

Chapitre 10

Notion d'héritage

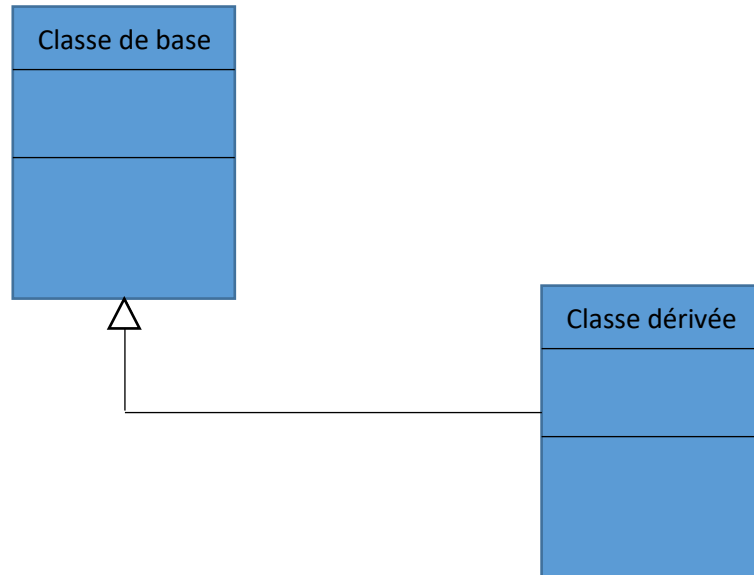
Table des matières

Introduction.....	2
I. Héritage en C#	2
II. Constructeurs des classes dérivées	3
III. Redéfinition et masquage.....	8
1. Redéfinition de méthode	8
2. Masquage de méthode.....	9
IV. Peut-on interdire l'héritage ?	10
V. Héritage et Modificateur de visibilité.....	11
VI. Exercice.....	12
VII. La classe Object	14
1. La méthode ToString :	15
2. La méthode Equals :	16
3. La méthode GetType :	18

Introduction

Un des mécanismes fondamentaux de la programmation orientée objet est : Héritage. L'héritage permet de définir une relation de **spécialisation** entre une **classe de base** (classe mère) et une **classe dérivée** (classe fille). Cette relation implique le passage des propriétés et des comportements d'une classe de base vers la ou les classes dérivées. La classe dérivée peut conserver ces propriétés et comportements ou les modifier.

La relation d'héritage est représentée de la façon suivante :



I. Héritage en C#

Pour déclarer une relation d'héritage entre deux classes on utilise la syntaxe suivante :

```
[modificateurs] class ClasseDerivee : ClasseBase
```

Il est très important de bien identifier la classe de base et la classe dérivée.

Pour identifier la relation d'héritage, essayer avec : **est un** ou **est une**.

- Un rectangle **est une** forme. Rectangle est la classe dérivée et forme est la classe de base.
- Une voiture **est un** moyen de transport. Voiture est la classe dérivée et MoyenTransport est la classe de base.

Exemple :

```
public class Forme {  
    //Définition des coordonnées du centre de gravité  
    private int x, y;
```

```

    public int X {
        get { return x; }
        set { x = value; }
    }

    public int Y
    {
        get { return y; }
        set { y = value; }
    }

    public void AfficherCentreGravite() {
        Console.WriteLine("Centre de gravité : ({0},{1})", X,Y);
    }
}

public class Rectangle : Forme {}

```

À ce stade, la classe Rectangle a hérité de toutes les propriétés de la classe Forme. En d'autres termes, un rectangle possède un centre de gravité avec les coordonnées X et Y qui proviennent de la classe Forme. La classe rectangle possède aussi une méthode nommée AfficherCentreGravite pour afficher les coordonnées du centre de gravité.

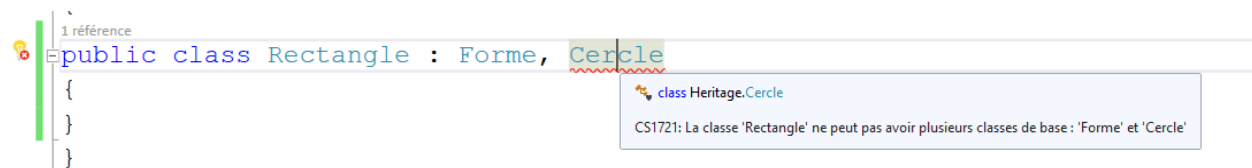
```

class Program {
    static void Main(string[] args) {
        Rectangle r = new Rectangle();
        r.X = 10;
        r.Y = 6;
        r.AfficherCentreGravite();
    }
}

```

Centre de gravité : (10,6)

Il est important de signaler qu'une classe en C# ne peut hériter que d'une seule classe au maximum (Il n'y a pas d'héritage multiple comme le langage C++). L'exemple suivant implique une erreur de compilation :



```

1 référence
public class Rectangle : Forme, Cercle
{
}

```

class Heritage.Cercle

CS1721: La classe 'Rectangle' ne peut pas avoir plusieurs classes de base : 'Forme' et 'Cercle'

II. Constructeurs des classes dérivées

Lors de la définition de la classe dérivée, il faut prêter une grande importance à la définition des constructeurs. Le constructeur de la classe dérivé doit faire appel au constructeur de la classe de base en utilisant le mot clé **base**.

Si aucun appel n'est fourni, alors un appel au constructeur par **défaut** de la classe de base est effectué. Dans le cas où la classe de base ne possède pas de constructeur par défaut alors une erreur de compilation se produit.

La syntaxe de création de constructeur est comme suit :

```
[modificateurs] ClasseDerivee ([argCD,...]) : base ([argCM,...])  
{...}
```

Exemple :

```
public class Forme {  
    //Définition des coordonnées du centre de gravité  
    private int x, y;  
  
    public int X {  
        get { return x; }  
        set { x = value; }  
    }  
  
    public int Y {  
        get { return y; }  
        set { y = value; }  
    }  
  
    //Définition d'un constructeur  
    public Forme(int x, int y) {  
        this.X = x;  
        this.Y = y;  
    }  
  
    public void AfficherCentreGravite() {  
        Console.WriteLine("Centre de gravité : ({0},{1})", X, Y);  
    }  
}
```

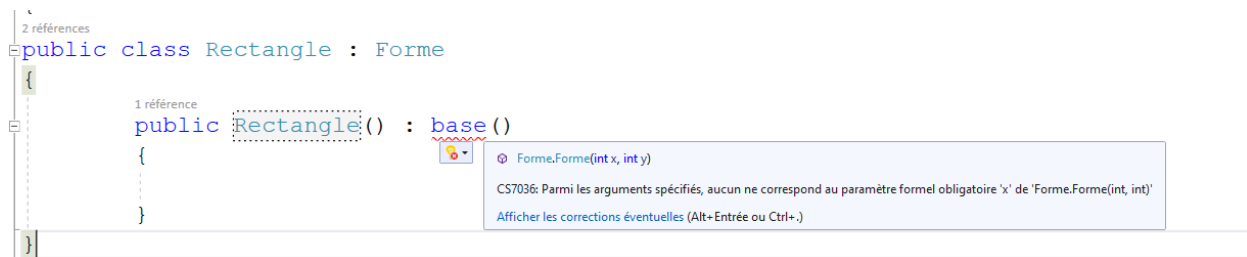
En laissant la classe **Rectangle** sans modification nous aurons une erreur de compilation.

```
1 référence  
public class Rectangle : Forme  
{  
}  
}
```

class Heritage.Rectangle

CS7036: Parmi les arguments spécifiés, aucun ne correspond au paramètre formel obligatoire 'x' de 'Forme.Forme(int, int)'

Le fait de laisser la classe Rectangle sans constructeur implique que le compilateur va lui générer un constructeur par défaut. Ce constructeur par défaut fait appel à son tour au constructeur par défaut de la classe Forme, qui n'existe pas, d'où l'erreur de compilation.



Pour corriger cette erreur, il faut prendre en considération, dans la construction d'un objet de la classe dérivée, la façon dont l'objet de la classe mère est construit.

Exemple :

```

public class Rectangle : Forme
{
    private int longueur, largeur;
    public int Longueur
    {
        get { return longueur; }
        set { longueur = value; }
    }
    public int Largeur
    {
        get { return largeur; }
        set { largeur = value; }
    }

    public Rectangle(int x, int y, int longueur, int largeur) :
        base(x, y)
    {
        this.Longueur = longueur;
        this.Largeur = largeur;
    }
}

```

Définition des attributs et propriétés de l'objet rectangle. Ces caractéristiques sont propres à un rectangle.

Définition du constructeur de la classe Rectangle. On doit initialiser les coordonnées du centre de gravité, provenant de la classe mère (base(x,y)), ainsi que la longueur et la largeur propre à la classe Rectangle.

Exemple : un étudiant est une personne. Une personne est caractérisée par son nom et son prénom. Un étudiant est caractérisé par son nom, son prénom et son numéro d'étudiant.

```

class Personne {
    private string nom, prenom;
    public string Nom {
        get { return nom; }
        set { nom = value; }
    }
}

```

```

    }

    public string Prenom {
        get { return prenom; }
        set { prenom = value; }
    }

    public Personne(string nom) { this.Nom = nom; }
    public Personne(string nom, string prenom): this(nom)
    { this.Prenom = prenom; }
}

class Etudiant : Personne {
    private int numEtudiant;

    public int NumEtudiant{
        get { return numEtudiant; }
        set { numEtudiant = value; }
    }

    public Etudiant(string nom, int numEtudiant) : base(nom) {
        this.NumEtudiant = numEtudiant;
    }

    public Etudiant(string nom, string prenom, int numEtudiant) :
        base(nom, prenom) {
        this.NumEtudiant = numEtudiant;
    }
}

```

Ou bien :

```

class Etudiant : Personne {
    private int numEtudiant;

    public int NumEtudiant {
        get { return numEtudiant; }
        set { numEtudiant = value; }
    }

    public Etudiant(string nom, int numEtudiant) :
        this(nom, null, numEtudiant) { }

    public Etudiant(string nom, string prenom, int numEtudiant) :
        base(nom, prenom) {
        this.NumEtudiant = numEtudiant;
    }
}

```

Appel au constructeur courant
(classe Etudiant) avec this.

```
}  
}
```

Le mot-clé **this** renvoie l'instance courante de la classe dans laquelle il est utilisé.
Le mot-clé **base** renvoie une référence vers un objet de la classe de base.
Le mot-clé **base** n'est pas utilisable seul.

Exemple : définir la méthode *AfficherPersonne* dans la classe **Personne** permettant d'afficher le nom et le prénom.

Définir la méthode *AfficherEtudiant* de la classe **Etudiant** permettant d'afficher, en plus du nom et du prénom, le numéro de l'étudiant.

```
class Personne {  
    ...  
    public void AfficherPersonne() {  
        Console.WriteLine("Nom = {0}\n{Prenom = {1}}", Nom, Prenom);  
    }  
}  
  
class Etudiant : Personne {  
    ...  
    public void AfficherEtudiant() {  
        base.AfficherPersonne();  
        Console.WriteLine("Numero Etudiant : {0}", this.NumEtudiant);  
    }  
}
```

On peut aussi choisir de garder le même nom de méthode comme présenter dans l'exemple suivant :

```
class Personne {  
    ...  
    public void Afficher () {  
        Console.WriteLine("Nom = {0}\n{Prenom = {1}}", Nom, Prenom);  
    }  
}  
  
class Etudiant : Personne {  
    ...  
    public void Afficher () {  
        base.Afficher ();  
        Console.WriteLine("Numero Etudiant : {0}", this.NumEtudiant);  
    }  
}
```

III. Redéfinition et masquage

Lors de la définition du comportement dans une classe dérivée, on est parfois amené à adapter le comportement en le modifiant ou en le changeant complètement. En C#, il existe deux façons de faire :

- La redéfinition
- Le masquage

1. Redéfinition de méthode

La redéfinition d'une méthode avec le mot clé **override** est applicable lorsque la méthode est définie dans la classe mère avec le mot-clé **virtual**. Ce mot-clé indique que la méthode a été conçue pour être redéfinie.

Exemple : Définir une classe nommée **Date**. Une date est caractérisée par le jour, le mois et l'année. Écrire une méthode nommée *Afficher* pour afficher la date au format français (jj/mm/aaaa).

Définir une seconde classe nommée **DateAnglaise**. La différence entre les classes est qu'une **DateAnglaise** **est une** **Date** et la date anglaise s'affiche sous la forme mm/jj/aaaa.

```
public class Date {
    public int JJ { get; set; }
    public int MM { get; set; }
    public int AAAA { get; set; }
    public Date(int j, int m, int a){
        JJ = j; MM = m; AAAA = a;
    }
    public virtual void Afficher(){
        Console.WriteLine(JJ+" / "+MM+" / "+ AAAA);
    }
}

public class DateAnglaise : Date {
    public DateAnglaise(int j, int m, int a) : base(j,m,a) {}

    public override void Afficher(){
        Console.WriteLine(MM + " / " + JJ + " / " + AAAA);
    }
}

class Program{
    static void Main(string[] args)
    {
        Date d = new Date(23, 2, 2000);
        Console.WriteLine("Date Française : ");
        d.Afficher();

        DateAnglaise dAng = new DateAnglaise(15, 3, 2020);
        Console.WriteLine("Date Anglaise : ");
        dAng.Afficher();
    }
}
```

Création d'une date française et affichage à l'aide de la méthode Afficher

Création d'une date Anglaise et affichage à l'aide de la méthode Afficher.


```

Date date = new DateAnglaise(20, 9, 2020);
Console.WriteLine("Date Ang enregistrée dans une date fr: ");
date.Afficher();
    }
}

```

Création d'une date Anglaise et enregistrement dans une date française. Appel de la méthode affichage.

```

Date Française : 23 / 2 / 2000
Date Anglaise : 3 / 15 / 2020
Date Anglaise enregistrée dans une date française: 9 / 20 / 2020

```

La méthode **override** doit avoir la même signature et la même visibilité que la méthode de la classe de base

Lorsqu'une méthode est déclarée avec le mot clé **virtual** elle déclenche le processus suivant :

- Au moment de son exécution, l'environnement d'exécution inspecte le type concret de l'objet appelant (date : DateAnglaise).
- L'environnement d'exécution recherche la méthode substituée Afficher dans le type concret (DateAnglaise). Si la méthode est trouvée alors il l'exécute. Sinon on remonte vers la classe mère du type concret jusqu'à trouver la méthode recherchée.

2. Masquage de méthode

Le masquage de méthode est semblable à la redéfinition dans l'implémentation mais sont différents dans l'exécution. Il permet de réécrire une méthode dans une classe dérivée, mais pour le masquage, on utilise le mot-clé **new** au lieu du mot-clé **override**.

Exemple : Nous allons reprendre le même exemple précédent en changeant **override** par **new** dans la redéfinition de la méthode *Afficher* de la classe **DateAnglaise**.

```

public class Date {
    public int JJ { get; set; }
    public int MM { get; set; }
    public int AAAA { get; set; }
    public Date(int j, int m, int a){ JJ = j;MM = m;AAAA = a;}
    public void Afficher(){
        Console.WriteLine(JJ+" / "+MM+" / "+ AAAA);
    }
}
public class DateAnglaise : Date {
    public DateAnglaise(int j, int m, int a) : base(j,m,a) {}

    public new void Afficher(){
        Console.WriteLine(MM + " / " + JJ + " / " + AAAA);
    }
}

```

```

class Program{
    static void Main(string[] args)
    {
        Date d = new Date(23, 2, 2000);
        Console.WriteLine("Date Française : ");
        d.Afficher();

        DateAnglaise dAng = new DateAnglaise(15, 3, 2020);
        Console.WriteLine("Date Anglaise : ");
        dAng.Afficher();

        Date date = new DateAnglaise(20, 9, 2020);
        Console.WriteLine("Date Ang enregistrée dans une date fr: ");
        date.Afficher();
    }
}

```

Création d'une date française et affichage à l'aide de la méthode Afficher.

Création d'une date Anglaise et affichage à l'aide de la méthode Afficher.

Création d'une date Anglaise et enregistrement dans une date française. Appel de la méthode affichage.

Console de débogage Microsoft Visual Studio

```

Date Française : 23 / 2 / 2000
Date Anglaise : 3 / 15 / 2020
Date Anglaise enregistrée dans une date française: 20 / 9 / 2020

```

Il n'est pas nécessaire que la méthode soit marquée comme **virtual** par la classe de base pour la masquer.

En utilisant l'opérateur new sur une méthode, on coupe l'arbre de recherche.

IV. Peut-on interdire l'héritage ?

Dans certains cas, il peut être intéressant de placer certaines contraintes sur une classe ou méthode pour interdire l'héritage à partir de cette classe ou la redéfinition d'une méthode. En C#, le mot-clé **sealed** permet d'arrêter l'héritage.

Exemple 1 : Nous voulons interdire l'héritage de la classe **DateAnglaise**.

```

public sealed class DateAnglaise : Date {
    public DateAnglaise(int j, int m, int a) : base(j,m,a){}

    public override void Afficher(){
        Console.WriteLine(MM + " / " + JJ + " / " + AAAA);
    }
}

```

En définissant une classe qui hérite de la classe **DateAnglaise** on obtient le message suivant.

0 références

```

class DateAng : DateAnglaise
{
}

```

class Heritage.DateAnglaise

CS0509: 'DateAng': dérivation du type sealed 'DateAnglaise' impossible
Afficher les corrections éventuelles (Alt+Entrée ou Ctrl+.)

Il est impossible de définir un des membres d'une classe scellée comme **virtual** car on ne peut pas lui fournir une implémentation dans une classe dérivée.

```
public sealed class DateAnglaise : Date
{
    2 références
    public DateAnglaise(int j, int m, int a) : base(j,m,a)
    {}

    4 références
    public virtual override void Afficher()
    {
        Console.WriteLine(MM + " / " +
    }
}
```

void DateAnglaise.Afficher()
CS0113: Un membre 'DateAnglaise.Afficher()' marqué comme override ne peut pas être marqué comme new ou virtual

Exemple 2: nous voulons interdire la redéfinition de la méthode Afficher de la classe DateAnglaise.

```
public class DateAnglaise : Date {
    public DateAnglaise(int j, int m, int a) : base(j,m,a) {}

    public sealed override void Afficher() {
        Console.WriteLine(MM + " / " + JJ + " / " + AAAA);
    }
}
```

```
1 référence
class DateAng : DateAnglaise
{
    0 références
    public DateAng(int x, int y, int z) : base(x, y, z) { }

    5 références
    public override void Afficher() { Console.WriteLine("Date"); }
}
```

void DateAng.Afficher()
CS0239: 'DateAng.Afficher()' : impossible de substituer le membre hérité 'DateAnglaise.Afficher()', car il est sealed

V. Héritage et Modificateur de visibilité

Avec l'héritage, nous faisons face à de nouveau problème d'accès aux attributs. C# propose différents modificateurs d'accès :

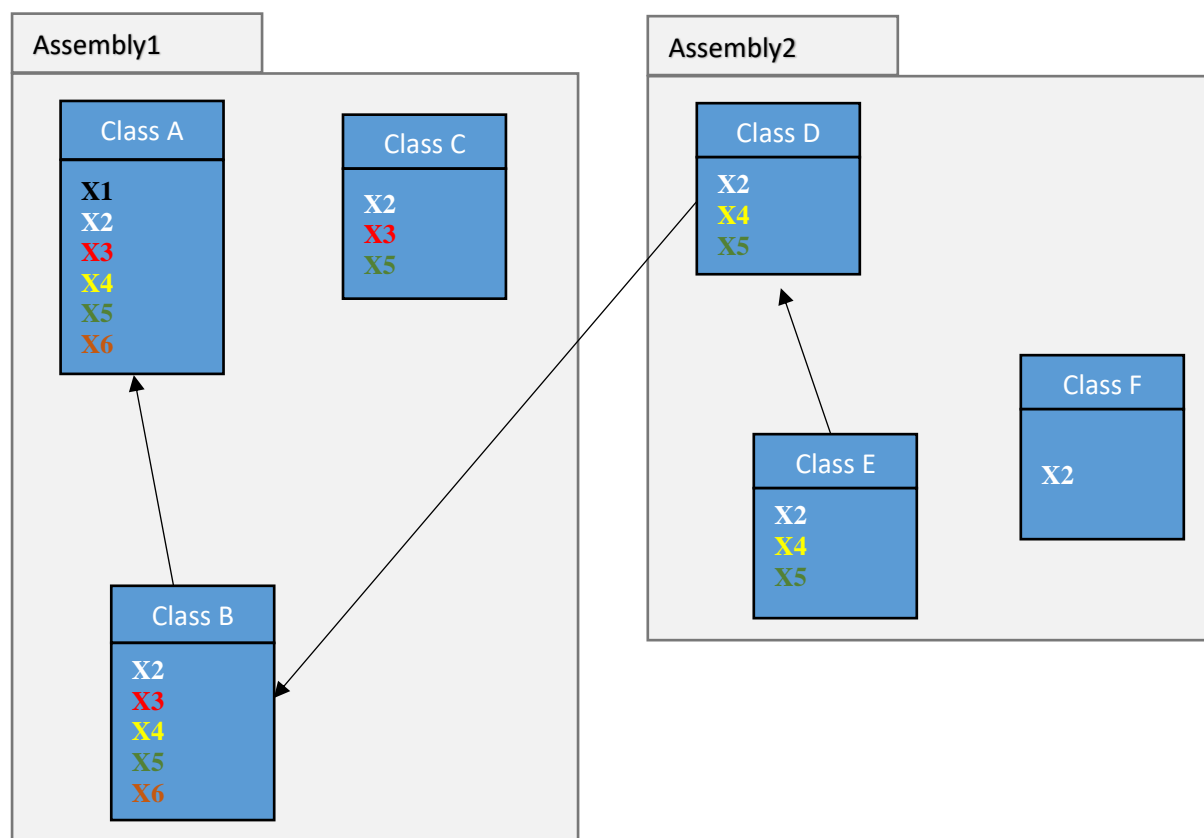
Mot-clé	Description ¹
private	L'accès est limité au type contenant.
internal	L'accès est limité à l'assembly actuel.
public	L'accès n'est pas restreint.
protected	L'accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur.

¹ <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/accessibility-levels>

protected internal	L'accès est limité à l'assembly actuel ou aux types dérivés de la classe conteneur
private protected	L'accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur dans l'assembly actuel. Disponible depuis C # 7.2.

Exemple : dans le schéma ci-dessous, nous considérons le jeu de couleur suivant pour les attributs (ou propriétés ou méthodes) :

X1 : private	X2 : public
X3 : internal	X4 : protected
X5 : protected internal	X6 : private protected



VI. Exercice

Un rectangle est caractérisé par sa longueur et sa largeur.

- Définir la classe **Rectangle** et ses attributs.
- Définir les propriétés. Implémenter la contrainte suivante : la longueur et la largeur est positif.
- Définir un constructeur ayant deux arguments représentant la longueur et la largeur.
- Définir les méthodes **Surface** et **Perimetre**.
- Définir la méthode **Afficher** permettant d'afficher un rectangle de la façon suivante :

Rectangle [longueur = 7, largeur = 4]

Un carré est un rectangle ayant sa longueur égale à sa largeur.

- Définir la classe **Carre**. Est-ce que la classe Carre possède des attributs?
- Définir un constructeur ayant un argument représentant le coté du carré.
- Doit-on redéfinir les méthodes *Surface* et *Périmètre*.
- Redéfinir la Méthode *Afficher* permettant d'afficher un carré de la façon suivante :

Carré [Côté = 5, Surface = 25 et Périmètre = 20]

Solution :

```
public class Rectangle
{
    //Définition des attributs largeur et longueur
    private double largeur = 1, longueur = 1;

    //Définition des propriétés pour les attributs largeur et longueur
    public double Largeur {
        get { return largeur; }
        set { largeur = (value > 0) ? value : 1; }
    }
    public double Longueur
    {
        get { return longueur; }
        set { longueur = (value > 0) ? value : 1; }
    }

    //Définition du constructeurs
    public Rectangle(double longueur, double largeur) {
        Longueur = longueur;
        Largeur = largeur;
    }

    /*
     * Arguments      : Aucun
     * Type de retour : double représentant la surface
     * Description    : Calcul de la surface du rectangle par la
formule suivante : longueur * largeur
    */
    public virtual double Surface() { return Largeur * Longueur; }

    /*
     * Arguments      : Aucun
     * Type de retour : double représentant le périmètre
     * Description    : Calcul du périmètre du rectangle par la
formule suivante : 2 * (longueur + largeur)
    */
}
```

```

        */
        public virtual double Perimetre() { return 2 * (Largeur +
Longueur); }

        /*
        * Arguments      : Aucun
        * Type de retour  : Pas de retour
        * Description    : Affichage d'un rectangle. La méthode est
virtual car elle va être redéfinie dans les classes filles
        */
        public virtual void Afficher() {
            Console.WriteLine("Rectangle [ Longueur : {0}\n" +
                "Largeur : {1}]", Longueur, Largeur);
        }
    }

    public class Carre : Rectangle
    {
        //Constructeur du carre faisant appel au constructeur du rectangle
        avec le mot clé base
        public Carre(int c) : base(c, c) { }
        /*
        * Arguments      : Aucun
        * Type de retour  : Pas de retour
        * Description    : Redéfinition de la méthode Affichage pour
afficher un Carre avec sa surface et son périmètre.
        */
        public override void Afficher()
        {
            Console.WriteLine("Carre [coté = {0}, Surface = {1} et
Périmètre = {2}]", Longueur, Surface(), Perimetre());
        }
    }
}

```

VII. La classe Object

Outre les types qui peuvent hériter via l'héritage simple, tous les types dans le système de types de .NET héritent implicitement de la classe **Object** ou d'un type dérivé. Les fonctionnalités communes de la classe **Object** sont disponibles pour n'importe quel type.²

Exemple :

```
public class MaClasse { }
```

² <https://docs.microsoft.com/fr-fr/dotnet/csharp/tutorials/inheritance>

L'héritage implicite à partir de la classe Object rend ces méthodes disponibles pour la classe MaClasse :

- La méthode ToString : Convertir l'objet courant vers une chaîne de caractère.
- La méthode Equals : Comparer l'objet courant avec l'objet passé comme paramètre.
- La méthode GetType : retourne un objet Type qui représente le type MaClasse.
- Et bien d'autre encore...

En raison de l'héritage implicite, vous pouvez appeler n'importe quel membre hérité d'un objet MaClasse exactement comme s'il était un membre défini dans la classe MaClasse.

1. La méthode ToString :

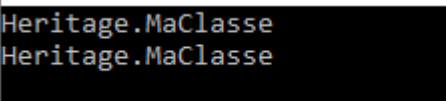
La méthode ToString convertit un objet vers sa représentation en chaîne de caractère. Par défaut cette méthode retourne le nom de type complet. En redéfinissant la méthode, vous pouvez lui donner la représentation que vous voulez.

L'avantage d'utiliser ToString est qu'on obtient directement la représentation de l'objet sans faire appel à la méthode.

Exemple : Sans redéfinition de la méthode ToString

```
namespace Heritage
{
    public class MaClasse
    {}

    public class TestMaClasse
    {
        public static void Main(String[] args)
        {
            MaClasse c1 = new MaClasse();
            Console.WriteLine(c1);
            Console.WriteLine(c1.ToString());
        }
    }
}
```



Exemple : Avec redéfinition de la méthode ToString

```
namespace Heritage
{
    public class MaClasse
    {
        public override string ToString()
        {
            return "Voici ma représentation de la classe MaClasse";
        }
    }
}
```

```

    }

    public class TestMaClasse
    {
        public static void Main(String[] args)
        {
            MaClasse c1 = new MaClasse();
            //afficher l'objet fait appel implicitement à la méthode
            //ToString
            Console.WriteLine(c1);
            Console.WriteLine(c1.ToString());
        }
    }
}

```

Voici ma représentation de la classe MaClasse
Voici ma représentation de la classe MaClasse

2. La méthode Equals :

La méthode Equals de la classe Object permet de définir un critère d'égalité entre l'objet courant et l'objet passé en paramètre. Par défaut, cette méthode teste l'égalité des références. Autrement dit, pour être égales, deux variables d'objet doivent faire référence au même objet.

Exemple :

```

namespace Heritage
{
    public class MaClasse
    {
        public int P { get; set; }
    }

    public class TestMaClasse
    {
        public static void Main(String[] args)
        {
            MaClasse c1 = new MaClasse();
            c1.P = 5;
            MaClasse c2 = new MaClasse();
            c2.P = 5;
            MaClasse c3 = c1;

            //Sans redéfinir la méthode Equals, on compare les
            //références
            //c1 et c2 pointent sur des références différentes.
            Console.WriteLine("c1 == c2 = {0}", c1 == c2);
        }
    }
}

```



```

        Console.WriteLine("c1.Equals(c2) = {0}", c1.Equals(c2));
        //c1 et c3 pointent sur la même référence.
        Console.WriteLine("c1 == c3 = {0}", c1 == c3);
        Console.WriteLine("c1.Equals(c3) = {0}", c1.Equals(c3));
    }
}

```



```

c1 == c2 = False
c1.Equals(c2) = False
c1 == c3 = True
c1.Equals(c3) = True

```

Il est préférable de redéfinir ce comportement pour implémenter votre propre critère de comparaison.

Exemple :

```

namespace Heritage
{
    public class MaClasse
    {
        public int P { get; set; }

        public override bool Equals(object obj)
        {
            //Tester si l'objet est null, dans ce cas il n'y a pas de
            //comparaison
            if (obj == null) return false;
            //Tester si obj est de meme type que la classe courante,
            //sinon il n'y a pas de comparaison
            if (obj.GetType() != this.GetType()) return false;
            //Convertir obj vers un objet de type la classe courante
            MaClasse maClasse = (MaClasse)obj;
            //Implémenter votre critère de comparaison
            if (maClasse.P != this.P) return false;
            return true;
        }
    }

    public class TestMaClasse
    {
        public static void Main(String[] args)
        {
            MaClasse c1 = new MaClasse();
            c1.P = 5;
            MaClasse c2 = new MaClasse();
            c2.P = 5;
            MaClasse c3 = c1;
        }
    }
}

```

```

//Sans redéfinir la méthode Equals, on compare les
//références
//c1 et c2 pointent sur des références différentes.
Console.WriteLine("c1 == c2 = {0}", c1 == c2);
Console.WriteLine("c1.Equals(c2) = {0}", c1.Equals(c2));
//c1 et c3 pointent sur la même référence.
Console.WriteLine("c1 == c3 = {0}", c1 == c3);
Console.WriteLine("c1.Equals(c3) = {0}", c1.Equals(c3));
    }
}
}

```



```

c1 == c2 = False
c1.Equals(c2) = True
c1 == c3 = True
c1.Equals(c3) = True

```

3. La méthode GetType :

La méthode GetType de la classe Object retourne un objet **Type** qui représente le type de la classe courante.

Exemple :

```

namespace Heritage
{
    public class MaClasse { }

    public class TestMaClasse
    {
        public static void Main(String[] args)
        {
            MaClasse c1 = new MaClasse();
            Console.WriteLine("c1.GetType() = {0}", c1.GetType());
        }
    }
}

```

```

c1.GetType() = Heritage.MaClasse

```