Cégep de Saint-Hyacinthe
Département d'informatique

Programmation orientée objet

420-2DP-HY

(3-3-3)

Notions sur les objets - Héritage (Introduction)

(Version 1.1)

3 heures

Préparé par

Martin Lalancette

Comprendre les éléments suivants :

- Classe de base et dérivée
- Diagramme de classes
- *Virtual* et *override*
- Modificateur d'accès « *protected* »

Table des matières

Introduction

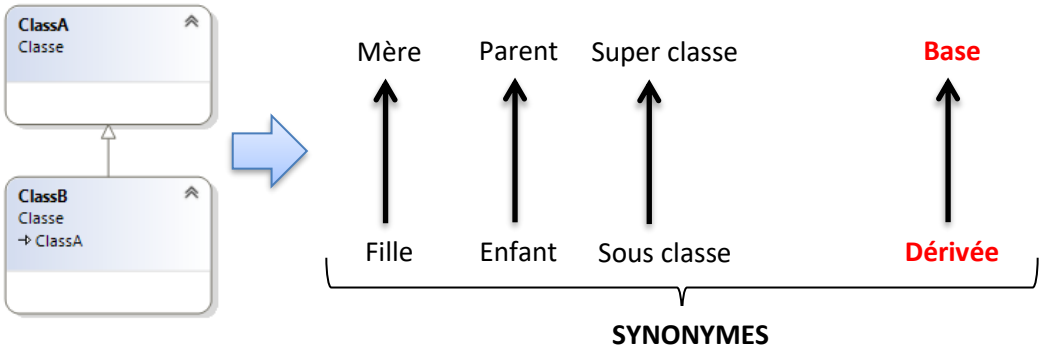
Cette séquence a pour but de vous initier aux nécessaires à la programmation à base d'objets et d'événements. Nous commencerons par énoncer les éléments théoriques appuyés d'exemples simples et faciles à reproduire. Afin d'axer l'attention sur la compréhension de ces notions, il y aura des exercices à faire tout au long de cette séquence. **Cette séquence traitera des notions reliées aux objets axées principalement sur l'héritage entre les objets**. Pour bien suivre les instructions qui vont être mentionnées tout au long des séquences d'apprentissage, une préparation de base s'impose. Il est important de créer un répertoire de travail (sur votre C : ou clé USB). Voici une suggestion d'arborescence :

Exercice 1. : S'assurer d'avoir créé l'arborescence ici haut mentionnée sur votre C ou votre clé USB. Récupérer une **solution de départ** disponible dans LÉA.

Qu'est-ce que l'héritage?

L'héritage est un concept permettant de créer une nouvelle classe à partir d'une classe déjà existante afin de pouvoir utiliser ses propriétés et ses méthodes pour ainsi ajouter ou modifier des fonctionnalités. C'est le concept le plus important de la programmation orientée objet (POO). En terme de programmeur, ce concept est utilisé pour **factoriser** le code. C'est-à-dire d'éviter de répliquer certaines fonctionnalités dans diverses classes, donc **centraliser le code**. L'héritage est important, car il permet de réutiliser l'existant pour l'adapter à de nouveaux besoins.

Voici les terminologies:



Les règles fondamentales concernant les classes (de base ou dérivée) ne changent pas :

- Une classe déclare des membres
- Une classe est propriétaire des membres qu'elle déclare
- Une classe gère l'accessibilité des membres par les modificateurs d'accès (private, public…).

Classe de base

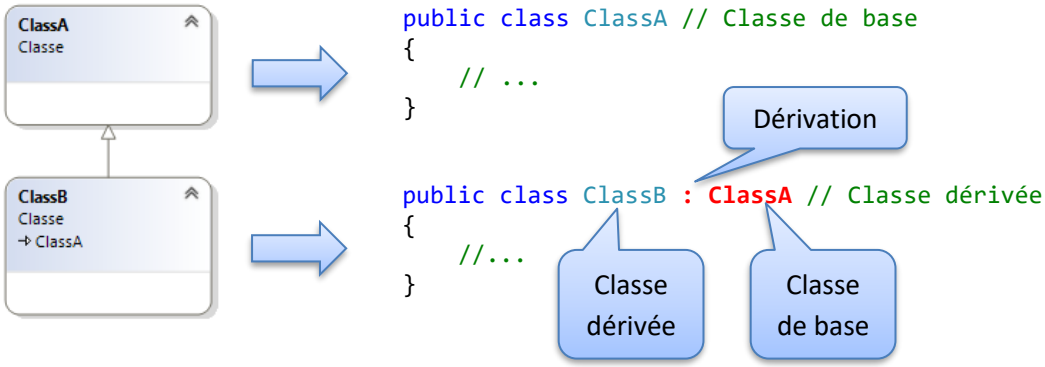
Une classe de base est une classe **qui offre** à d'autres classes, **ses membres en héritage**.

Classe dérivée

Une classe dérivée est une classe **qui demande** à une classe de base **l'héritage de ses membres**. À savoir :

- En C#, une classe ne peut hériter que **d'une seule classe** (ATTENTION ne pas confondre avec les interfaces).
- Lorsqu'une classe dérive d'une classe de base, elle peut accéder à tous les membres de la classe de base **qui ne sont pas privés**. Le choix des modificateurs d'accès (private, public, protected) est très important dans la définition de la classe de base (ainsi que la dérivée).
- Les membres hérités ne changent pas de propriétaire. Ils appartiennent à la classe de base.
- La classe de base est **un noyau** autour duquel la classe dérivée ajoute sa propre **couche**.

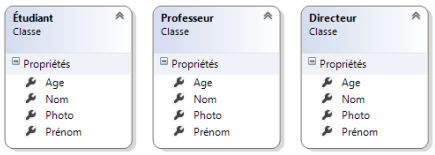
La ponctuation associée à la dérivation est le « **:** ». Voici un exemple :



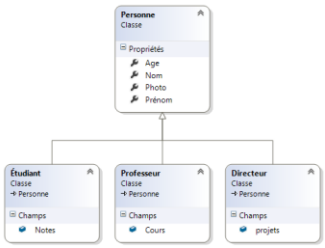
Voici maintenant un exemple concret d'utilisation de l'héritage.

Exemple #1 : Je veux gérer les informations concernant un **étudiant**, un **professeur** et un **directeur**. Ils possèdent tous un nom, un prénom, un âge et une photo.

Avec les notions vues jusqu'à maintenant nous pourrions créer des classes de la façon suivante :



Cependant si nous portons une attention particulière à ces classes, nous constatons que nous répétons la même information dans chacune d'elle. Voici comment optimiser le tout en utilisant l'héritage. Il suffit de créer une classe de base contenant les **informations communes**. Ici, je crée une classe de base **Personne** qui contiendra les données (Age, Nom, Prénom et photo). Par la suite, nous dérivons les autres classes comme suit :



Dans les classes dérivées, nous y retrouvons alors les informations spécifiques (**Notes** pour l'étudiant, **Cours** pour le professeur et **Projets** pour le directeur). Voici maintenant le :

```
public class Personne
{
    #region Propriétés
    public string Nom { get; set; }
    public string Prénom { get; set; }
    public int Age { get; set; }
    public Image Photo { get; set; }
    #endregion
}
```

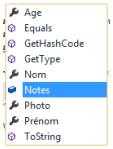
```
public class Étudiant :
Personne
{
 public List<int> Notes;
}
```

```
public class Professeur :
Personne
{
 public List<string> Cours;
}
```

```
public class Directeur :
Personne
{
  public List<string> projets;
}
```

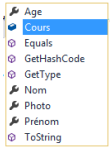
Dans le bouton :

```
...
Étudiant e1 = new Étudiant();
e1.
```



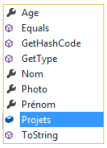
Dans le bouton :

```
...
Professeur p1  = new Professeur();
p1.
```



Dans le bouton:

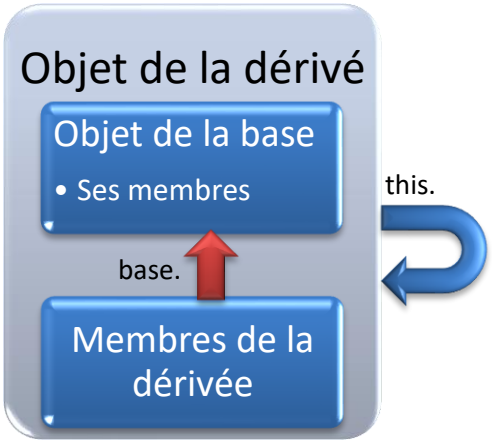
```
...
Directeur d1 = new Directeur();
d1.
```



Étant donné que les propriétés de la classe **Personne** sont publiques, elles sont exposées via les dérivées.

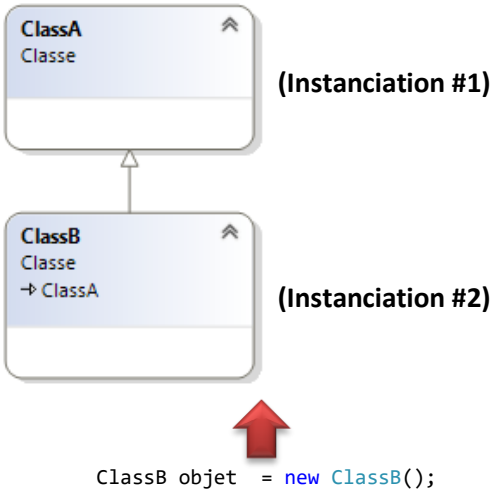
Structure interne d'une instance dérivée

- La classe dérivée possède sa propre référence appelée this.
- La classe dérivée possède une autre référence appelée base.
- La référence base est un lien privé pour accéder aux membres hérités de l'instance de la base.
- La référence base n'est accessible, de nulle part ailleurs qu'à l'intérieur de l'instance dérivée.



Cascade d'instanciations

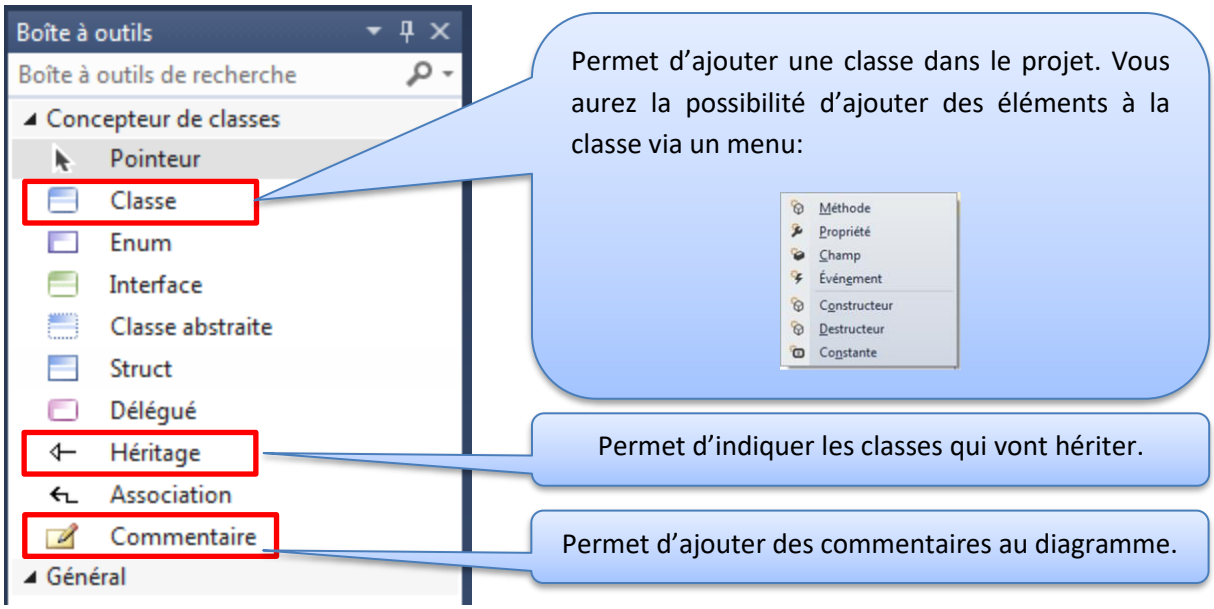
- Chaque instance d'une classe dérivée comprend une instance de la classe base dont elle dépend.
- L'instance de la classe de base doit être construite d'abord puis l'instance de la classe dérivée doit être construite après.
- L'instance finale de la classe dérivée est donc le résultat d'une cascade d'instanciations. En tout temps, au moins 2 instanciations.



Maintenant, regardons un outil intégré dans Visual Studio qui facilite grandement la conception de ce type de classe.

Outil « Diagramme de classes » en XML de Visual Studio

Un diagramme peut être ajouté à un projet en sélectionnant celui-ci dans l'explorateur de solution, en cliquant avec le bouton droit de la souris, choisir **Ajouter/Nouvel élément…/Diagramme de classes**. Une autre façon toute simple est de cliquer sur le projet avec le bouton droit de la souris et de choisir **Affichage/Afficher le diagramme de classes**. S'il n'y a pas déjà de diagramme dans le projet, un nouveau sera créé. Une fois en mode conception, vous aurez accès aux éléments suivants dans la boîte à outils :



Dans cette séquence, nous allons voir les éléments encadrés en rouge. Les autres suivront dans les prochaines séquences.

Exercice 2. : Récupérer une solution de départ. Dans le projet **Théorie**, reproduire le diagramme contenant les classes **Personne**, **Etudiant**, **Professeur** et **Directeur** avec leurs propriétés dans un fichier **ClassesÉcole.cd**. Déclarer une instance de chaque objet dans le bouton associé à cet exercice. Suivre les directives du professeur.

Exercice 3. : Créer le **diagramme de classes** nommé **ClassesAnimaux.cd**. Faire un diagramme de classe qui permet d'optimiser le code servant à contenir les animaux suivants : Saumon, Chien, Chat, Hirondelle, l'homme, crocodile, abeille, truite et le moustique en sachant que les catégories suivantes existent : Mammifères, oiseaux, poissons, reptiles, insectes. Il y a plusieurs niveaux d'héritage dans cet exercice.

Redéfinir une propriété ou une méthode grâce à *virtual* et *override*

Il arrive souvent qu'une propriété ou une méthode ait un certain comportement dans la classe de base et que l'on souhaite changer dans la classe dérivée. On peut le faire grâce aux deux mots-clés *virtual* et *override*:

virtual

Ce mot ajouté devant la déclaration de la propriété ou de la méthode se trouvant dans la **classe de base**, autorise la classe dérivée à redéfinir les actions à faire dans celle-ci.

override

Ce mot ajouté devant la déclaration de la propriété ou de la méthode se trouvant dans la **classe dérivée** portant **le même nom que celle déclarée dans la classe de base** pourra redéfinir les actions de celle-ci.

Exercice 4. : Ajouter la méthode **ObtenirRole**() dans la classe **Personne** qui consiste à retourner « **Je suis une personne**. ». Faire appel à cette méthode pour les trois variables (etudiant, professeur et directeur) et afficher le tout dans **txtRésultat**.

Après cet exercice, nous pouvons constater que les trois appels à la méthode **ObtenirRole** retournent le même résultat. Maintenant dans le prochain exercice nous allons personnaliser la méthode **ObtenirRole** pour chacune des dérivées.

Exercice 5. : Modifier la méthode **ObtenirRole**() dans la classe **Personne** afin **qu'elle autorise** les dérivées à définir leur propre méthode **ObtenirRole**(). Dans le cas de l'étudiant, elle devra retourner « **Je suis un étudiant** », le professeur : « **Je suis un professeur** » et le directeur : « **Je suis un directeur** ». Afficher le tout dans **txtRésultat**.

Appeler des méthodes(propriétés) de la classe de base dans la dérivée

Dans le cas de plusieurs héritages, c'est la méthode la plus dérivée (c'est-à-dire celle se trouvant dans la classe la plus dérivée) qui sera appelée. Cependant, il est possible d'appeler la méthode de la classe de base qui a été redéfinie grâce au mot-clé « **base** ». À savoir :

- La référence base est un lien privé pour accéder aux membres hérités de l'instance de la base.
- La référence base n'est accessible, de nulle part ailleurs qu'à l'intérieur de l'instance dérivée.

Exercice 6. : Modifier la méthode **ObtenirRole** de chaque dérivée afin qu'elle puisse appeler celle de la classe de base et de concaténer son retour avec la mention déjà existante. Afficher le tout dans **txtRésultat**.

Nouveau modificateur d'accès « protected »

Nous avons vu jusqu'à présent deux modificateurs d'accès soit : *public* et *private*. Un petit rappel :

public: Ce modificateur rend accessible un membre d'une classe à l'externe. Les dérivées y ont accès.

private : Ce modificateur rend accessible un membre à l'intérieur de la classe même. Les dérivées **n'y ont pas accès du tout**. Encore moins l'externe.

Il fallait donc trouver un juste milieu pour les dérivées, c'est alors que *protected* fit son apparition.

protected: Ce modificateur rend accessible un membre à l'intérieur de la classe même ainsi qu'aux dérivées de celle-ci. Il est utilisé pour les membres d'une classe de base. Réserve l'accès des membres aux classes dérivées de cette descendance seulement.

Exercice 7. : Ajouter une méthode **CréerIdUnique(char cType)** à la classe **Personne** qui sera appelée dans le constructeur de chaque dérivée. Ajouter un paramètre de type char à cette méthode afin de spécifier le type de personne (ex. 'E'=Étudiant, 'P'=Professeur et 'D'=Directeur). Cette méthode devra concaténer ce type à un GUID (*est un entier 128 bits (16 octets) qui peut être utilisé sur tous les ordinateurs et réseaux lorsqu'un identificateur unique est nécessaire.Un tel identificateur a peu de chance d'être dupliqué.*) et le conserver à l'interne. Vous devez ajouter d'autres membres nécessaires à la classe Personne pour conserver cette information. Cette méthode doit être appelée dans le contructeur de chaque dérivée. Faire afficher un exemple dans txtRésultat.

Quelques exercices

Exercice 8. : Créer le **diagramme de classes** nommé **ClassesEmployés.cd**, en sachant que les **actions sont des méthodes** et les **informations des propriétés**, en fonction des affirmations suivantes : Une compagnie gère des employés qui ont des vocations différentes : Concierges, programmeur, gestionnaires, vendeurs. Tous possèdent un nom, prénom et un numéro d'employé. Les gestionnaires et les vendeurs ont la possibilité d'avoir une voiture fournie par la compagnie. Pour chaque programmeur, nous voulons connaitre la liste des langages de programmation qu'il maîtrise. Les programmeurs ont comme tâches d'analyser, programmer et documenter. Concernant les vendeurs nous souhaitons connaitre le nombre total de kilomètres parcourus. Ils effectuent comme tâches : appeler le client, le rencontrer et signer des contrats. Pour les gestionnaires, nous voulons connaitre la liste des projets en cours et le nombre d'employés sous leur responsabilité. Leurs tâches : Planifier, débuter des projets et le terminer.

Exercice 9. : La compagnie STH Informatique requiert votre expertise afin de créer une classe appelée **STHTextBox** qui servira à encapsuler la classe TextBox du *framework .NET* afin de lui donner un comportement personnalisé. Voici les directives :

- Au départ, la zone de texte doit avoir un fond de couleur JAUNE.
- Lorsque la souris entre dans la zone, le fond devient BLEU
- Lorsque la souris quitte la zone, elle redevient JAUNE
- Lorsque l'utilisateur clique dans la zone, elle devient ROUGE (jusqu'au relâchement du bouton).

Il faut ajouter la ligne suivante pour donner accès à notre objet dans le formulaire (code XAML):

```
xmlns:custom="clr-namespace:Théorie.Models"
```

dans Window

Utilisation :

```
<custom:STHTextBox x:Name="txtChamp1" TextWrapping="Wrap" Margin="3" Grid.Column="1"/>
```

Il ne faut pas nuire au fonctionnement normal de la zone.

Bibliographie

Aucune source spécifiée dans le document actif.