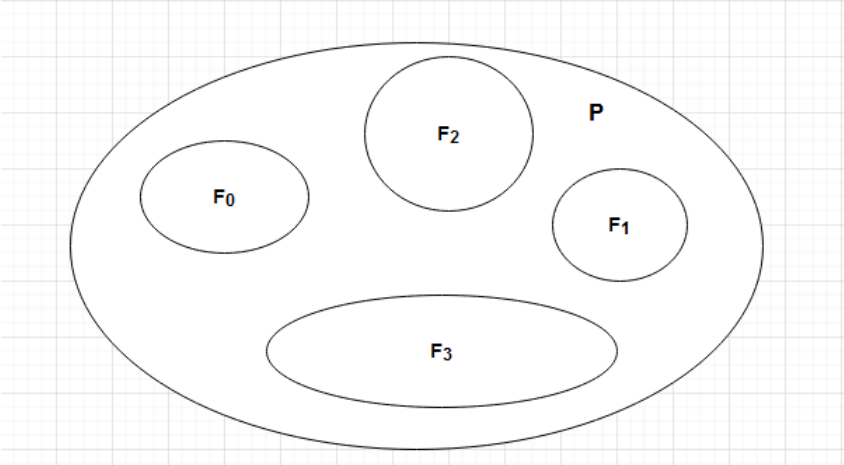
1 Module B.4 : Polymorphisme

1.1 Définition

Le polymorphisme signifie « qui peut prendre plusieurs formes » et est l'une des conséquences les plus fondamentales des principes de la programmation orientée objet. L'héritage et la hiérarchie des classes établissent le lien entre les différents objets. Pour rappel si A et B sont deux objets et que B hérite de A, alors une instance de B *est* une instance de A. La réciproque n'est pas explicitement vraie. De plus si I est une interface et C un objet implémentant I, alors C *est capable de* I.

Pour commencer, il existe plusieurs formes de polymorphisme. Le polymorphisme *ad hoc* a déjà été abordé et désigne la capacité de surcharger les méthodes d'une classe. Potentiellement, une même méthode désignée par son nom peut disposer de plusieurs implémentations si ces signatures sont différentes. De plus, il existe le polymorphisme *paramétrique* qui désigne la capacité de plusieurs objets à permettre un choix de type qui les caractérisent. Typiquement, c'est le cas des collections qui ont été étudiées dans la première partie où le type des éléments est à spécifier (ou le couple de types dans le cadre des dictionnaires). Le polymorphisme *paramétrique* est parmi en C# par l'introduction des types *génériques*. La programmation générique dépasse très largement le cadre de cette modeste formation. Le dernier type de polymorphisme est le polymorphisme *d'héritage* et c'est celui que ce module approfondit.

Le polymorphisme *d'héritage* s'appuie sur le concept d'héritage. Une classe *fille* dispose de l'ensemble des propriétés et méthodes de sa classe *parente*. De ce fait, une instance de la classe *fille* est par définition une instance de la classe *parente*. L'ensemble des instances de la classe fille est inclus dans celui des instances de la classe *parente*. On désigne la classe *parente* par un P et ses classes *filles* potentielles par l'ensemble $\{F_k, k \in \mathbb{N}^+\}$.



Une variable de type P peut référencer une instance de type F ou dérivée de F. Les canaux de manipulation de cette variable sont ceux disponibles dans P.
La version des méthodes et des propriétés de P utilisée lors d'un appel d'une de celles-ci respecte les règles suivantes :

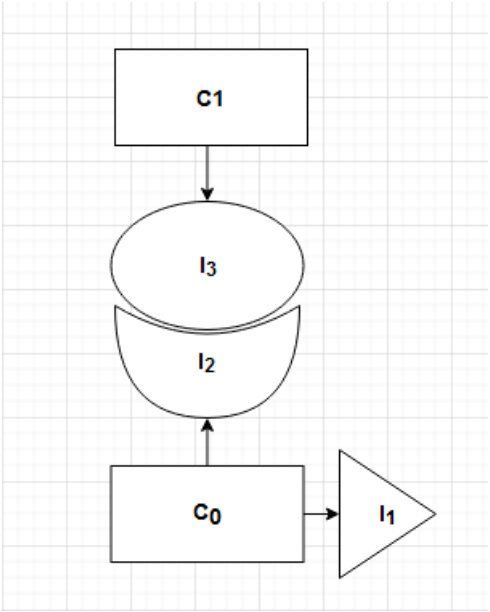
— Si l'instance est de type P, alors la version appelée est celle du type P.

— Si l'instance est de type F_k et la méthode est *substituée*, alors la version appelée est celle du type F_k.

— Si l'instance est de type F_k et la méthode est *remplacée*, alors la version appelée est celle du type P.

La *substitution* des méthodes permet de changer la version de la méthode appelée sans connaître d'avance le type spécifique de l'instance. Cette réalité permet d'écrire des traitements sur des instances incluses dans un type primitif sans se soucier des détails de l'implémentation de la méthode considérée.

Par exemple, en reprenant notre classe **Voiture** qui fournit des méthodes **Accélère**, **Freine**, **Change Direction**. On définit une collection de **Voitures** en ajoutant des instances du type **Voiture** et des types dérivés de **Voiture** (comme **Voiture Sportive**, **Voiture Sans Permis**, **Limousine**). Les trois méthodes ont des implémentations qui dépendent du type de l'instance sélectionné. Une voiture de sport accélère et freine beaucoup plus rapidement qu'une voiture sans permis ou qu'une voiture simple, mais consomme beaucoup plus de carburants. L'optimisation de son utilisation est beaucoup plus primordiale qu'avec les autres types. Une limousine change de direction avec plus de difficulté.
Ces détails sont sans importance au niveau de notre collection de **Voitures**. Si cette collection représente des voitures dans un circuit automobile, on peut se concentrer sur l'interaction entre les différentes voitures, la gestion des trajectoires et du classement.



Une autre manière d'utiliser le polymorphisme *d'héritage* est de passer par les interfaces. L'héritage est pratique pour faire du polymorphisme, mais on est limité par le fait que le type primitif doit être concret. C'est suffisant dans de nombreuses situations comme dans l'exemple précédent. Le polymorphisme fonctionne de la même façon avec les interfaces. On définit une variable avec le type de l'interface I et le traitement choisira d'appeler l'implémentation de la méthode du type de l'instance référencée par la variable de type I. Les interfaces permettent de se concentrer sur les relations entre les objets par la contrainte sur le type d'objets à utiliser.

Par exemple, dans le monde réel, il existe ce que l'on appelle des pilotes informatiques ou « drivers » qui ont pour objectif de permettre à un système d'exploitation d'interagir avec un type de périphérique comme une imprimante, un clavier, une souris... Cet intermédiaire permet d'éviter d'avoir à prévoir toutes les possibilités de type de clavier, mais de fournir une interface standard d'interaction.
Le parallèle peut se faire avec le réseau électrique et les adaptateurs électriques permettant de fournir le bon courant avec la bonne intensité et tension aux différents appareils électriques. Le principe est

le même dans le monde des logiciels.

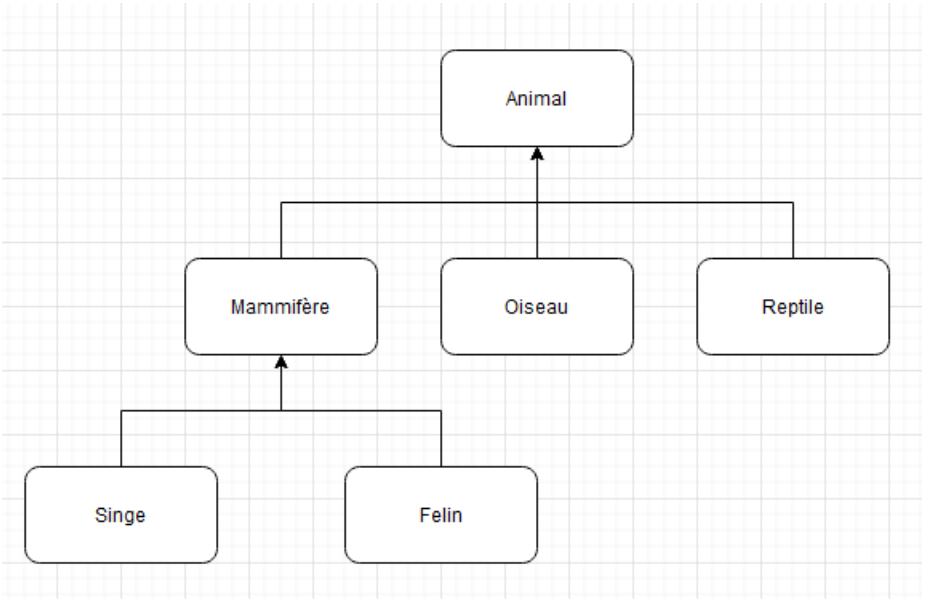
1.2 Prise en compte en C#

1.2.1 Applications

Polymorphisme sur collections

Comme indiqué précédemment, le polymorphisme par héritage permet de ne pas se soucier des détails d'implémentation des méthodes virtuelles. La prise en compte de l'implémentation des classes *filles* se fait automatiquement ce qui permet de ne pas s'en soucier. Une utilisation courante concerne la gestion et les traitements de masse sur des collections d'instances de types hétérogènes.

L'exemple suivant est une illustration de cette utilisation. On définit la hiérarchie de classes suivante. La classe **Animal** déclare une méthode *virtuelle* **TypeAnimal** qui est redéfinie dans toutes les classes qui en héritent.



Exemple : Collections hétérogènes et polymorphisme.

```
1  // Déclaration de la hiérarchie de classes.
2  public class Animal
3  {
4    public virtual void TypeAnimal ()
5    {
6      Console.WriteLine ("Je suis un animal.");
7    }
8  }
9  public class Mammifere : Animal
10 {
11   public override void TypeAnimal ()
12   {
13     Console.WriteLine ("Je suis un mammifère.");
14   }
```

```
}
...
// Utilisations dans une collection hétérogène.
List<Animal> animaux = new List<Animal>()
{
  new Mammifere(), new Reptile(), new Oiseau,
  new Singe(), new Felin()
};

foreach (var animal in animaux)
  animal.TypeAnimal();

// Le résultat est :
// Je suis un mammifère.
// Je suis un reptile.
// Je suis un oiseau.
// Je suis un singe.
// Je suis un félin.
```

Polymorphisme avec les interfaces

Pour montrer l'utilisation des interfaces dans le cadre du polymorphisme, il est important de considérer les relations entre les objets avant de s'intéresser aux objets en eux-mêmes. En reprenant l'exemple de la section sur les interfaces avec **ICompteBancaire**. Elle est encore généralisable. Les paramètres **somme** des deux méthodes peuvent être remplacés par une instance d'une interface qui représenterait une somme. Une implémentation possible revient à considérer une interface **ITransaction** représentant une transaction monétaire. Cette interface devra fournir au minimum une propriété donnant une valeur décimale.

Exemple : Relation entre les interfaces.

```
// Interface d'une transaction.
public interface ITransaction
{
  decimal Somme { get; }
}
// Interface du transfert d'argent.
public interface ICompteBancaire
{
  // Méthodes
  void Retirer (ITransaction transaction);
  void Ajouter (ITransaction transaction);
  // Propriété
  decimal Solde { get; }
}

// Implémentation de ces interfaces.
public class Transaction : ITransaction
{
  private decimal _somme;
```

```
20    public decimal Somme
21    {
22      get { return Math.Abs(_somme); }
23    }
24  }
25
26  public class CompteBancaire : ICompteBancaire
27  {
28    // Implémentation propriété.
29    public decimal Solde { get; protected set; }
30
31    // Implémentation méthodes.
32    public void Retirer (ITransaction transaction)
33    {
34      if (transaction.Somme > Solde)
35        throw new ArgumentOutOfRangeException ("Le retrait n'est pas
              possible.");
36
37      Solde -= transaction.Somme;
38    }
39
40    public void Ajouter (ITransaction transaction)
41    {
42      if (transaction.Somme < 0.0m)
43        throw new ArgumentOutOfRangeException ("L'ajout doit être positif.");
44
45      Solde += transaction.Somme;
46    }
47  }
```

1.2.2 Autour de la notion de type

Le polymorphisme est une application fort utile de l'héritage. Cela renseigne le rapport des objets et de leur type. Comme dans le cas d'une variable d'un type *parent* qui référence un objet d'une de ses classes *filles*. Il est temps de formaliser les conversions de type.

Conversions

Une référence d'un objet peut être :

— Implicitement *upcast* vers une référence d'une classe *parente*.

— Explicitement *downcast* vers une référence d'une classe *fille*.

Ces deux types de conversion ne concernent que les références des objets et non les instances des objets. Dans les conversions, une nouvelle référence est créée sur l'instance de l'objet. La conversion *upcast* correspond à la conversion utilisée dans les applications du polymorphisme. Une référence d'une classe *parente* est utilisée pour référencer une instance d'une classe *fille*. La conversion *upcast* fonctionne toujours.

Exemple : Conversion *upcast*.

```
public class Voiture
{
  public string Nom { get; set; }
  ...
}
public class Limousine : Voiture
{
  public double Taille { get; set; }
  ...
}
...
Limousine limousine = new Limousine();
Voiture voiture = limousine;              // Upcast.
// Les deux référencent la même instance.
Console.WriteLine (limousine == voiture);// true.
Console.WriteLine (voiture.Nom);          // OK.
Console.WriteLine (voiture.Taille);       // Erreur : Taille n'est pas définie.
```

La conversion *downcast* doit être explicite, car il s'agit de référencer une instance d'une classe *parente* par une variable d'une classe *fille*. Cette conversion peut échouer lors de l'exécution du programme. Lorsqu'elle échoue, une exception de type **InvalidCastException** est levée. Cette exception est une erreur de programmation que l'on ne doit pas gérer avec un système de gestion des exceptions.

Exemple : Conversion *downcast*.

```
public class VoitureSportive : Voiture
{ ... }
...
Limousine limousine = new Limousine();
Voiture voiture = limousine;          // Upcast.
Limousine limousineCopie = voiture;   // Downcast : réussi.
VoitureSportive sportive = voiture;   // Downcast : échec.
```

Opérateurs is et as

La conversion *downcast* peut échouer et lever une exception. Si l'on ne connaît pas en avance le résultat, une manière plus souple est d'utiliser l'opérateur **as**. Cet opérateur **as** réalise une conversion explicite. Si cette conversion échoue, l'expression renvoie une référence **null**. Cette référence peut être testée ensuite afin de traiter la réussite ou l'échec de la conversion.

Exemple : Opérateur **as**.

```
Voiture voiture = new Voiture();
// L'opération échoue, limousine = null.
Limousine limousine = voiture as Limousine;

if (limousine != null) // test.
  Console.WriteLine (limousine.Taille);
```

L'opérateur **is** teste si une conversion de référence réussirait. Cet opérateur **is** permettrait de vérifier

si une conversion *downcast* réussirait ou échouerait. Une expression contenant cet opérateur est une expression booléenne.

Exemple : Opérateur **is**.

```
1  Voiture voiture = new Voiture();
2
3  if (voiture is Limousine) // test : ici false.
4    Console.WriteLine (((Limousine)voiture).Taille);
```