Concepteur Développeur en Informatique

Développer des composants d'interface

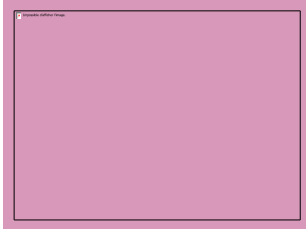
POO Héritage et Polymorphisme

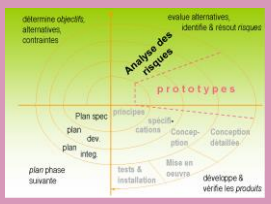
Accueil

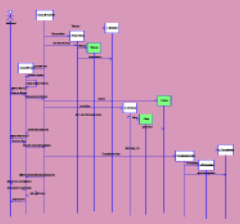
Apprentissage

PAE

Evaluation







Localisation : U03-E03-S01

SOMMAIRE

1. Introduction

Dans ce deuxième document seront mis en œuvre deux autres concepts essentiels de la programmation objet que sont l'héritage et le polymorphisme. L'héritage constitue l'un des fondements de la programmation objet après le principe d'encapsulation. La mise en œuvre de l'héritage conduit souvent à des implémentations sous plusieurs formes des opérations, introduisant ainsi le polymorphisme. Le polymorphisme, ici associé à l'héritage, existe sous d'autres formes que nous verrons ultérieurement.

1.1. Objectifs

- S'approprier les principes de l'héritage par la mise en œuvre de classes et objets dérivés.
- Adapter et étendre les comportements des objets dérivés au travers de la surcharge et la substitution de méthodes dans les classes dérivées.

2. Le concept d'héritage

L'héritage consiste en la création d'une nouvelle classe dite ***classe dérivée*** à partir d'une classe existante dite ***classe de base*** ou ***classe parente***.

L'héritage permet:

- **De récupérer** le comportement standard d'une classe d'objet (classe parente) à partir de propriétés et des méthodes définies dans celles-ci,
- **D'ajouter** des fonctionnalités supplémentaires en créant de nouvelles propriétés et méthodes dans la classe dérivée,
- **De modifier** le comportement standard d'une classe d'objet (classe parente) en surchargeant certaines méthodes de la classe parente dans la classe dérivée.

Mettre en œuvre correctement les principes de l'héritage suppose que nous ayons correctement pensé notre système de classes et ayant donc fait un effort de classification.

2.1. Représentation de l'héritage

Le concept d'héritage peut être utilisé pratiquement à l'infini.

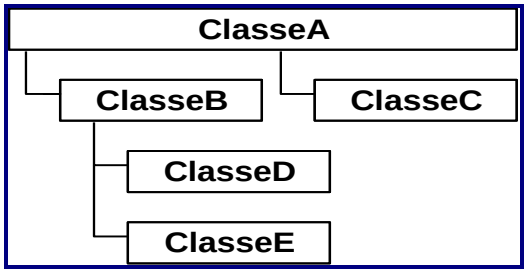
On peut créer des classes dérivées à partir de n'importe quelle autre classe (sauf avis contraire), y compris celles qui sont déjà des classes dérivées.

Supposons que **ClasseB** et **ClasseC** soient des classes dérivées de **ClasseA** et que **ClasseD** et **ClasseE** soient des classes dérivées de **ClasseB**.

Les instances de la classe **ClasseE** auront des données et des fonctions membres communes avec les instances de la classe **ClasseB**, voire de la classe **ClasseA**.

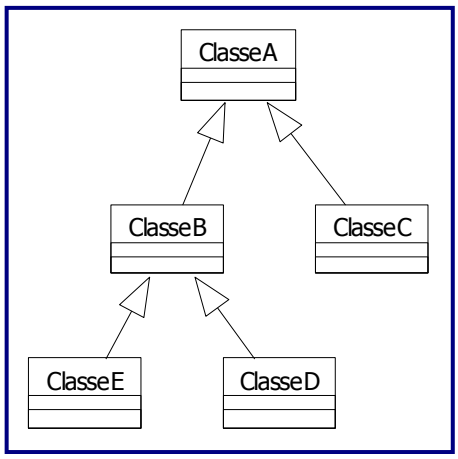
Si les dérivations sont effectuées sur plusieurs niveaux, une représentation graphique de l'organisation de ces classes facilitera la compréhension de la hiérarchie de classes.

Voici la représentation graphique de l'organisation de ces classes :



Le diagramme ci-dessus constitue la représentation graphique de la ***hiérarchie de classes*** construite à partir de **ClasseA**.

Dans le cadre de la conception orientée objet, la méthode UML (Unified Modeling Language) propose une autre représentation graphique d'une telle hiérarchie mais qui ne traduit pas autre chose :

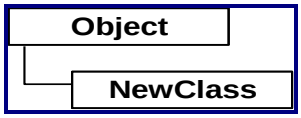


2.1.1. Insertion d'une classe dans une hiérarchie

Afin de rendre homogène le comportement des instances d'une nouvelle classe il est important de l'insérer dans une bibliothèque de classe existante.

Cette bibliothèque est fournie sous la forme d'une ***hiérarchie de classes*** construite à partir d'une ***classe racine***.

En C#, il est impossible de créer une classe isolée. En effet, lorsqu'on crée une nouvelle classe sans mentionner de classe de base, c'est la classe **Object** , la classe racine de toutes les classes C#, qui est utilisée.



NewClass est une nouvelle classe insérée dans la hiérarchie de classes construite à partir de **Object**.

Le Framework .NET propose une hiérarchie de classes normalisées prêtes à l'emploi, ou à dériver.

3. Mise en œuvre de l'héritage : Les bases

Pour hériter d'une classe de base il convient d'utiliser la syntaxe suivante :

```
class StagiaireFC : Stagiaire
{
}
```

Nous indiquons ainsi que la classe StagiaireFC hérite de la classe Stagiaire.

Par ce mécanisme la classe StagiaireFC hérite des propriétés et méthodes de la classe de niveau supérieur Stagiaire.

L'héritage de classes multiples n'est pas autorisé en C# et dans dot Net plus généralement. Ainsi, dans cet exemple, la classe StagiaireFC ne peut hériter que de la seule classe Stagiaire.

Nous verrons par la suite que cette limite peut être contournée par le recours aux interfaces mais ces mécanismes sont loin d'être identiques.

3.1. Interdiction d'hériter

Vous pouvez décider d'empêcher l'héritage d'une classe pour éviter le plus souvent des comportements de nature à créer des dysfonctionnements au niveau de votre système de classes.

L'utilisation du mot clé **sealed** interdit que votre classe puisse être une classe de base d'une classe dérivée.

```
sealed class StagiaireFC : Stagiaire
{
}
```

La classe StagiaireFC sera terminale dans la hiérarchie et ne pourra être dérivée.

De la même manière, vous pourrez restreindre la substitution d'une méthode d'une classe dérivée en la précisant comme **sealed.** Cela signifie qu'une classe ne pourra plus substituer un autre comportement que celui prévu dans cette dernière classe. Une méthode **sealed** est la dernière implémentation d'une méthode.

3.2. Obligation d'hériter

Une classe peut être codée sans autoriser la création d'objets concrets construits sur son modèle. Cette classe est alors dite **abstraite**.

Lorsqu'une classe ne correspond pas à une réalité physique mais seulement utilisée au niveau conceptuel, elle sera dite abstraite et ne pourra faire l'objet d'une instanciation.

L'utilisation du mot clé **abstract** précise que cette classe doit être héritée et qu'elle ne peut être instanciée comme telle.

Dans cet exemple, la classe stagiaire est abstraite.

Elle existe uniquement pour permettre le regroupement des propriétés et méthodes communes à tous les stagiaires.

```
abstract class Stagiaire
{

}
```

Les objets concrets seront toujours des stagiaires :

- de la formation continue créés à partir de la classe StagiaireFC
- du conseil régional créés à partir de la classe StagiaireCR.

Autre exemple au niveau de la classe **Image**.

Nous ne pouvons pas créer une instance de la classe **Image** car il n'existe pas dans la réalité des objets de type image. Seuls existent des objets constitués d'un ensemble de points **(images bitMap)** ou des objets constitués d'un ensemble de méthodes graphiques (images **vectorielles**).

```
abstract class Image
{
    // propriétés et méthodes communes
}
class ImageBitmap : Image
{
    // propriétés spécifiques
    // méthodes spécifiques et substituées
}
class ImageVectorielle : Image
{
    // propriétés spécifiques
    // méthodes spécifiques et substituées
}
```

3.3. Construction d'un objet d'une classe dérivée

Chaque constructeur d'une classe dérivée doit obligatoirement appeler le constructeur équivalent de la classe de base.

Si **BaseClass** est une classe de base appartenant à la hiérarchie de classe construite à partir de **Object**, si **NewClass** est une classe dérivée de **BaseClass**, l'instanciation d'un objet de la classe dérivée NewClass se fera en invoquant le mot clé **base** lors de la définition du constructeur.

La syntaxe utilisée ressemble à celle de l'héritage **:base()**

Observez l'image page suivante qui illustre l'invocation du constructeur de la classe de base.

```
class BaseClass
{
}
class NewClass : BaseClass
{
    public NewClass():base()
    {
    }
}
```

3.3.1. Instanciation avec constructeur par défaut

Le constructeur par défaut de la classe de base est appelé implicitement. Mais il est préférable de l'appeler explicitement.

Appel du constructeur par défaut de la classe StagiaireFC héritant de Stagiaire :

```
public sealed class StagiaireFC : Stagiaire
{
    int _x;
    public int X
    {
        get { return _x; }
        set { _x = value; }
    }


    public StagiaireFC()
        : base()
    {
    }
}
```

3.3.2. Instanciation avec constructeur d'initialisation

Le mécanisme est tout à fait similaire. Il convient d'appeler le bon constructeur de la classe de base en respectant la signature de celui-ci.

```
public StagiaireFC(string nom, DateTime dateNaissance, int x)
    : base(nom, dateNaissance)
{
    this._x = x;

}
```

4. Surcharge, substitution et polymorphisme

La classe dérivée d'une classe parente hérite des propriétés et méthodes de celle-ci.
Les méthodes dérivées doivent parfois faire l'objet d'adaptation afin de prendre en compte les spécificités des instances de la classe dérivée.

Ces modifications provoquent chez les objets créés à partir des classes dérivées des comportements différents de ceux de la classe de base : c'est ce qui définit le terme **polymorphique** qui signifie **plusieurs formes** des objets.

Deux mécanismes sont mis en œuvre : la substitution et la surcharge. S'ils présentent des similitudes, ces deux mécanismes recèlent des différences qu'il est important de connaître.

Le mécanisme de **substitution** d'une méthode de la classe de base par celle de la classe dérivée repose sur la redéfinition du code de la méthode en en conservant la signature.

Le mécanisme de **surcharge** consiste, quant à lui, à déclarer **plusieurs versions** d'une méthode, chaque version ayant sa propre signature. La signature d'une méthode se compose du nom de la méthode ainsi que du type et du genre (valeur, référence ou sortie) de chacun de ses paramètres formels, considérés de gauche à droite.

La surcharge n'est pas un mécanisme propre à la dérivation. Vous pouvez l'avoir déjà mis en œuvre préalablement.

Il peut être **obligatoire** d'envisager de substituer au comportement initial d'un class parente un autre comportement dans la classe dérivée.

Plusieurs mots clés sont mobilisés dans le cadre de la substitution et surcharge de méthodes.

- Une méthode définie avec le mot clé **override** est une autre implémentation d'une méthode
- Une méthode définie avec le mot clé **new** est une nouvelle méthode qui masque la méthode d'origine de la classe de base. Elle a la même signature que la méthode de la classe de base.
- Une méthode **sealed** est la dernière implémentation d'une méthode.

4.1. Substitution de méthodes

Deux exemples courants qui illustrent la nécessité de substituer dans la classe dérivée les méthodes de la classe de base :

- La substitution de la méthode ToString() de la classe Object
- La substitution de la méthode Equals() de la classe Object

Ces méthodes de base ne rendent pas le service attendu dans la classe dérivée.

4.1.1. Substitution de la méthode ToString()

Dans le contexte de notre classe Entité métier, cette méthode doit retourner une chaîne de caractères (instance de la classe **String**) qui représente les propriétés des instances de la classe. Par exemple, pour la classe **Stagiaire**, la chaîne de caractères représentant un stagiaire devrait renvoyer son identifiant, son nom, son prénom, et sa date de naissance.

```
/// <summary>
/// Chaine représentant l'objet instancié
/// </summary>
/// <returns>Valeurs des propriétés de l'objet</returns>
public override string ToString()
{
    return string.Format(@"{0}/{1}", this._nom,
    this._dateNaissance);
}
```

Remarquez l'usage du modificateur **override.**

Ce mot clé désigne que la méthode ou la propriété d'une classe dérivée se **substitue** à la méthode ou la propriété de la classe de base portant la même signature. Il ne peut être fait recours à **override** que si la méthode de base existe et autorise la substitution.
La méthode de base, qui doit être substituée, peut être déclarée comme **virtual**, **abstract** ou **override** (dans les classes dérivées uniquement) pour indiquer qu'elle est substituable.
Une méthode déclarée **abstract** n'est pas implémentée. Nous ne disposons que de sa signature et elle devra être implémentée dans la classe dérivée.

Qu'il s'agisse de la méthode ou propriété substituée ou de la méthode ou propriété de substitution, elles doivent avoir les mêmes modificateurs de niveau d'accès.

Remarquez l'utilisation de l'arrobase pour éviter l'interprétation des caractères d'échappement. Particulièrement utile lorsque l'on précise des chemins fichiers de dossiers ou fichiers.

4.1.2. Substitution de la méthode Equals()

Cette méthode doit retourner **vrai** si deux instances sont rigoureusement égales.
Deux conditions sont à vérifier :

- Il faut que les deux objets soient de la même classe. Le paramètre de la méthode **Equals** étant de type **Object** dans le premier cas, notre instance peut donc être comparée à une instance d'une classe quelconque dérivée de **Object**.

- Il faut qu'une règle d'égalité soit appliquée. Par exemple, deux objets seront égaux s'ils sont de même type et si les propriétés qui les identifient sont de même valeurs.

La méthode Equals substituée à celle de la classe de base, aura une **signature identique** :

```
    /// <summary>
    /// Chaine représentant l'objet instancié
    /// </summary>
    /// <returns>Valeurs des propriétés de l'objet</returns>
    public override string ToString()
    {
        return string.Format(@"{0}/{1}", this._nom,
        this._dateNaissance);
    }
    /// <summary>
    /// Détermine si un objet équivaut au stagiaire considéré
    /// </summary>
    /// <param name="st">Objet</param>
    /// <returns>Vrai si égaux</returns>
    public override bool Equals(object obj)
    {
        Stagiaire stStagiaire = obj as Stagiaire;
        if (stStagiaire == null) return false;
        if (this._dateNaissance == stStagiaire._dateNaissance &&
           this._nom == stStagiaire._nom) return true;
        else  return false;

    }
```

Autre exemple de substitution :

```
public class MaBase
{
    public virtual int Mult()
    {
        return 1 * 2;
    }
}
public class HeriteA:MaBase
{
    public override int Mult()
    {
        return 1 * 3;
    }
}
```

Résultat de :

```
MaBase b = new MaBase();
Console.WriteLine("Méthode appelée sur instance de base : {0}",b.Mult());
MaBase bh = new HeriteA();
Console.WriteLine("Méthode appelée sur instance Heritier déclarée en type parent  : {0}", bh.Mult());
HeriteA h = new HeriteA();
Console.WriteLine("Méthode appelée sur instance Heritier déclarée en type héritier : {0}", h.Mult());
Console.ReadLine();
```

```
Méthode appelée sur instance de base : 2
Méthode appelée sur instance Heritier déclarée en type parent  : 3
Méthode appelée sur instance Heritier déclarée en type héritier : 3
```

Observez bien l'exemple ci-dessus. Il s'agit d'une illustration du polymorphisme d'objets. Dans le cadre d'une méthode **virtual** substituée dans une classe héritée, le choix de la méthode retenue est réalisé lors de l'exécution en fonction de la nature de l'objet.

Bh déclarée du type du parent (Mabase) contient un objet de type enfant (HeriteA), c'est la méthode de l'enfant qui est invoquée puisqu'elle existe. C'est le fonctionnement classique de la substitution : la méthode invoquée est toujours celle de l'objet dérivé si elle existe.

4.2. Remplacement d'une méthode

Il est possible de définir une **nouvelle méthode** dans une classe dérivée en remplacement d'une méthode de la classe de base en recourant au mot clé **new.**

Ce mécanisme est nécessaire notamment lorsque la méthode de base n'est pas marquée comme virtuelle.

```
public class MaBase2
{
    public  int Mult()
    {
        return 1 * 2;
    }
}
public class HeriteA2 : MaBase2
{
    public new int Mult()
    {
        return 1 * 3;
    }
}
```

Reprenons l'exemple précédent adapté au nouveau contexte. Résultat de :

```
MaBase2 b2 = new MaBase2();
Console.WriteLine("Méthode appelée sur instance de base : {0}", b2.Mult());
MaBase2 bh2 = new HeriteA2();
Console.WriteLine("Méthode appelée sur instance Heritier déclarée en type parent  : {0}", bh2.Mult());
HeriteA2 h2 = new HeriteA2();
Console.WriteLine("Méthode appelée sur instance Heritier déclarée en type héritier : {0}", h2.Mult());
Console.ReadLine();
```

```
Méthode appelée sur instance de base : 2
Méthode appelée sur instance Heritier déclarée en type parent  : 2
Méthode appelée sur instance Heritier déclarée en type héritier : 3
```

Dans ce contexte c'est la méthode de la classe de base qui est invoquée lorsque l'objet est stocké dans une variable de type Parent.
Pour obtenir la méthode fille, il conviendrait de réaliser une conversion de l'objet dans le type Enfant comme ci-dessous :

```
Console.WriteLine("Méthode appelée sur instance Heritier déclarée en type parent  : {0}",
    ((HeriteA2)bh2).Mult());
```

Attention donc aux phénomènes induits par ces différents choix.

4.3. Surcharge de méthodes

On peut aussi écrire une implémentation supplémentaire de la méthode avec une **signature différente**. Il s'agit alors ici d'une surcharge. Ici, une nouvelle implémentation de la méthode Equals avec une signature différente : le type passé en argument.

```
/// <summary>
/// Détermine si deux stagiaires sont identiques
/// </summary>
/// <param name="st">Objet Stagiaire</param>
/// <returns>Vrai si égaux</returns>
public bool Equals(Stagiaire st)
{
    if (this._dateNaissance == st._dateNaissance &&
      this._nom == st._nom) return true;
    else return false;
}
```

L'exemple suivant illustre bien le comportement polymorphe induit par la surcharge d'une méthode. Les deux méthodes ont des fonctions similaires mais opère avec des arguments différents.

```
public int Addition(int nombreA, int nombreB)
{
    return nombreA + nombreB;
}
public float Addition(float nombreA, float nombreB)
{
    return nombreA + nombreB;
}
```

Nous pourrions créer une méthode Addition avec des chaînes. La concaténation est un exemple d'Addition particulière.

4.4. Appel aux méthodes de la classe de base

Lors de la surcharge d'une méthode de la classe de base dans une classe dérivée, il peut être utile de reprendre les fonctionnalités standards pour y ajouter les nouvelles fonctionnalités des méthodes de la classe dérivée.

Pour ne pas avoir à réécrire le code de la classe de base (dont on ne dispose pas forcément), il est naturel de faire appel à la méthode de la classe de base.

Pour cela nous devons recourir à nouveau au mot-clé **base**.

```
public override void Imprimer()
{
    base.Imprimer();
    // Complément
}
```

Ainsi, la méthode ImprimerInformations() de la classe dérivée complète la définition de la méthode ImprimerInformations() de la classe dont elle hérite.

5. Polymorphisme

Nous venons d'introduire dans ce document quelques exemples illustrant le comportement polymorphe de nos objets induit par :

- la substitution de méthodes au sein d'une classe fille (héritage)
- le remplacement de méthodes au sein d'une classe fille (héritage)
- la surcharge de méthodes

Le polymorphisme permet donc à un même code d'être utilisé avec différents types. En utilisant un même nom de méthode pour plusieurs types d'objets différents, le polymorphisme permet une programmation beaucoup plus générique.

Nous pouvons prendre l'exemple de la méthode « Calculer Intérêts » qui peut être définie pour tout compte bancaire mais implémentée différemment en fonction de la nature des comptes (courants, épargne, logement, …).
Le développeur de l'application n'a pas à connaître précisément comment est effectué le calcul les intérêts mais doit s'assurer que la méthode est implémentée.