Cégep de Saint-Hyacinthe
Département d'informatique

Programmation orientée objet

420-2DP-HY

(3-3-3)

Notions sur les objets (Héritage - Polymorphisme)

(Version 1.1)

3 heures

Préparé par

Martin Lalancette

Comprendre les éléments suivants :

- Polymorphisme
- Cascade de constructeurs
- Les classes scellées
- Classes abstraites
- Interfaces

Table des matières

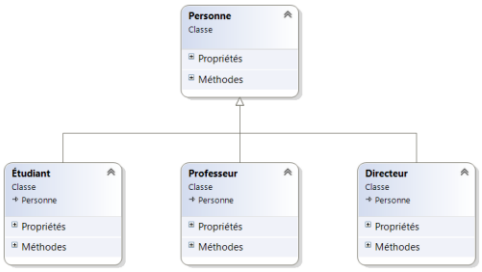
Introduction

Cette séquence a pour but de vous initier aux nécessaires à la programmation à base d'objets et d'événements. **Cette séquence traitera des notions reliées aux objets axées principalement sur le polymorphisme dans 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.

Initiation au polymorphisme

Dans la séquence précédente nous avions modélisé les classes suivantes :

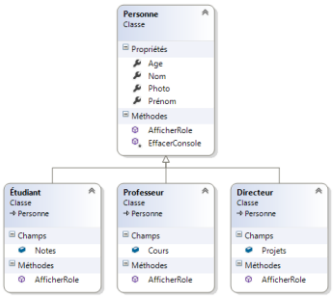


En considérant ces héritages, nous pouvons affirmer ceci :

- Un étudiant est une personne
- Un professeur est une personne
- Un directeur est une personne
- Mais personne **n'est pas forcément** un étudiant ou un professeur ou un directeur.

Ceci me permet de vous introduire au concept du « **polymorphisme** ». « Le nom de **polymorphisme** vient du grec et signifie ***qui peut prendre plusieurs formes.*** » (Comment ça marche .NET, 2013). En l'appliquant aux notions

informatiques, nous obtenons la définition suivante : « Par héritage, une classe peut être utilisée comme plusieurs types; elle peut être utilisée comme **son propre type**, tout type de base ou tout type d'interface si elle implémente des interfaces.» (MSDN, 2013). Revenons avec notre exemple :



Nous savons jusqu'à maintenant que nous pouvons déclarer nos variables de la façon suivante :

```
Étudiant étudiant = new Étudiant();
Professeur professeur = new Professeur();
Directeur directeur = new Directeur();
```

Mais, grâce au polymorphisme, nous pouvons déclarer les mêmes variables avec le type de base comme suit :

```
Personne étudiant = new Étudiant();
Personne professeur = new Professeur();
Personne directeur = new Directeur();
```

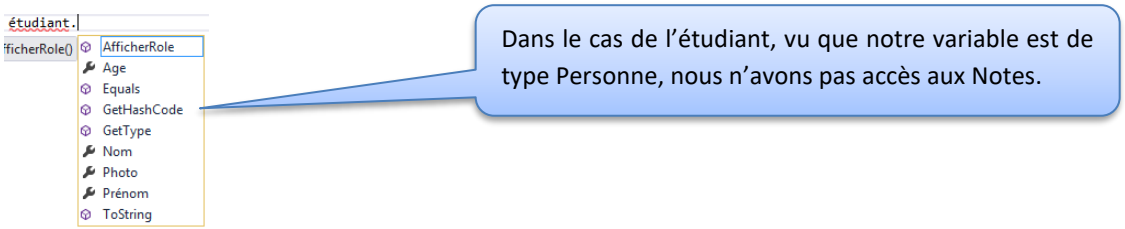
Voici le résultat :

```
Je suis un étudiant.
Je suis un professeur.
Je suis un directeur.
```

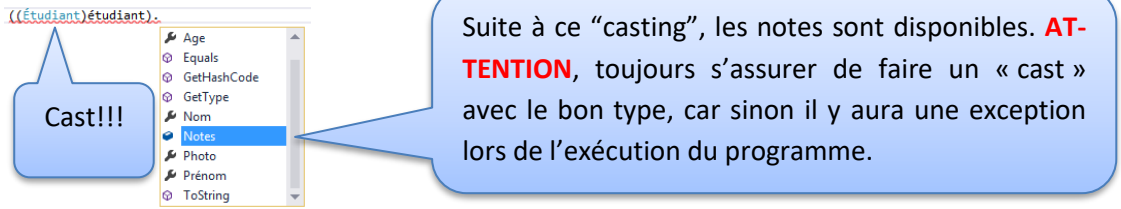
Dans les deux déclarations si j'appelle les lignes suivantes, nous obtenons les mêmes résultats :

```
étudiant.ObtenirRole();
professeur.ObtenirRole();
directeur.ObtenirRole();
```

Étant donné que nous avons fait des *override* sur la méthode ObtenirRole() dans chaque dérivé, même si nous appelons celle se trouvant dans Personne, c'est la méthode ObtenirRole de chaque dérivée qui est exécutée. Cependant, dans la deuxième déclaration, vous n'avez pas accès aux spécificités des classes dérivées (Ex. : les **notes** pour l'étudiant, les **cours** pour le professeur et les **projets** pour le directeur). Exemple :



Pour y avoir accès, il suffit d'effectuer un « cast » en fonction du type de la dérivée. Exemple :



Exercice 2. : Récupérer la solution de la séquence précédente et ouvrir le projet **Théorie :**

1) Modifier la méthode ObtenirRole de chaque dérivée pour ajouter le prénom et le nom dans l'affichage. Mettre en commentaire de la méthode de la base s'il y a lieu.
2) Dans MainWindow, ajouter une liste de personne qui va contenir les individus suivants :
   - Line Savard (étudiante)
   - Jade Couture (professeur)
   - Serge Roy (étudiant)
   - Carl Fortin (directeur)
   - Marc Tremblay (étudiant)
   - Simon Boulay (professeur)
3) Afficher la liste dans txtRésultat en les regroupant par type (LINQ). Exemple :

```
Théorie.Professeur:
        Je suis une personne: Jade Couture
        Je suis une personne: Simon Boulay
Théorie.Étudiant:
        Je suis une personne: Marc Tremblay
        Je suis une personne: Line Savard
        Je suis une personne: Serge Roy
Théorie.Directeur:
        Je suis une personne: Carl Fortin
```

Les opérateurs *is* et *as*

Il existe deux opérateurs qui permettent d'effectuer des opérations de « casting » de façon sécuritaire, c'est-à-dire qu'ils permettent d'éviter que des exceptions surviennent.

Opérateur	Description

is	Vérifie si un objet est compatible avec un type donné. Retourne *true* ou false. Exemples :
	`Directeur directeur = new Directeur();`
	`Personne personne = new Personne();`
	`if (personne is Directeur) // Ici, faux`
	`{`
	`    // ...`
	`}`
	
	`if (directeur is Directeur) // Ici, vrai`
	`{`
	`    // ...`
	`}`
	
	`if (directeur is Personne) // Ici, vrai`
	`{`
	`    personne = (Personne)directeur;`
	`}`

as	Permet certains types de conversions entre des types de référence compatibles ou des types Nullable.
	Exemple #1 :
	`Directeur directeur = new Directeur();`
	`Personne personne = directeur as Personne; // Effectue le cast.`
	`if (personne != null) // Ici, vrai`
	`{`
	`    // ...`
	`}`
	
	Exemple #2 :
	`Personne personne = new Personne();`
	`Directeur directeur = personne as Directeur;`
	`if (directeur != null) // Ici, faux`
	`{`
	`    // ...`
	`}`

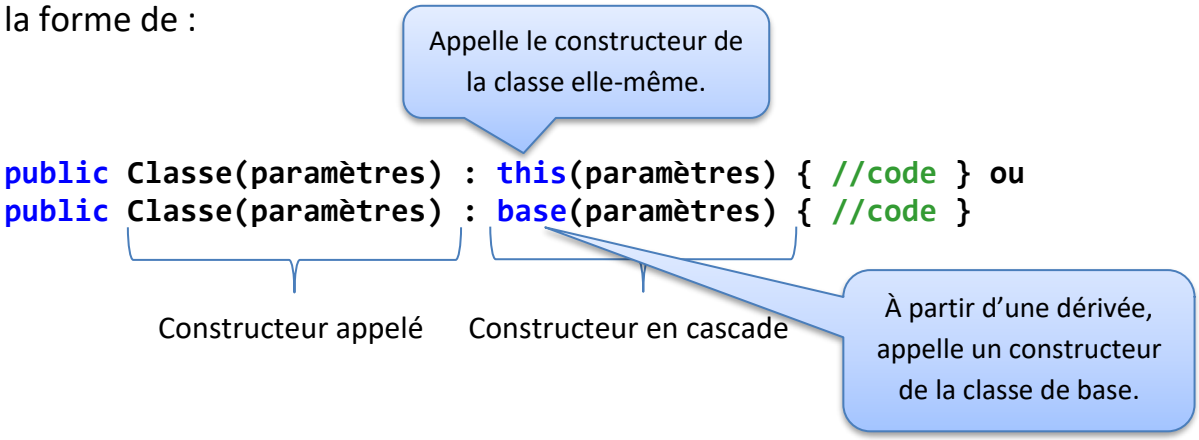
L'héritage et les instanciations (Cascade de constructeurs)

Comment se produit l'instanciation entre la classe de base et la dérivée? Voici la réponse :

- Chaque instance d'une classe dérivée comprend une instance de la classe de 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**.
- Donc, l'instance finale de la classe dérivée est donc le résultat d'une cascade d'instanciations. En tout temps, au moins 2 instanciations.

Lorsque l'on mentionne le mot « instanciation », il y a le mot « construction » qui nous vient à l'esprit. Donc, qui dit « Construction » en programmation dit « faire appel à un constructeur d'une classe ». Dans la déclaration d'une classe, il est possible de lui indiquer d'effectuer une **cascade de constructeurs**. Une cascade de constructeurs est ***l'appel d'un constructeur par un autre constructeur***.

L'appel est effectué à la fin de la ligne de déclaration du constructeur sous la forme de :



La **cascade** est **exécutée avant** et le **code du constructeur** est **exécuté après**. On peut réutiliser les paramètres reçus par le constructeur appelé (à gauche du :) pour alimenter l'appel du constructeur (à droite du :).

Exemple #1 : Ajouter deux constructeurs dont le premier prend le nom et le prénom en paramètre. Le deuxième prend le nom, le prénom et l'âge. Ces paramètres servent à initialiser les propriétés de la classe.

Code non factorisé:

```
public Personne(string sPrénom, string sNom)
{
    Nom = sNom;
    Prénom = sPrénom;
}

public Personne(string sPrénom, string sNom,
                int iAge)
{
    Nom = sNom;
    Prénom = sPrénom;
    Age = iAge;
}
```

Code factorisé:

```
public Personne(string sPrénom, string sNom)
{
    Nom = sNom;
    Prénom = sPrénom;
}

public Personne(string sPrénom, string sNom,
                int iAge) : this(sPrénom, sNom)
{
    Age = iAge;
}
```

Exercice 3. : Ajouter les constructeurs ci-haut factorisés dans la classe Personne.

Exercice 4. : Ajouter un **constructeur sans paramètre** à la classe **Personne** qui initialise le nom et le prénom avec la valeur « Inconnu ». Utiliser la cascade des constructeurs.

Exercice 5. : Ajouter un **constructeur avec les paramètres (nom, prénom et âge)** à la classe **Étudiant** qui fera appel au constructeur de la classe Personne correspondant. Utiliser la cascade des constructeurs.

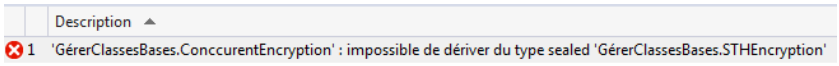
Les classes scellées

Une classe scellée ***signifie qu'elle ne peut être dérivée***. Quand devrait-on sceller nos classes? Selon certaines littératures, **la réponse est le plus souvent possible!!!** Il est préférable de sceller nos classes par défaut. Pourquoi? Pour la simple et bonne raison que l'héritage est un mécanisme qui ne s'improvise pas. Exemple : Vous créez une classe rapidement pour certains besoins ponctuels en vous disant que vous allez y revenir plus tard pour la refaire adéquatement. Six mois plus tard, le temps vous est alloué

pour la retravailler, mais vous vous apercevez que plusieurs de vos collègues ont dérivé de votre classe pour programmer d'autres produits de la compagnie. Imaginez l'impact??? Donc, le mot-clé à utiliser est *sealed*.

```
public sealed class STHEncryption
{
    //...
}

public class ConcurentEncryption : STHEncryption
{
    //...
}
```



Exercice 6. : Ajouter le mot-clé *sealed* à la classe Personne. Qu'arrive-t-il? Après explications, retirer ce mot.

Exercice 7. : Empêcher la dérivation des classes : Étudiant, Directeur et Professeur.

Les classes abstraites

Une classe abstraite ***sert à imposer la présence d'un ou de plusieurs membres dans toute la descendance de la classe de base***. À savoir :

- Vous pouvez déclarer une classe comme abstraite si vous souhaitez empêcher l'instanciation directe au moyen du mot clé new . Dans ce cas, la classe peut être utilisée uniquement si une classe nouvelle est dérivée d'elle.
- Utilisé pour la classe de base et surtout pour les propriétés, indexeurs, méthodes.
- Les membres imposés n'ont que leurs signatures. Donc par de bloc { } et pas de code.
- À partir du moment où un membre *abstract* existe dans une classe, la dite classe doit aussi être déclarée *abstract*.
- Une classe déclarée *abstract* ne peut être instanciée, car certains membres n'auront aucun code à exécuter!
- Mot-clé → abstract.

Exemple :

```
abstract class Base
{
    // Propriété déjà implémenté
    // en incluant le get et le set
    public string Propriete { get; set; }
    // Indexeur à implémenter par les dérivées
    // incluant le get (et pas le set)
    public abstract string this[ int index ] { get; }
    // Méthode à implémenter
    public abstract void MethodeBase( int entier );
}
class Derivee : Base
{
    public override string this[ int i ] { get { return ":)"; } }
    public override void MethodeBase( int e ) { return; }
}
```

Touche à retenir : CTRL + . → Pour implémenter les membres abstraits d'une classe abstraite.

Exercice 8. : Créer un diagramme de classe nommé **Formes.cd**. Créer une classe abstraite de base nommée **Forme**. Elle doit contenir les propriétés suivantes : PositionX (int), PositionY(int), CouleurFond(Brush). Ajouter une classe **Cercle** dérivée de Forme ayant la propriété **Rayon** (int). Ajouter une classe **Carré** dérivée de Forme ayant la propriété **Côté** (int). Ajouter une classe **Rectangle** dérivée de Forme ayant les propriétés **Largeur** (int) et **Hauteur** (int).

Forme	Périmètre	Aire
Carré	4 X Côté	Côté X Coté
Rectangle	(Largeur X Hauteur) X 2	Largeur X Hauteur
Cercle	2 X Rayon X PI	$Rayon^2$ X PI
Triangle	Coté + Coté + Côté	Base X Hauteur / 2

Exercice 9. : Ajouter une propriété **Aire** (double) qui oblige les dérivées à implémenter une propriété **Aire** (double) et d'effectuer le calcul en fonction de ce qu'elle représente. Ajouter un bouton et coder des exemples.

Exercice 10. : Ajouter une propriété **Périmètre** (double) qui oblige les dérivées à implémenter une propriété **Périmètre** (double) et d'effectuer le calcul en fonction de ce qu'elle représente. Ajouter un bouton et coder des exemples.

Nous pouvons ajouter à notre vocabulaire les 2 nouvelles formulations qui décrivent une classe :

- **Classe abstraite** : La classe est abstraite ne peut être instanciée parce qu'elle contient au moins un membre abstrait non fonctionnel (qui ne peut être exécuté, car pas de code).
- **Classe concrète** : est une classe pouvant être instanciée, car tous ses membres sont fonctionnels.

Les interfaces

Une interface contient uniquement les signatures des méthodes, des propriétés, des événements ou des indexeurs.Une classe qui implémente l'interface doit implémenter les membres de l'interface spécifiés dans la définition d'interface. À savoir :

- Une interface est comme une classe de base abstraite. Toute classe qui implémente l'interface doit implémenter tous ses membres.
- Une interface ne peut pas être instanciée directement. Ses membres sont implémentées par une classe qui implémente l'interface.
- Les interfaces ne contiennent aucune implémentation de méthodes.
- Une classe peut implémenter plusieurs interfaces.Une classe peut hériter d'une classe de base et également implémenter une ou plusieurs interfaces.
- Associer au terme « contrat » ou « contrat de service ».
- Mot-clé → *interface*.

Différences entre classe abstraite et interface

Points en commun
• Ne peut pas être instanciée (opérateur *new*) • Doit être dérivée • Sert à définir des propriétés, indexeurs, méthodes et événements • Les membres imposés n'ont que leurs signatures. Donc pas de bloc { } et pas de code.

Classe abstraite	Interface
• Mot-clé *abstract* • Une classe dérivée ne peut dériver que d'une seule classe de base. • Peut contenir des membres non abstraits, c'est-à-dire avec du code implémenté. • Factoriser le code.	• Mot-clé *interface* • Une classe dérivée peut dériver de plusieurs interfaces. • Tous les membres ne peuvent contenir du code implémenté. • Ajouter des fonctionnalités (appelées « contrats »).

Exercice 11. : Dans le diagramme **ClassesEcole.cd**, ajouter une classe nommée **Communicateur** ayant une liste d'**événements** comme propriété. Ajouter une classe nommée **Voiture** avec les membres suivants : NbKmParcourus (int), DateDébut (DateTime), DateFin (DateTime), Active (bool) et Description (string). Ajouter une interface **IVoiture** qui forcera les acteurs **Directeur** et **Communicateur** à implémenter une liste de voitures empruntées durant leur emploi.

Comment choisir entre interface ou classe abstraite ?

Une classe abstraite servira à construire des classes similaires en factorisant le code. Elles auront toutes une implémentation en commun, celle de la classe abstraite.

Une interface est généralement utilisée pour définir des fonctionnalités (le « contrat ») supplémentaires, même si les classes n'ont pas grand-chose en commun.

Bibliographie

Comment ça marche .NET. (2013, 05 01). *POO - Le polymorphisme*. Consulté le 05 08, 2013, sur Comment ça marche .NET: http://www.commentcamarche.net/contents/811-poo-le-polymorphisme

Mastriani, R. (2013, 05 08). PowerPoint - Héritage et polymorphisme. Saint-Hyacinthe, QC, Canada.

MSDN. (2013, 01 01). *Polymorphisme (Guide de programmation C#)*. Consulté le 05 08, 2013, sur MSDN: http://msdn.microsoft.com/fr-fr/library/ms173152(v=vs.80).aspx

Tourreau, G. (2010). *C#: L'essentiel du code et des classes.* Paris: Pearson Education France.