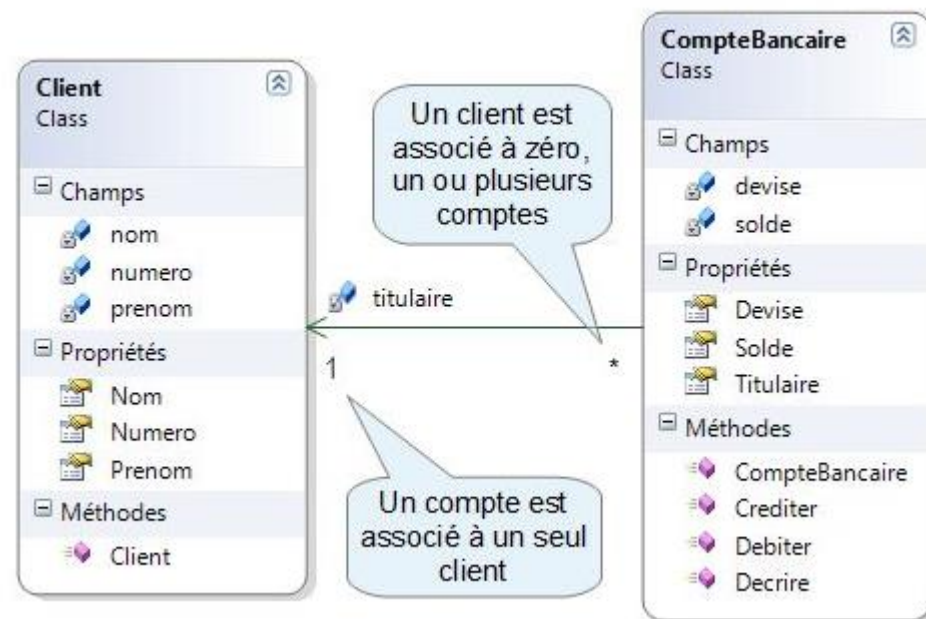
Association de type simple (code en C#)

```
Client pierre = new Client(123456, "Kiroul", "Pierre");  
Client paul = new Client(987654, "Ochon", "Paul");  
  
CompteBancaire compte1 = new CompteBancaire(pierre, 500, "euros");  
CompteBancaire compte2 = new CompteBancaire(paul, 1000, "euros");  
  
Console.WriteLine("N° du titulaire du compte 1 : " + compte1.Titulaire.Numero);  
Console.WriteLine("Nom du titulaire du compte 2 : " + compte2.Titulaire.Nom);
```



Association entre classes en POO : **Association Multiple**

Exemple d'une association de type Multiple



Association entre classes en POO : **Association Multiple**

Notion de Multiplicité en POO

Multiplicité	Signification
0..1	Zéro ou un
1	Un
*	De zéro à plusieurs
1..*	De un à plusieurs

Association entre classes en POO : **Association Multiple**

Association de type Multiple (code en C#)

```
public class Client
{
    // ...
    private List<CompteBancaire> comptes;

    public Client(int numero, string nom, string prenom)
    {
        comptes = new List<CompteBancaire>();
        // ...
    }

    public List<CompteBancaire> Comptes
    {
        get { return comptes; }
    }

    // ...
}
```



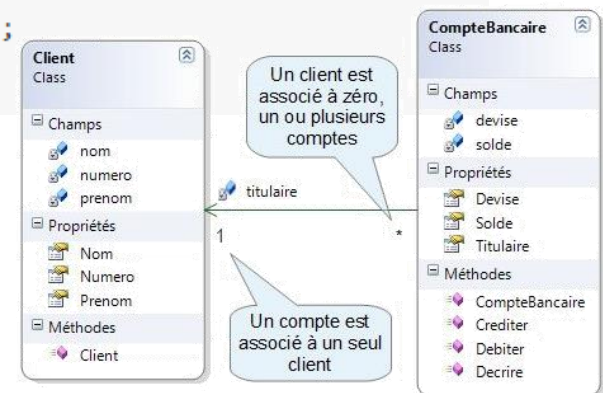
Association entre classes en POO : **Association Multiple**

Association de type Multiple (code en C#)

```
Client pierre = new Client(123456, "Kiroul", "Pierre");
Client paul = new Client(987654, "Ochon", "Paul");

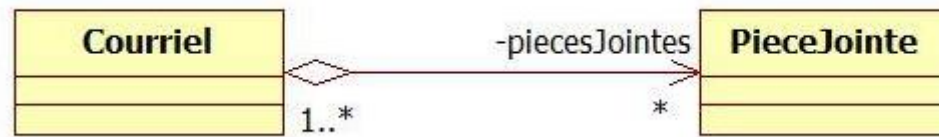
// association entre pierre et compte1
CompteBancaire compte1 = new CompteBancaire(pierre, 500, "euros");
pierre.Comptes.Add(compte1);

// association entre paul et compte2
CompteBancaire compte2 = new CompteBancaire(paul, 1000, "euros");
paul.Comptes.Add(compte2);
```



Association entre classes en POO : **Agrégation**

Association de type Agrégation (code en C# et java)



une **agrégation** est une association qui modélise une relation "se compose de".

La traduction en code source d'une agrégation est identique à celle d'une association

```

public class Courriel {
    private List<PieceJointe> piecesJointes;
    //...
}
  
```

Association entre classes en POO : **Composition**

Association de type Composition (code en C# et java)



- La traduction en code source d'une composition ressemble à celle d'une agrégation.
- Les composants sont parfois instanciés par le constructeur du composé.

```
class Livre
{
    private List<Page> pages;
    // ...
}
```




Introduction à l'Orienté Objet :

Présentation du paradigme Objet

Intérêt de l'orienté objet

Etapes création d'applications OObjet

Notions Fondamentales de la POO :

Les Classes en POO

Notion de Visibilité/accessibilité POO

Méthodes spéciales en POO

L'association des classes en POO

Notions Avancées de la POO :

L'héritage en POO

Le Polymorphisme en POO

Classes et méthodes statiques en POO

Classes et méthodes abstraites en POO

Les interfaces en POO

Les classes internes en POO

Auteur: Sarra KOUIDER

Conception et Programmation Orientées Objet

Notions Avancée de la POO

L'Héritage en POO

Problématique

- Une application a besoin de services dont une partie seulement est proposée par une autre classe déjà définie (classe dont on ne possède pas nécessairement le source),
- Ne pas réécrire le code

Un Point

- a une position
- peut être déplacé
- peut calculer sa distance à l'origine
- ...

Application a besoin

- de manipuler des points (comme le permet la classe `Point`)
- mais en plus de les dessiner sur l'écran.

`PointGraphique` = `Point`

- + une couleur
- + une opération d'affichage

L'Héritage en POO

Solution en POO : l'Héritage

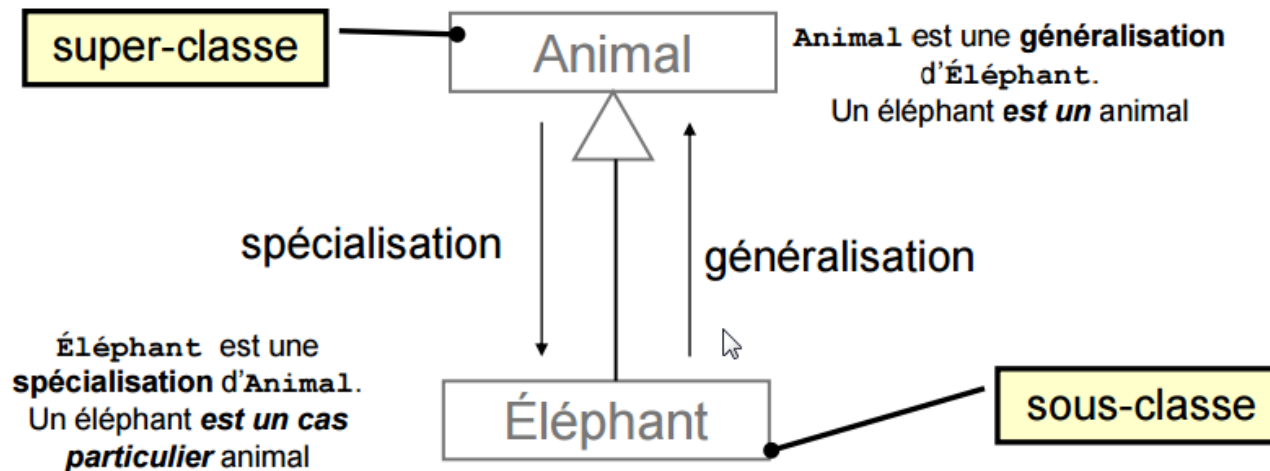
Héritage (inheritance) : Définir une nouvelle classe à partir de la classe déjà existante.

Définition

- L'**Héritage** permet de reprendre les caractéristiques d'une classe **M** existante pour les étendre et définir ainsi une nouvelle classe **F** qui hérite de **M**.
- Les objets de **F** possèdent toutes les caractéristiques de **M** avec en plus celles définies dans **F**.
- La relation d'héritage peut être vue comme une relation de « **généralisation/spécialisation** » entre une classe (la super-classe) et plusieurs classes plus spécialisées (ses sous-classes).

L'Héritage en POO : Généralisation/Spécialisation

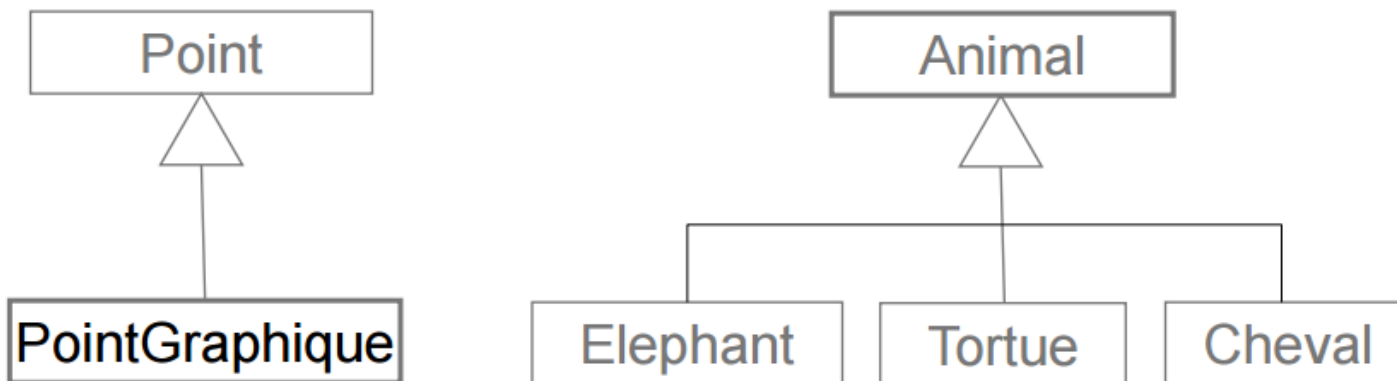
- La généralisation exprime une relation “**est-un**” entre une classe et sa super-classe (chaque instance de la classe est aussi décrite de façon plus générale par la super-classe).



- La spécialisation exprime une relation de “**particularisation**” entre une classe et sa sous-classe (chaque instance de la sous-classe est décrite de manière plus spécifique)

L'Héritage en POO : Généralisation/Spécialisation

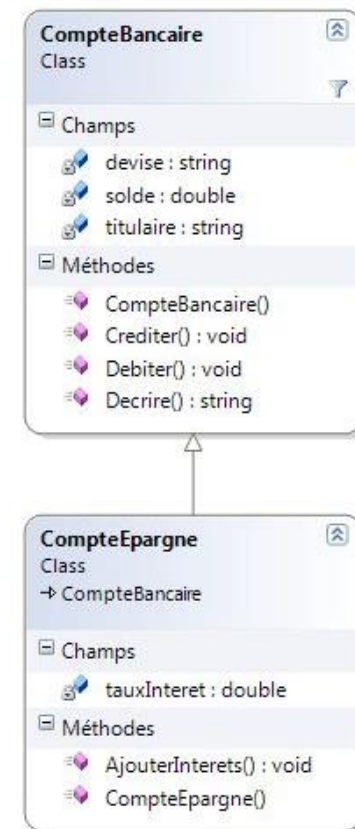
- Utilisation de l'héritage :
 - dans le sens “spécialisation” pour **réutiliser** par modification incrémentielle les descriptions existantes.
 - dans le sens “généralisation” pour **abstraire** en factorisant les propriétés communes aux sous-classes,



L'Héritage en POO : Implémentation en C# et en Java

Reprenons l'exemple du **CompteBancaire**.

Faisons en sorte de créer une class **CompteEpargne** qui
hérite de la classe **CompteBancaire**



L'Héritage en POO : Implémentation en C# et en Java

Code en C#

```
public class CompteEpargne : CompteBancaire
{
    private double tauxInteret;

    // appel du constructeur de la classe CompteBancaire, le mot-clé "base" permet d'accéder à la classe parente
    public CompteEpargne(string leTitulaire, double soldInitial, string laDevise, double leTauxInteret) : base(leTitulaire,
                                                                                                         soldInitial, laDevise)
    {
        // Calcule et ajoute les intérêts au solde du compte
        tauxInteret = leTauxInteret;
    }

    public void AjouterInterets()
    {
        double interets = Solde * tauxInteret; // calcul des intérêts sur le solde
        Solde += interets; // ajout des intérêts au solde
    }
}
```

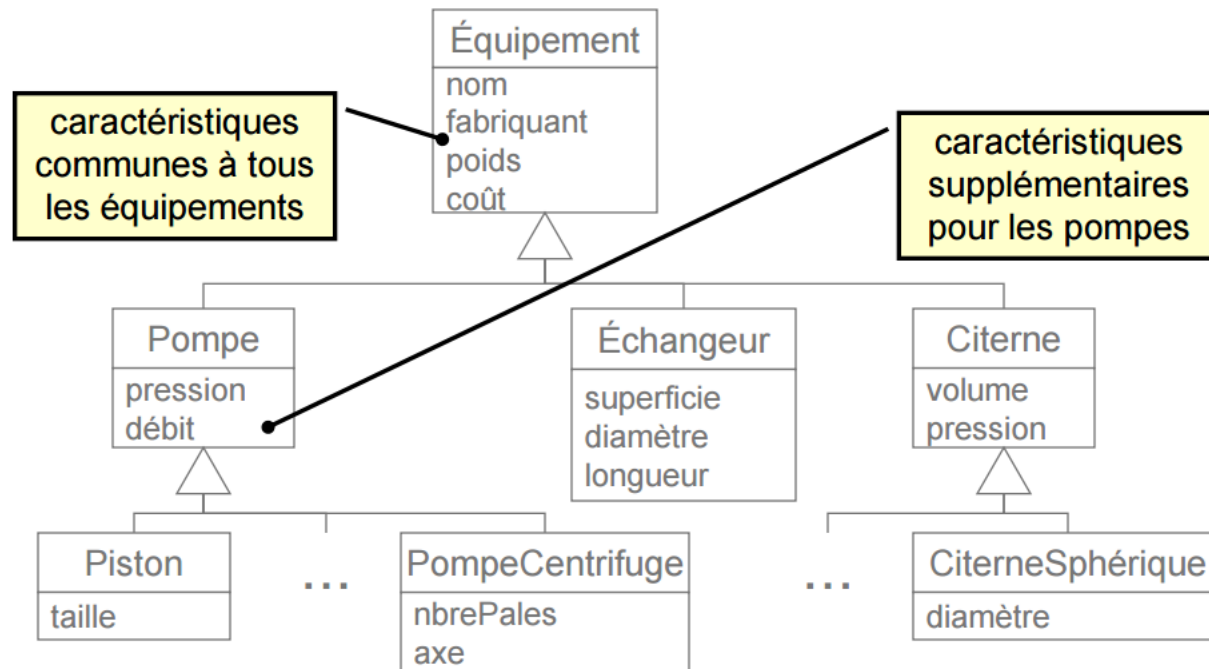
L'Héritage en POO : Implémentation en C# et en Java

Code en Java

```
public class CompteEpargne extends CompteBancaire {  
    private double tauxInteret;  
  
    public CompteEpargne(string leTitulaire, double soldelInitial, string laDevise, double leTauxInteret) {  
  
        //Appel du constructeur de la super-classe grâce au mot clé super  
        super(leTitulaire, soldelInitial, laDevise);  
  
        // Calcule et ajoute les intérêts au solde du compte  
        tauxInteret = leTauxInteret;  
    }  
  
    public void AjouterInterets() {  
        double interets = Solde * tauxInteret; // calcul des intérêts sur le solde  
        Solde += interets; // ajout des intérêts au solde  
    }  
}
```


L'Héritage en POO : L'Héritage à plusieurs niveaux

- pas de limitation dans le nombre de niveaux dans la hiérarchie d'héritage
- méthodes et variables sont héritées au travers de tous les niveaux



L'Héritage en POO : L'Héritage à plusieurs niveaux

```
public class A {  
    public void hello() {  
        System.out.println(«Hello»);  
    }  
}
```

```
public class B extends A {  
    public void bye() {  
        System.out.println(«Bye Bye»);  
    }  
}
```

```
public class C extends B {  
    public void ouns() {  
        System.out.println(«ouns!»);  
    }  
}
```

- Pour résoudre un message, la hiérarchie des classes est parcourue de manière ascendante jusqu'à trouver la méthode correspondante.

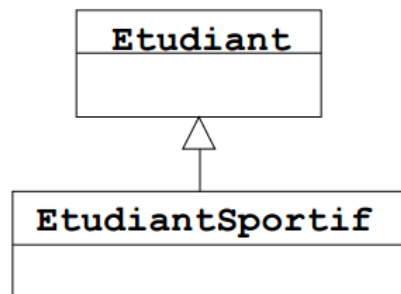
```
C c = new C();  
c.hello();  
c.bye();  
c.ouns();
```

Le Polymorphisme en POO

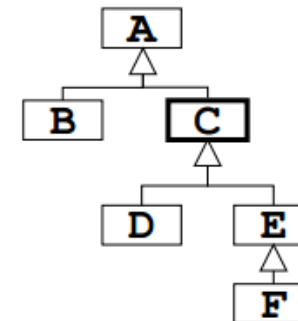
- Le terme polymorphisme décrit la caractéristique d'un élément qui peut se présenter sous différentes formes.
- En programmation par objets, on appelle polymorphisme
 - *le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe.*
 - *le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.*
- "Le **polymorphisme** constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage" Bruce Eckel "Thinking in JAVA"

Le Polymorphisme en POO : **Surclassement**

- tout objet instance de la classe **B** peut être aussi vu comme une instance de la classe **A**.
- à une référence déclarée de type **A** il est possible d'affecter une valeur qui est une référence vers un objet de type **B** (**surclassement** ou **upcasting**)



```
Etudiant e;  
e = new EtudiantSportif(...);
```



```
C c;  
c = new D();  
c = new E();  
c = new F();
```

Le Polymorphisme en POO : Surclassement

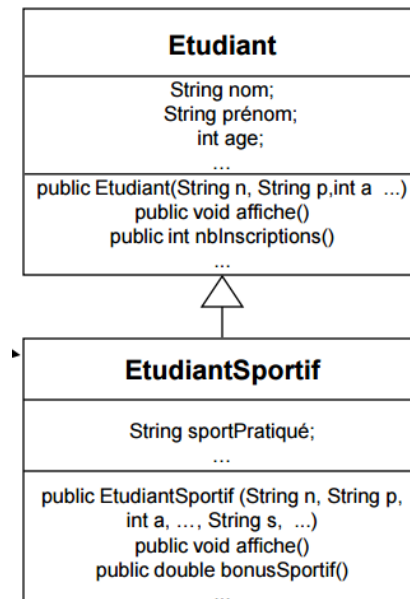
- Lorsqu'un objet est "sur-classé" il est vu par le compilateur comme un objet du type de la référence utilisée pour le désigner
 - Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence*

```
EtudiantSportif es;  
es = new EtudiantSportif("DUPONT", "Jean",  
                          25, ..., "Badminton", ...);
```

```
Etudiant e;  
e = es; // upcasting
```

```
e.nbInscriptions();  
es.nbInscriptions();
```

```
es.bonusSportif();  
e.bonusSportif();
```

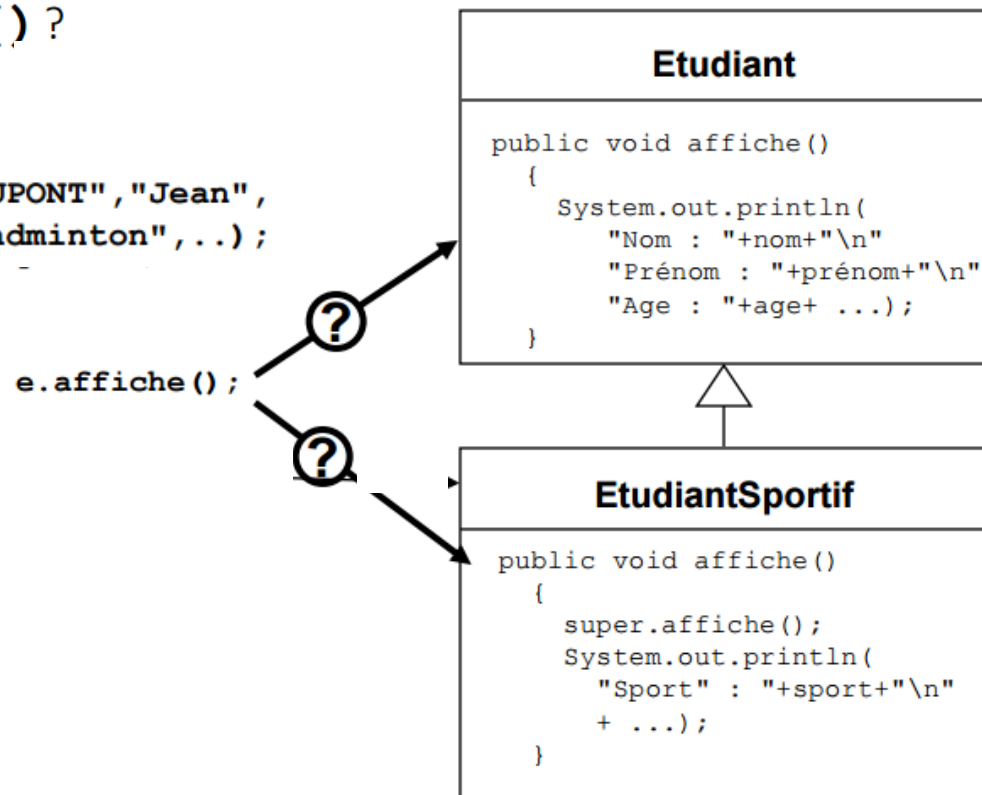


Le Polymorphisme en POO : Surclassement

- Que va donner **e.affiche()** ?

```
EtudiantSportif es;  
es = new EtudiantSportif("DUPONT", "Jean",  
    25, ..., "Badminton", ...);
```

e.affiche();



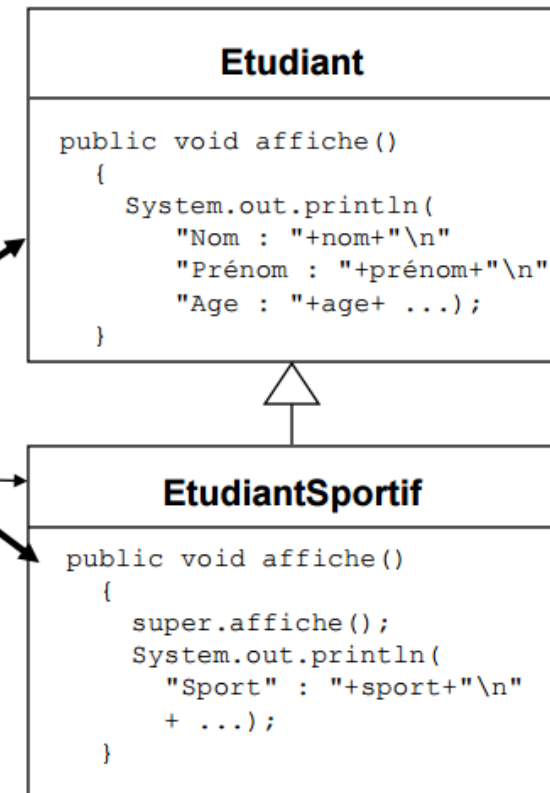
Le Polymorphisme en POO : Surclassement

- Que va donner **e.affiche()** ?

```
EtudiantSportif es;
es = new EtudiantSportif("DUPONT", "Jean",
    25, ..., "Badminton", ...);
```

e.affiche();

Lorsqu'une méthode d'un objet est accédée au travers d'une référence "surclassée", c'est la méthode telle qu'elle est définie au niveau de **la classe effective** de l'objet qui est en fait invoquée et exécutée



Le Polymorphisme en POO : Redéfinition (Overriding)

• Redéfinition d'une méthode (method overriding)

- lorsque la classe définit une méthode dont le nom, le **type de retour** et le type des arguments sont identiques à ceux d'une méthode dont elle hérite

```
public class A {

    public void hello() {
        System.out.println(«Hello»);
    }

    public void affiche() {
        System.out.println(«Je suis un A»);
    }
}
```

```
public class B extends A {

    public void affiche() {
        System.out.println(«Je suis un B»);
    }
}
```

Code en Java

```
A a = new A();
```

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

```
a.hello();    --> Hello
```

```
a.affiche();  --> Je suis un A
```

```
b.hello();    --> Hello
```

```
b.affiche();  --> Je suis un B
```


Le Polymorphisme en POO : Redéfinition (Overriding)

- **Redéfinition d'une méthode (method overriding)**
 - lorsque la classe définit une méthode dont le nom, le **type de retour** et le type des arguments sont identiques à ceux d'une méthode dont elle hérite

Code en C#

```
public class B : A
{
    public override void affiche()
    {
        Console.WriteLine("Je suis un B");
    }
}
```

Le Polymorphisme en POO : Redéfinition (Overriding)

- possibilité de réutiliser le code de la méthode héritée (**super** / **Base** en C #)

```
public class Etudiant {
    String nom;
    String prénom;
    int age;
    ...
    public void affiche()
    {
        System.out.println("Nom : " + nom + " Prénom : " + prénom);
        System.out.println("Age : " + age);
        ...
    }
    ...
}
```

this permet de faire référence à l'objet en cours

super permet de désigner la superclasse

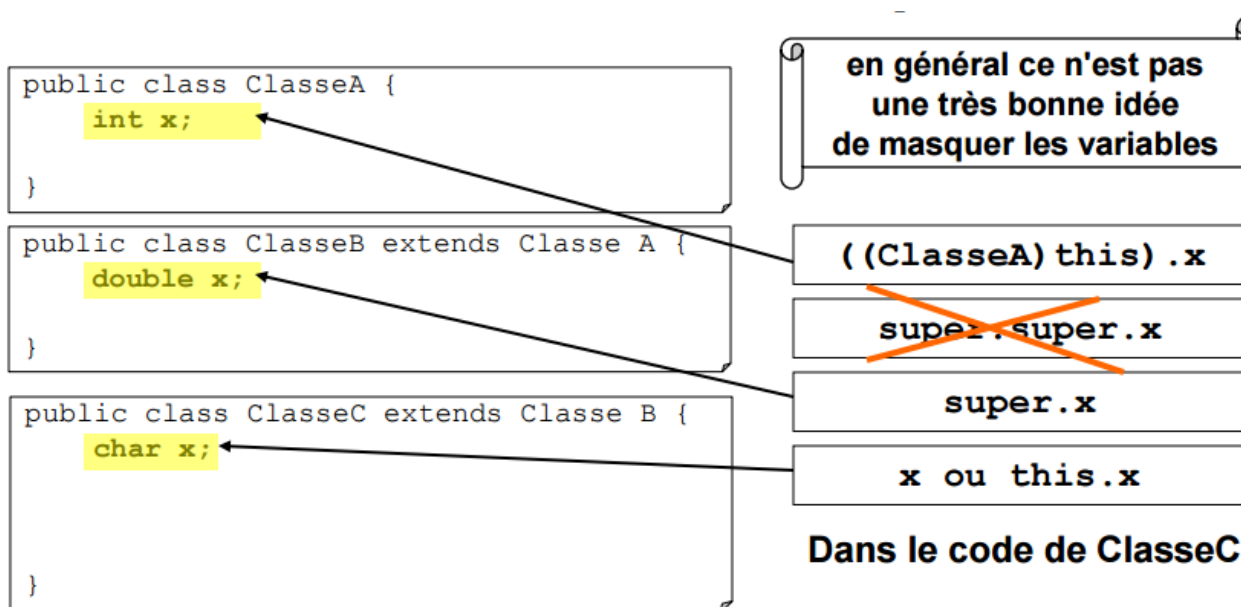
```
public class EtudiantSportif extends Etudiant {
    String sportPratiqué;
    ...
    public void affiche()
    {
        super.affiche();

        System.out.println("Sport pratiqué : "+sportPratiqué);
        ...
    }
}
```

l'appel **super** peut être effectué
n'importe où dans le corps de la
méthode

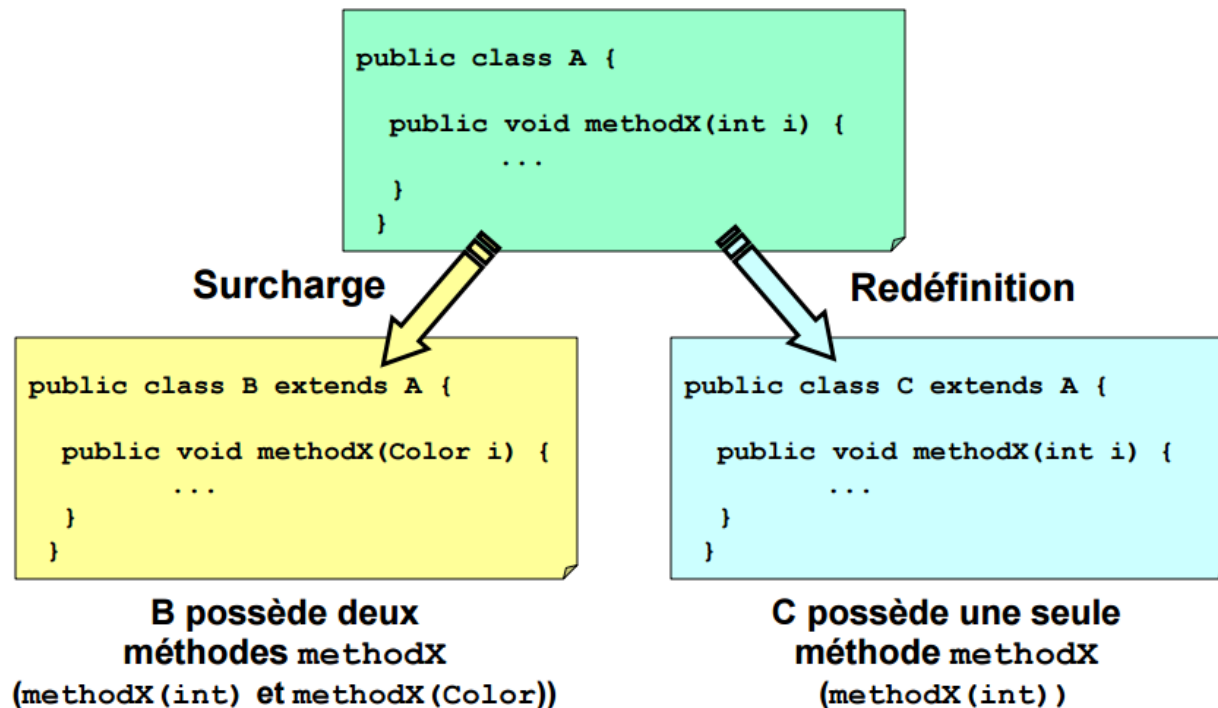
Le Polymorphisme en POO : Redéfinition des attributs

- Lorsqu'une sous classe définit une variable d'instance dont le nom est identique à l'une des variables dont elle hérite, **la nouvelle définition masque la définition héritée**




Le Polymorphisme en POO : Redéfinition Vs Surchage

- Ne pas confondre **redéfinition** (*overriding*) avec **surchage** (*overloading*)



Le Polymorphisme en POO : **Surcharge (Overload)**

Le choix de la méthode à exécuter est effectué à la compilation en fonction des types déclarés : **Sélection statique**



```
public class ClasseC {  
    public static void methodeX(ClasseA a) {  
        System.out.println("param typeA");  
    }  
    public static void methodeX(ClasseB b) {  
        System.out.println("param typeB");  
    }  
}
```

Le Polymorphisme en POO : **comparer des objets**

```
public class Object {
    ...
    → public boolean equals(Object o)
        return this == o;
    }
    ...
}
```

```
public class Point {

    private double x;
    private double y;

    ...

}
```

Méthode: **equals()** en java et **Equals()** en C#

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
```

```
Object o = p2;
p1.equals(o)    --> false 😞
```

```
o.equals(p1)    --> false
```

Le Polymorphisme en POO : comparer des objets

```
public class Object {
    ...
    public boolean equals(Object o)
        return this == o
    }
    ...
}
```

```
public class Point {
    private double x;
    private double y;
    ...
}
```

```
@Override
public boolean equals(Object o) {
    if (this == o)
        return true;

    if (! (o instanceof Point))
        return false;

    Point pt = (Point) o; // downcasting
    return this.x == pt.x && this.y == pt.y;
}
```

redéfinir (overrides) la méthode
equals(Object o) héritée de Object

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
```

```
p1.equals(p2)    --> true
```

```
Object o = p2;
```

```
p1.equals(o)    --> true
```



```
o.equals(p1)    --> true
```

Classes et Méthodes Statiques en POO

Méthodes et classes statiques

- Une méthode **statique** est une méthode qui peut être appelée même **sans avoir instancié la classe**.
- Une **méthode statique** ne peut **accéder** qu'à des **attributs et méthodes statiques**.
- Si une **classe** est définie **statique**, **tous les membres** de cette classe doivent être **statiques**.
- Utilisation du mot clé **static** en Java et C# pour définir une classe ou membre statique.

Classes et Méthodes Statiques en POO

```
public class Test
{
    public int test;
    public static String chaine = "bonjour";

    public Test()
    {
        MaMethodeStatique();
    }

    public static void MaMethodeStatique()
    {
        int nombre = 10;

        System.out.println("Appel de la méthode statique : " + nombre + chaine);
    }
}
```

exemple en java

Classes et Méthodes Abstraites en POO

Méthodes et classes abstraites

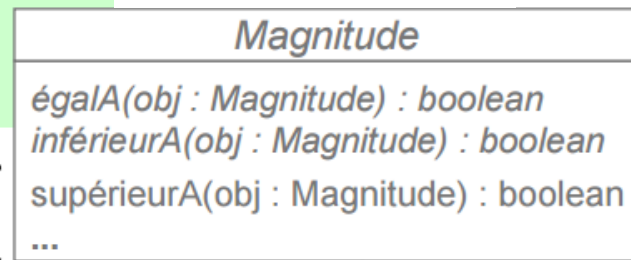
- une classe **abstraite** définit un **concept abstrait**, **incomplet** ou **théorique**.
- Une classe abstraite **regroupe** des caractéristiques **communes** à plusieurs classes **dérivées**.
- Une classe abstraite **n'est pas** destinée à être **instanciée**.
- Utilisation du mot clé **abstract** en Java et en C#.

Classes et Méthodes Abstraites en POO

```
public abstract class Magnitude {  
  
    public abstract boolean egalA(Magnitude m) ;  
  
    public abstract boolean inferieurA(Magnitude m) ;  
  
    public boolean superieurA(Magnitude m) {  
        return !egalA(m) && !inferieurA(m);  
    }  
    ...  
}
```

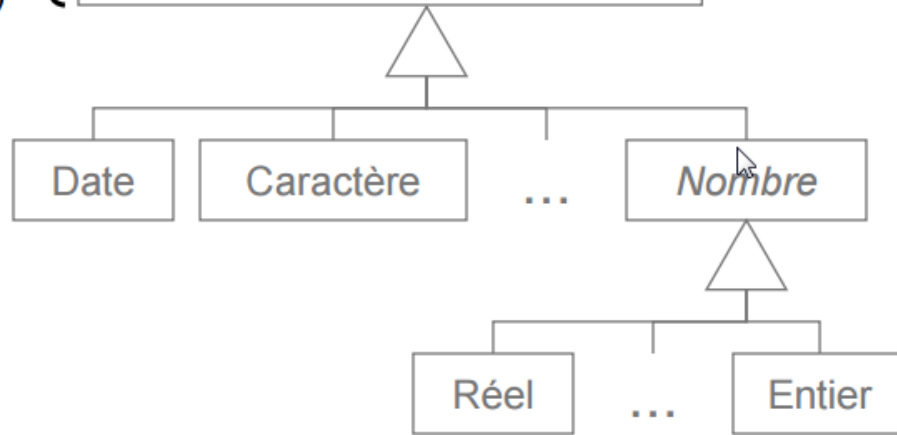
Classe abstraite

opérations concrètes
(basées sur les 2
opérations abstraites)



opérations
abstraites

chaque sous-classe
concrète admet une
implémentation différente
pour `egalA()` et
`inferieurA()`

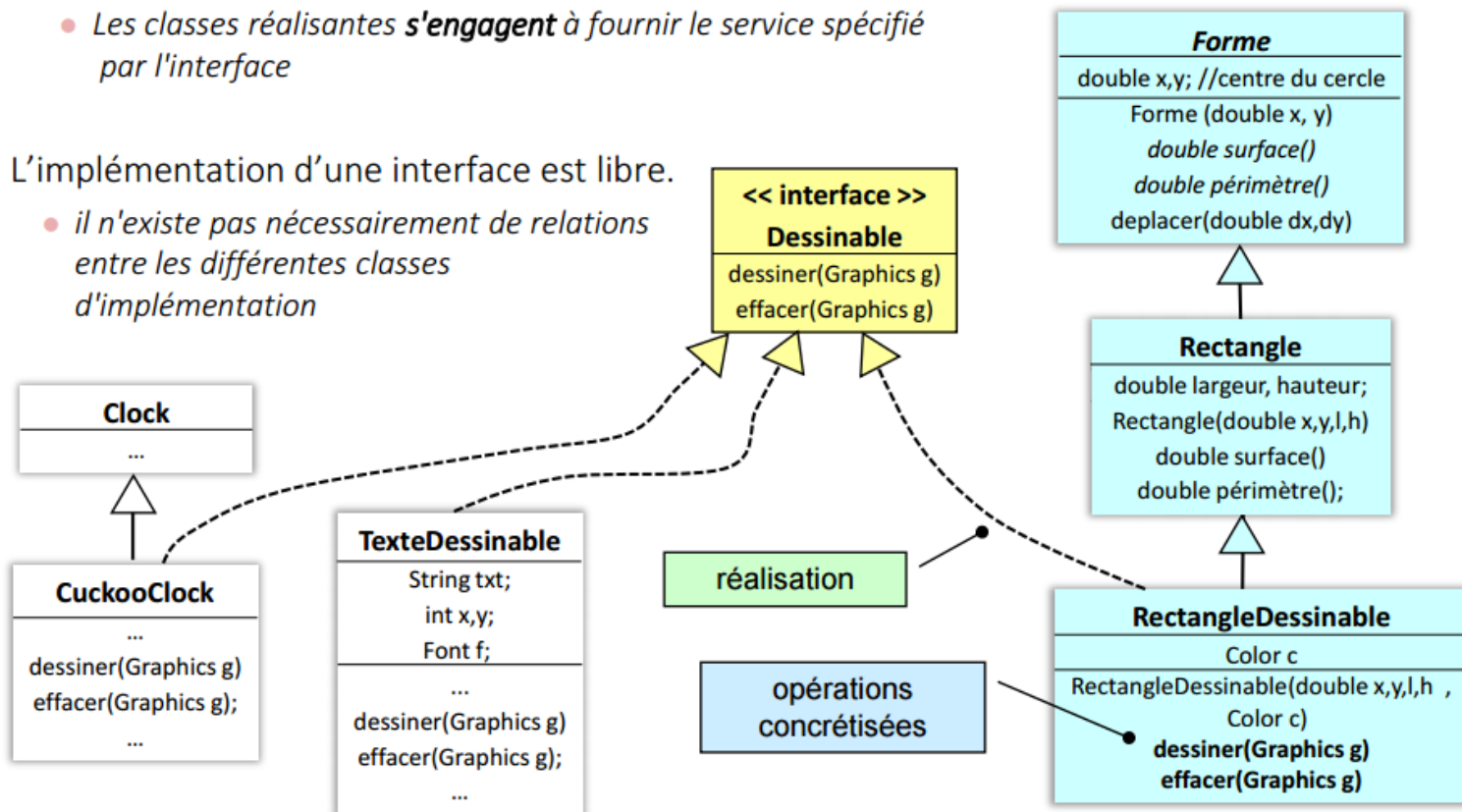


Les Interfaces en POO

- Une **interface** est une **collection d'opérations** utilisée pour spécifier **un service offert** par une **classe**.
- Une **interface** peut être vue comme une **classe 100% abstraite** sans **attributs** et dont toutes les **opérations sont abstraites**.
- Une **interface** est destinée à être “réalisée” (**implémentée**) par d'**autres classes** (celles-ci en héritent toutes les descriptions et concrétisent les opérations abstraites).
- De la même manière qu'une **classe** étend sa **super-classe** elle peut de manière optionnelle **implémenter une ou plusieurs interfaces**.
- Si une classe peut avoir des sous-classes, une interface peut avoir également des "sous-interfaces"

Les Interfaces en POO

- Les classes réalisantes **s'engagent** à fournir le service spécifié par l'interface
- L'implémentation d'une interface est libre.
 - il n'existe pas nécessairement de relations entre les différentes classes d'implémentation



Les Classes Internes en POO : Implémentation en Java et C#

Exemple d'une Interface en Java et en C#

Code en C#	Code en Java
<pre>public interface IDessinable { public void dessinerr(Graphics g); public void effacer(Graphics g); }</pre>	<pre>public interface Dessinable { public void dessinerr(Graphics g); public void effacer(Graphics g); }</pre>

Les Classes Internes en POO : Implémentation en Java et C#

Exemple de classe réalisant une interface en Java et en C#

Code en Java

```
class RectangleDessinable extends Rectangle implements Dessinable
{
    private Color c;

    public RectangleDessinable(double x, double y,
                               double l, double h, Color c) {
        super(x,y,l,h);
        this.c = c;
    }

    public void dessiner(Graphics g){
        g.drawRect((int) x, (int) y, (int) largeur, (int) hauteur);
    }
    public void effacer(Graphics g){
        g.clearRect((int) x, (int) y, (int) largeur, (int) hauteur);
    }
}
```

Les Classes Internes en POO : Implémentation en Java et C#

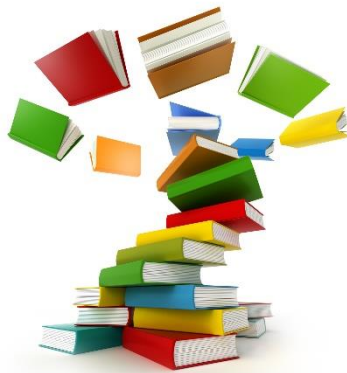
Exemple de classe réalisant une interface en Java et en C#

Code en Java

```
class RectangleDessinable : IDessinable
{
    private Color c;

    public RectangleDessinable(double x, double y,
        double l, double h, Color c) {
        super(x,y,l,h);
        this.c = c;
    }

    public void dessiner(Graphics g){
        g.drawRect((int) x, (int) y, (int) largeur, (int) hauteur);
    }
    public void effacer(Graphics g){
        g.clearRect((int) x, (int) y, (int)largeur, (int) hauteur);
    }
}
```

Merci

pour votre attention

