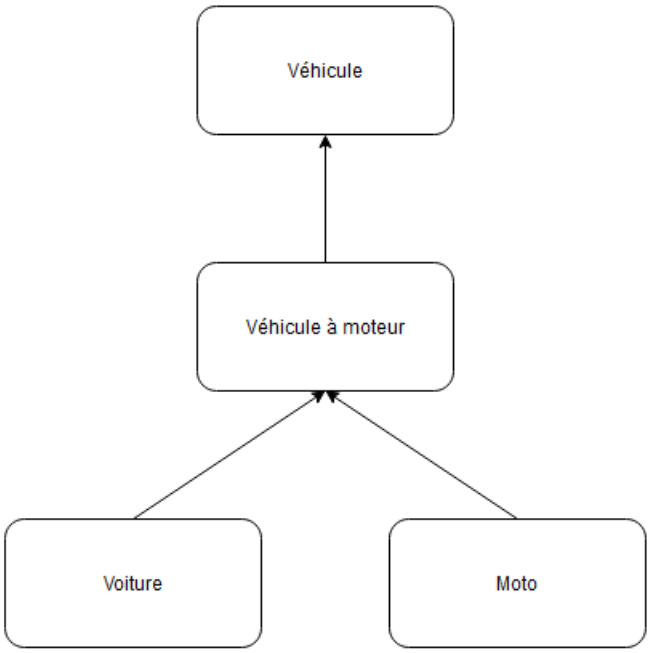
1 Module B.3 : Héritage

1.1 Définition

Jusqu'à présent, les objets définis se suffisaient à eux-mêmes. Bien entendu dans la réalité les objets peuvent posséder des caractéristiques communes et des singularités. Par exemple, une voiture et une moto sont des véhicules avec un moteur à combustion, possédant beaucoup de mécanismes de conduite en commun, comme des freins, un accélérateur, des phares et aussi des différences. Une voiture a une carrosserie, avec sa mécanique propre, la moto, non. Une moto a deux roues, la voiture quatre.
Des objets peuvent également être des extensions d'autres types d'objets, des spécialisations de concepts plus abstraits. Par exemple, la moto et la voiture sont toutes les deux des extensions du concept de véhicules avec moteur à combustion qui lui même est une extension du concept de véhicule. De cette façon, on peut facilement établir une hiérarchie entre les différents types d'objets.

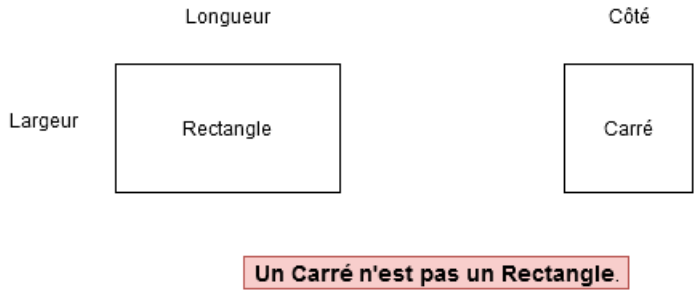
Exemple : Hiérarchie simplifiée des types de véhicules



Dans le paradigme de la programmation-objet, cette hiérarchie peut être représentée par la notion *d'héritage*. Lorsqu'un objet A hérite d'un objet B, il est un objet de type B. C'est-à-dire qu'il en possède tous les éléments (champs, propriétés et méthodes). L'objet B est une extension de l'objet A car les comportements de B peuvent être modifiés ou précisés par A et de nouveaux comportements peuvent être ajouter. L'objet **Voiture** est un objet **Véhicule à moteur**, mais il possède des caractéristiques que l'objet **Véhicule à moteur** n'a pas comme une carrosserie ou quatre roues. Le fonctionnement de la direction est précisé avec la présence d'un volant notamment. De la même façon, l'objet **Moto** hérite de l'objet **Véhicule à moteur**.

Par contre, il est important de comprendre que la hiérarchie des objets ou concepts du monde réel ne représente pas toujours celle du paradigme de l'orienté-objet. L'objet B qui hérite de l'objet A ne peut pas limiter la définition qu'apporte A, notamment mettre des contraintes sur ses attributs.
Un exemple de piège est celui des concepts géométriques. Par exemple, en mathématique un carré est un type particulier de rectangle, comme un cercle pour l'ellipse. Naïvement, on peut supposer que l'objet **Carré** hérite de **Rectangle**. Le problème de cette affirmation est que l'on impose une contrainte sur les attributs de l'objet **Rectangle** car l'on force le fait que la largeur et la longueur doivent être égales. Il est préférable de définir deux objets **Carré** et **Rectangle** séparément. L'un ayant un attribut côté et le second deux attributs longueur et largeur.

Exemple : Hiérarchie erronée en géométrie



1.2 Héritage en C#

Cette section montre l'illustration des concepts qui viennent d'être présentés dans le langage C#. Premièrement, on montre simplement l'implémentation de la hiérarchie entre les classes puis on présente la manière d'étendre ou modifier le comportement de la classe parente.

1.2.1 Implémentation de l'héritage

Définition

Pour indiquer à une classe *fille* qu'elle hérite d'une classe *parente*, on doit utiliser le caractère **:** après le nom de la classe *fille* suivi du nom de la classe *mère* dans la définition de cette classe.

```
modificateur nom Classe fille : nom Classe parente
{
// Corps de la classe fille
}
```

Une classe *fille* ne peut avoir qu'une classe *parente* car le C# interdit l'héritage multiple à l'instar de Java et contrairement au C++. On voit dans ce module comment contourner cette contrainte. Une classe *fille* peut elle-même être une classe *parente* pour de nombreuses autres classes.
Toute classe *fille* dispose de tous les éléments de sa classe *parente*. Elle peut librement déclarer de nouveaux éléments et utiliser ceux qui lui sont accessibles, à savoir ceux ayant un modificateur différent de **private**.

Exemple : Héritage

```
1  public class CompteBancaire
2  {
3    // Attribut privé accessible seulement dans le corps de CompteBancaire.
4    private decimal _solde;
5    // Propriété protégée accessible dans le corps dans le classe
6    // et en lecture seulement dans ces classes filles.
7    protected string Proprietaire { get; private set; }
8    ...
9    public void EffectuerRetrait (decimal somme)
10   { ... }
11 }
12 // Héritage de CompteBancaire.
13 public class CompteBancaireJunior : CompteBancaire
14 {
15   // Déclaration d'une méthode publique.
16   public void AfficherNom ()
17   {
18     // Accès à la propriété protégée de CompteBancaire.
19     Console.WriteLine ("Le nom du propriétaire est " + Proprietaire );
20   }
21 }
```

La déclaration d'une instance d'une classe *fille* se réalise de la même façon que pour une classe sans héritage. L'utilisateur externe de la classe *fille* peut utiliser tous les éléments de la classe (ceux définis par la classe et hérités) dont le modificateur d'accès est différent de **private** ou **protected**.

Exemple : Déclaration d'instance de classe fille.

```
1  CompteBancaireJunior compteJunior = new CompteBancaireJunior();
2  compteJunior.AfficherNom ();
```

Constructeurs

Les constructeurs des classes *parentes* sont accessibles aux classes *filles* mais ils ne sont pas hérités. Les classes *filles* doivent disposer de leurs propres constructeurs, les constructeurs de leur classe *parente* ne sont pas utilisables tels quels pour initialiser une instance d'une des classes *filles*.
Lorsque l'on déclare un constructeur d'une classe *fille*, il va appeler un constructeur de sa classe

parente. Pour faire cela, on utilise le mot clé **base** pour désigner le constructeur de la classe *parente*.

La syntaxe de la déclaration du constructeur de la classe *fille* est :

```
modificateur nom Classe (Liste paramètres classe fille) : base (Liste paramètres classe parente)
{
// Corps du constructeur de la classe fille.
}
```

Si le mot clé **base** n'est pas utilisé, le constructeur de la classe *fille* appelle le constructeur par défaut, sans paramètres de la classe *parente*. Si le constructeur sans paramètres n'est pas utilisable, la définition du constructeur de la classe *fille* n'est pas valide. Cette règle s'applique autant aux constructeurs d'instances qu'aux constructeurs statiques.

Exemple : Constructeurs et héritages

```
1  public class CompteBancaire
2  {
3    public CompteBancaire (decimal solde) { ... }
4    public CompteBancaire (string nom, decimal solde) { ... }
5    ...
6  }
7  // Héritage de CompteBancaire.
8  public class CompteBancaireJunior : CompteBancaire
9  {
10   // Constructeurs possibles.
11   public CompteBancaireJunior (string nom) : base(nom, 100.0m)
12   { ... }
13   public CompteBancaireJunior (string nom, decimal solde, int age) : base
         (solde)
14   { ... }
15   // Constructeur invalide.
16   public CompteBancaireJunior (string nom, decimal solde, int age)
17   { ... }
18 }
```

Pour présenter l'ordre d'exécution des processus de création d'une instance d'une classe *fille* :

— De la classe *fille* à la classe *parente* :

- Initialisation des champs avec la valeur par défaut de leur type.
- Initialisation des champs avec leur valeur par défaut si elle existe.
- Evaluation des arguments de l'appel au constructeur de la classe *parente*.

— De la classe *parente* à la classe *fille* :

- Exécution du corps des constructeurs.

Exemple : Ordre d'exécution.

```
1  public class CompteBancaire
2  {
3    private decimal _solde = 100m;                    // ordre 3
4    public CompteBancaire (decimal solde)
```

```
  {
    _solde = solde;                                    // ordre 4
  }
  ...
}
public class CompteBancaireJunior : CompteBancaire
{
  private int _age = 10;                               // ordre 1
  public CompteBancaireJunior (decimal solde, int age)
  : base(solde)                                        // ordre 2
  {
    _age = age;                                        // ordre 5
  }
  ...
}
```

Interdire l'héritage

On peut interdire à une classe d'être la classe *parente* d'autres classes en utilisant le modificateur **sealed** devant la définition de la classe en question.

Exemple : Interdiction de l'héritage.

```
// Classe qui peut être héritée.
public class CompteBancaire
{ ... }
// Classe qui ne peut pas être héritée.
public sealed class CompteBancaireVital : CompteBancaire
{ ... }
```

1.2.2 Les méthodes et l'héritage

Substitution des méthodes

On a vu précédemment qu'une classe *fille* héritait des méthodes de sa classe *parente*. En l'état, le fonctionnement d'une de ces méthodes sera le même si l'instance qui l'appelle est une référence à la classe *parente* ou à la classe *fille*. Ce comportement ne convient pas nécessairement à toutes les situations. Dans certains cas, le comportement de certaines méthodes doit être modifié pour correspondre à des contraintes fonctionnelles. Cette pratique consiste à *substituer* des méthodes de la classe *parente* par des implémentations de la classe *fille*.

Pour indiquer qu'une méthode peut être substituée, on doit utiliser le modificateur **virtual** dans la définition de la méthode de la classe *parente*. Cette méthode est dite *virtuelle*. Pour substituer une méthode dans une classe fille, il faut utiliser le modificateur **override** dans la définition de la méthode de substitution. Seules les méthodes avec la même signature qu'une méthode virtuelle peuvent être substituées.

Exemple : Substitution de méthodes

```
public class CompteBancaire
{
```

```
  ...
  // Méthode virtuelle.
  public virtual void RetirerSolde (decimal somme) // (decimal)
  {
    if (somme > _solde)
      throw new ArgumentOutOfRangeException ("Le retrait n'est pas
          possible.");

    _solde -= somme;
  }
}
public class CompteBancaireJunior : CompteBancaire
{
  private int _age;
  private const decimal _palier = 10m;
  ...
  // Substitution de la méthode.
  public override void RetirerSolde (decimal somme)
  {
    decimal sommeMaximale = _palier * _age;

    if (somme > sommeMaximale)
      somme = sommeMaximale;

    if (somme > _solde)
      throw new ArgumentOutOfRangeException ("Le retrait n'est pas
          possible.");

    _solde -= somme;
  }
}
```

Contrairement au Java où toutes les méthodes sont virtuelles par défaut, en C# on décide quelles sont les méthodes pouvant être substituées. Les méthodes non virtuelles sont *scellées*. Pour empêcher la substitution d'une méthode virtuelle, on doit utiliser le modificateur **sealed**. L'utilisation de **sealed** devant une méthode est moins restrictive que devant une classe, car la classe peut être héritée avec des restrictions sur son utilisation.

Exemple : Méthode scellée

```
public class Compte
{
  public virtual void MethodeImportante() { ... }
  public virtual void MethodePasImportante() { ... }
}
public class CompteImportant : Compte
{
  // Méthode scellée.
  sealed public override void MethodeImportante() { ... }
  // Méthode substituable.
```

```
  public override void MethodePasImportante() { ... }
}
public class ComptePirate : CompteImportant
{
  // Possible de substituer.
  public override void MethodePasImportante() { ... }
  // Pas possible.
  /*public override void MethodeImportante() { ... }*/
}
```

Appel de la méthode de base

On a vu dans l'exemple des comptes, la substitution de la méthode **RetirerSolde** avec une redondance de code. Pour éviter de dupliquer du code, l'appel de la méthode substituée s'obtient avec le mot clé **base**.

Exemple : Substitution de méthodes

```
public class CompteBancaire
{
  ...
  // Méthode virtuelle
  public virtual void RetirerSolde (decimal somme) // (decimal)
  {
    if (somme > _solde)
      throw new ArgumentOutOfRangeException ("Le retrait n'est pas
          possible.");

    _solde -= somme;
  }
}
public class CompteBancaireJunior : CompteBancaire
{
  ...
  // Substitution de la méthode
  public override void RetirerSolde (decimal somme)
  {
    decimal sommeMaximale = _palier * _age;
    if (somme > sommeMaximale)
      somme = sommeMaximale;
    base.RetirerSolde(somme); // Appel de la méthode à substituer.
  }
}
```

Le mot clé **base** permet d'accéder à n'importe quel élément de la classe de base accessible dans la classe *fille*.

Remplacement des méthodes

Dans n'importe quelle classe *fille*, un élément de cette classe (champ, propriété, méthodes ...) peut porter le même nom, avoir la même signature dans le cas d'une méthode, qu'un élément de la classe *parente*.

Dans ce cas, on parle de remplacement de l'élément en le cachant. Cela provoque un avertissement de la part du compilateur, car cette situation correspond en général à une erreur. Si le remplacement est intentionnel, il est conseillé d'utiliser le modificateur **new** devant l'élément masquant dans la classe *fille*. Le mot clé **new** ne fait que supprimer l'avertissement.

Exemple : Substitution de méthodes.

```
1  // Le fait que la méthode est virtuelle ne change rien.
2  public class CompteBancaireJuniorBis : CompteBancaire
3  {
4    ...
5    // Remplacement de la méthode.
6    public new void RetirerSolde (decimal somme)
7    {
8      decimal sommeMaximale = _palier * _age;
9      if (somme > sommeMaximale)
10       somme = sommeMaximale;
11     base.RetirerSolde(somme); // Appel de la méthode de base.
12   }
13 }
```

La différence entre les méthodes substituées et les méthodes remplacées se trouvent au niveau du choix de la méthode utilisée lors de son appel par un utilisateur. Pour faire simple pour l'instant (le sujet est traité dans le chapitre suivant), une variable d'une classe donnée peut référencer une instance de cette classe ou de n'importe quelle instance d'une classe *fille* de cette classe.
Dans le cas d'une méthode substituée, la méthode appelée correspond à celle de la classe de l'instance alors que dans le cas d'une méthode remplacée, il s'agit de celle de la variable.

Exemple : Différence substitution et remplacement.

```
1  // Méthode substituée.
2  CompteBancaireJunior junior = new CompteBancaireJunior (200, 10);
3  CompteBancaire compte1 = junior;
4  junior.RetirerSolde (20);  // CompteBancaireJunior.RetirerSolde.
5  compte1.RetirerSolde (20); // CompteBancaireJunior.RetirerSolde.
6
7  // Méthode remplacée.
8  CompteBancaireJuniorBis juniorBis = new CompteBancaireJuniorBis (200, 10);
9  CompteBancaire compte2 = juniorBis;
10 junior.RetirerSolde (20);  // CompteBancaireJunior.RetirerSolde.
11 compte2.RetirerSolde (20); // CompteBancaire.RetirerSolde.
```