

SL228_POBJ Programmation Objet**Langage C#****Chapitre 2****Différences par rapport au C++**

CONTENU DU CHAPITRE 2

2. Différences par rapport au C++	2-1
2.1. Objectifs	2-1
2.2. Ou se situe le C#	2-1
2.3. Hello world version C#	2-1
2.3.1. Plusieurs constatations	2-2
2.3.2. Résultat dans la Console	2-2
2.4. Quelques différences	2-2
2.4.1. Extensions des fichiers	2-3
2.4.2. Les types du C#	2-3
2.4.2.1. Particularité de certain types C#	2-4
2.4.2.2. Indications post fixe	2-4
2.4.2.3. Le type String	2-4
2.4.2.4. Types, exemple	2-4
2.4.2.5. Affichage	2-5
2.5. Différences aux niveau des classes	2-6
2.5.1. Modèle UML utilisé	2-6
2.5.2. Classe Dessinateur	2-6
2.5.2.1. Contenu C++ : Dessinateur.h	2-6
2.5.2.2. Contenu C++ : Dessinateur.cpp	2-7
2.5.2.3. Contenu C# : Dessinateur.cs	2-7
2.5.3. Classe Figure	2-8
2.5.3.1. Contenu C++ : Figure.h	2-8
2.5.3.2. Contenu C++ : Figure.cpp	2-8
2.5.3.3. Contenu C# : Figure.cs	2-8
2.5.4. Classe Point	2-9
2.5.4.1. Contenu Point.cs	2-9
2.5.5. Classe Ligne	2-10
2.5.5.1. Contenu C++ : Ligne.h	2-10
2.5.5.2. Contenu C++ : Ligne.cpp	2-10
2.5.5.3. Contenu C# : Ligne.cs	2-10
2.5.6. Modification du projet HelloWorld	2-11
2.5.6.1. Fichiers composant le projet	2-11
2.5.6.2. Diagramme de classe	2-11
2.5.6.3. Ajout du dessinateur	2-11
2.6. Conclusion	2-12
2.7. Historique des versions	2-12
2.7.1. Version 1.0 Mars 2016	2-12

2. DIFFERENCES PAR RAPPORT AU C++

2.1. OBJECTIFS

L'objectif de ce chapitre est de permettre à l'étudiant d'utiliser les compétences acquises avec le C++ pour utiliser le C# en vue de réaliser des applications Windows Form.

Etant donnée la différence entre les applications console réalisée en C++ et les applications Windows Form, ce chapitre présentera les différences essentiel au niveau du langage.

2.2. OU SE SITUE LE C#

Le langage C# est plus proche du java que du C++, mais il n'apporte pas comme java l'avantage de l'exécution sur différente plateforme grâce aux machines virtuelles. Par contre la relation étroite avec le .NET Framework fait de C# le langage le plus approprié pour réaliser des applications Windows avec le Visual Studio.

2.3. HELLO WORLD VERSION C#

Pour mieux comprendre les différences voici la version C# du classique "HelloWorld", mais avec un Button pour déclencher l'action.

```
using System;
using System.Windows.Forms;

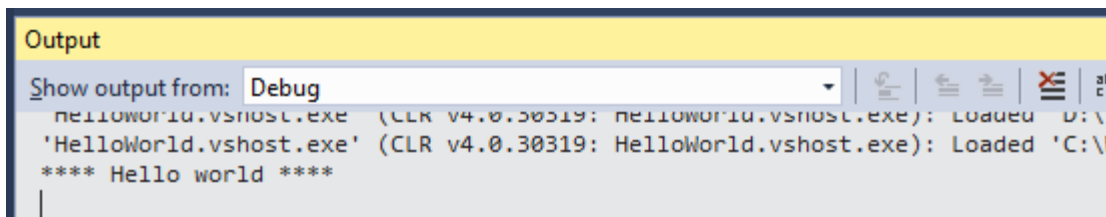
namespace HelloWorld
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnTest_Click(object sender, EventArgs e)
        {
            Console.WriteLine("**** Hello world ****");
        }
    }
}
```

2.3.1. PLUSIEURS CONSTATATIONS

- Un seul fichier .cs
- Un **namespace** correspondant au nom du projet contenant une classe qui correspond dans notre cas au formulaire.
- L'implémentation des méthodes est réalisée directement dans la classe.
- Dans la classe il n'y a pas de section public ou private, mais le mot clé est placé au début de chaque méthode.
- Il n'y a pas d' #include mais des **using**

2.3.2. RESULTAT DANS LA CONSOLE



```

Output
Show output from: Debug
HelloWorld.vshost.exe (CLR v4.0.30319: HelloWorld.vshost.exe): Loaded 'D:\C
'HelloWorld.vshost.exe' (CLR v4.0.30319: HelloWorld.vshost.exe): Loaded 'C:\V
**** Hello world ****

```

2.4. QUELQUES DIFFERENCES

Voici obtenu d'une source dont la référence a été perdue, une liste de quelques différences par rapport au C/C++ ; On notera particulièrement les points suivants :

- La manipulation directe de pointeurs ne peut se faire qu'au sein d'un code marqué *unsafe*, et seuls les programmes avec les permissions appropriées peuvent exécuter des blocs de code *unsafe*. La plupart des manipulations de pointeurs se font via des références sécurisées, dont l'adresse ne peut être directement modifiée, et la plupart des opérations de pointeurs et d'allocations sont contrôlées contre les dépassements de mémoire. Les pointeurs ne peuvent pointer que sur des *types de valeurs*, les types *objets*, manipulés par le [ramasse-miettes](#), ne pouvant qu'être référencés.
- Les objets ne peuvent pas être explicitement détruits. Le [ramasse-miettes](#) s'occupe de libérer la mémoire lorsqu'il n'existe plus aucune référence pointant sur un objet. Toutefois, pour les objets gérant des types non managés, il est possible d'implémenter l'interface `IDisposable` pour spécifier des traitements à effectuer au moment de la libération de la ressource.
- L'héritage multiple de classes est interdit, mais une classe peut implémenter un nombre illimité d'interfaces, et une interface peut hériter de plusieurs interfaces.
- Le C# est beaucoup plus [typé](#) que le C++ ; les seules conversions implicites sont celles entre les différentes gammes d'entiers et celles d'un type dérivé à un type parent. Aucune conversion implicite n'a lieu entre booléens et entiers, entre membres d'énumération et entiers, ni de pointeurs sur un type *void* (quoique pour ce dernier point l'utilisation de références sur le type `Object` permette d'obtenir le même effet). Les conversions définies par l'utilisateur peuvent être définies comme implicites ou explicites.
- La syntaxe pour la déclaration des [tableaux](#) n'est pas la même : `int[] a = new int[5]` remplace `int a[5]`. Car il s'agit d'une allocation dynamique, `int[] a` étant la déclaration d'une référence (nulle si non initialisée). L'allocation manuelle d'un tableau sur la pile reste cependant possible avec le mot clé `stackalloc`⁵.

- Les membres d'une énumération sont rassemblés dans leur propre [espace de noms](#).
- Le C# ne gère pas les *templates*, mais cette fonctionnalité a été remplacée par les [types génériques](#) apparus avec C# 2.0.
- Les propriétés ont été introduites, et proposent une syntaxe spécifique pour l'accès aux données membres (ainsi que la facilitation de l'accès simultané par plusieurs *threads*).
- La [réflexion](#) totale des types est disponible.
- Les délégués, qui sont des listes de pointeurs sur fonctions, sont utilisés notamment pour la programmation événementielle.

2.4.1. EXTENSIONS DES FICHIERS

👉 Un seul type de fichier avec l'extension **.cs**. Le fichier source C# contient une classe avec l'implémentation des méthodes.

2.4.2. LES TYPES DU C#

C# utilise les mêmes types que C++, mais n'accepte pas le mot clé **unsigned**, pour cela il y a des types spécifique qui correspondent aux type du :NET Framework.

Le tableau (source OpenClassroom), ci-dessous résume les types disponibles:

Type C#	Type .NET	Signification	Taille en mémoire (en octets)	Domaine de valeurs
char	Char	character (caractère)	2	caractère Unicode (UTF-16) allant de U+0000 à U+ffff
string	String	chaîne de caractères	variable	référence sur une séquence de caractères Unicode
int	Int32	integer (nombre entier)	4	$[-2^{31}; 2^{31}-1]$
uint	UInt32	unsigned integer (nombre entier non signé)	4	$[0; 2^{32}-1]$
long	Int64	nombre entier long	8	$[-2^{63}; 2^{63}-1]$
ulong	UInt64	unsigned long (nombre entier long non signé)	8	$[0; 2^{64}-1]$
sbyte	SByte	signed byte (octet signé)	1	$[-2^7; 2^7-1]$
byte	Byte	octet	1	$[0; 2^8-1]$
short	Int16	nombre entier court	2	$[-2^{15}; 2^{15}-1]$
ushort	UInt16	unsigned short (nombre entier court non signé)	2	$[0; 2^{16}-1]$
float	Single	flottant (nombre réel)	4	$\pm 1,5 \cdot 10^{-45}$ à $\pm 3,4 \cdot 10^{+38}$ (7 chiffres de précision)
double	Double	double flottant (nombre réel)	8	$\pm 5,0 \cdot 10^{-324}$ à $\pm 1,7 \cdot 10^{+308}$ (15 à 16 chiffres de précision)
decimal	Decimal	nombre décimal	16	$\pm 1,0 \cdot 10^{-28}$ à $\pm 7,9 \cdot 10^{+28}$ (28 à 29 chiffres significatifs)
bool	Boolean	booléen	1	true / false
object	Object	référence d'objet	variable	référence d'objet

Chacun des types C# présentés est un **alias** du type .NET associé, cela signifie que lorsque que vous écrivez par exemple **int**, c'est comme si vous aviez écrit **System.Int32**. Pour raccourcir et par habitude, on privilégie le type C# au type .NET.

2.4.2.1. PARTICULARITE DE CERTAIN TYPES C#

Le type **char** est 16 bits et n'est pas signé. Il supporte les caractères unicode. Un caractère unicode s'exprime par '\uCCCC' avec CCCC qui représente le code du caractère exprimé en hexadécimal.

Le type **byte** remplace le **unsigned char** du C++.

2.4.2.2. INDICATIONS POST FIXE

Les valeurs numériques doivent être post fixée pour correspondre au type, ceci en particulier pour le type float et long. Par exemple :

```
int toto = 25;
long big = 123456789000000L;
```

```
double y = 1.25;
float x = 1.25F;
```

2.4.2.3. LE TYPE STRING

Comme en C++ n'est pas un type mais une classe. En C# c'est comme en C++ **string**.

2.4.2.4. TYPES, EXEMPLE

Voici un exemple de déclaration et affectations de valeur à des variables de différent types. Ainsi que l'affichage du contenu des variables.

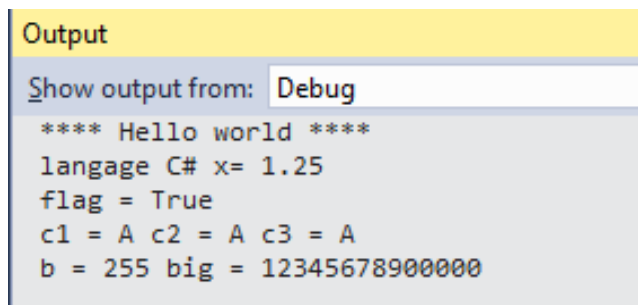
```
namespace HelloWorld
{
    public partial class Form1 : Form
    {
        int toto = 25;
        short titi = 234;
        char c1 = 'A';
        char c2 = (char)0x41;
        char c3 = '\u0041';
        byte b = 0xFF;
        long big = 123456789000000L;
        bool flag = true;
        float x = 1.25F;
        double y = 1.25;
        string Mess = "langage C#";

        public Form1()
        {
            InitializeComponent();
        }
    }
}
```



```
private void btnTest_Click(object sender, EventArgs e)
{
    Console.WriteLine("**** Hello world ****");
    Console.WriteLine("langage C# x= " + x);
    Console.WriteLine("flag = " + flag);
    Console.WriteLine("c1 = " + c1 + " c2 = " + c2 +
        " c3 = " + c3);
    Console.WriteLine("b = " + b + " big = " + big);
}
}
```

Voici le résultat au niveau de la console:



```
Output
Show output from: Debug
**** Hello world ****
langage C# x= 1.25
flag = True
c1 = A c2 = A c3 = A
b = 255 big = 123456789000000
```

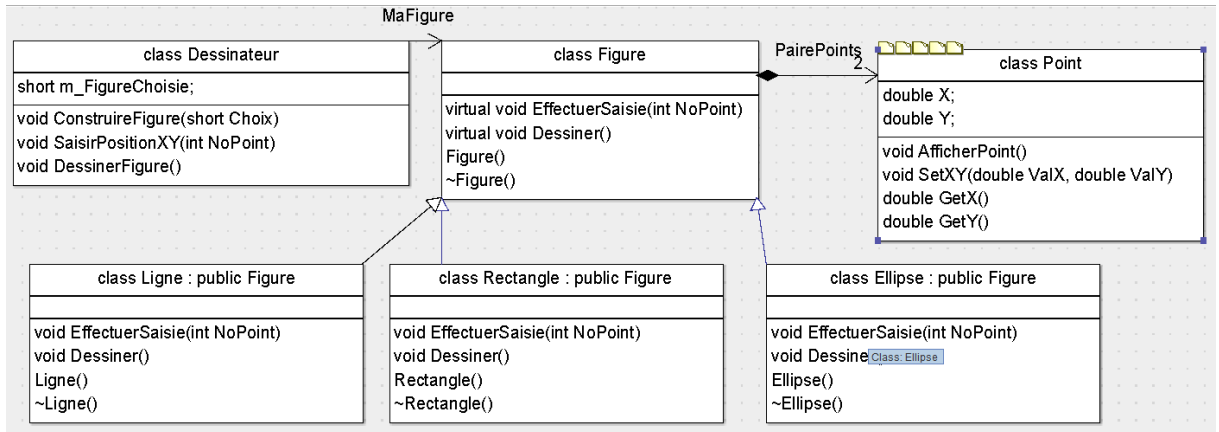
2.4.2.5. AFFICHAGE

Au niveau de l'affichage on observe l'utilisation du `Console.WriteLine`, avec la possibilité de mélanger du texte et des valeurs numériques. L'opérateur `+` joue le rôle d'opérateur de concaténation.

2.5. DIFFERENCES AUX NIVEAU DES CLASSES

Pour illustrer les différences nous partons d'un modèle UML, mais comme la sélection du CSharp ne produit pas de code nous allons transformer le java en C#.

2.5.1. MODELE UML UTILISE



Dans ce modèle on trouve de l'héritage ainsi une association et une agrégation.

2.5.2. CLASSE DESSINATEUR

2.5.2.1. CONTENU C++ : DESSINATEUR.H

```

#ifndef Dessinateur_h
#define Dessinateur_h

class Figure;

class Dessinateur {
public:
    void ConstruireFigure(short Choix);
    void SaisirPositionXY(int NoPoint);
    void DessinerFigure();

private:
    short m_FigureChoisie;
public:
    /**
     * @element-type Figure
     */
    Figure *MaFigure;
};

#endif // Dessinateur_h
  
```

2.5.2.2. CONTENU C++ : DESSINATEUR.CPP

```
#include "Dessinateur.h"

void Dessinateur::ConstruireFigure(short Choix)
{
}

void Dessinateur::SaisirPositionXY(int NoPoint)
{
}

void Dessinateur::DessinerFigure()
{
}
```

2.5.2.3. CONTENU C# : DESSINATEUR.CS

```
public class Dessinateur {

    private short m_FigureChoisie;

    Figure MaFigure = new Figure();

    public void ConstruireFigure(short Choix) {
    }

    public void SaisirPositionXY(int NoPoint) {
    }

    public void DessinerFigure() {
    }
    // besoin d'un constructeur
    public Dessinateur() {
    }
}
```

On constate plusieurs différences :

- L'accessibilité (**public** ou **private**) précède la class ainsi que chaque méthode et attributs.
- L'implémentation suit directement la déclaration de la méthode.
- L'association qui en C++ est un pointeur, est réalisée en utilisant une référence à un objet avec allocation.
- il n'y a pas de référence à la class Figure, elle est connue du projet pour autant que le fichier **Figure.cs** soit présent dans le projet et valable.

2.5.3. CLASSE FIGURE

2.5.3.1. CONTENU C++ : FIGURE.H

```
#ifndef Figure_h
#define Figure_h

#include "Point.h"

class Figure {

public:
    virtual void EffectuerSaisie(int NoPoint);
    virtual void Dessiner();
    Figure();
    ~Figure();
public:
    /**
     * @element-type Point
     */
    Point PairePoints[2];
};

#endif // Figure_h
```

2.5.3.2. CONTENU C++ : FIGURE.CPP

Voici seulement le début de figure.cpp, car le principe reste le même pour toutes les méthodes.

```
#include "Figure.h"

void Figure::EffectuerSaisie(int NoPoint)
{
}
```

2.5.3.3. CONTENU C# : FIGURE.CS

```
public class Figure {
    // composition
    PointDemo[] PairePoints = new PointDemo[2];
    public virtual void EffectuerSaisie(int NoPoint) {
    }
    public virtual void Dessiner() {
    }
    public Figure() {
    }
    ~Figure() {
    }
}
```

On constate encore quelques différences :

- Même principe pour la composition qu'en C++, par contre la déclaration d'un tableau est spécifique au C# et nécessite de l'allocation.
- Le destructeur s'écrit sans private ou public.

2.5.4. CLASSE POINT

Il n'y a rien de particulier au niveau de la classe Point. Voici uniquement le contenu du .cs. Remarque: la class Point a été renommée PointDemo car il existe une classe Point avec le .NET Framework.

2.5.4.1. CONTENU POINT.CS

```
public class PointDemo {  
  
    private double X;  
    private double Y;  
  
    public void AfficherPoint() {  
    }  
  
    public void SetXY(double ValX, double ValY) {  
        X = ValX;  
        Y = ValY;  
    }  
  
    public double GetX() {  
        return X;  
    }  
  
    public double GetY() {  
        return Y;  
    }  
  
    public PointDemo() {  
    }  
}
```

Les méthodes GetX, GetY et SetXY sont entièrement implémentées.

2.5.5. CLASSE LIGNE

Il s'agit d'une classe qui hérite de Figure. Le principe sera le même pour les 2 autres classes dérivées.

2.5.5.1. CONTENU C++ : LIGNE.H

```
#ifndef Ligne_h
#define Ligne_h

#include "Figure.h"

class Ligne : public Figure {

public:
    void EffectuerSaisie(int NoPoint);
    void Dessiner();
    Ligne();
    ~Ligne();
};
#endif // Ligne_h
```

2.5.5.2. CONTENU C++ : LIGNE.CPP

Voici seulement le début de Ligne.cpp, car le principe reste le même pour toutes les méthodes.

```
#include "Ligne.h"

void Ligne::EffectuerSaisie(int NoPoint)    {
}
```

2.5.5.3. CONTENU C# : LIGNE.CS

```
public class Ligne : Figure {
    public override void EffectuerSaisie(int NoPoint) {
    }

    public override void Dessiner() {
    }

    public Ligne() : base() {
    }

    ~Ligne() {
    }
}
```

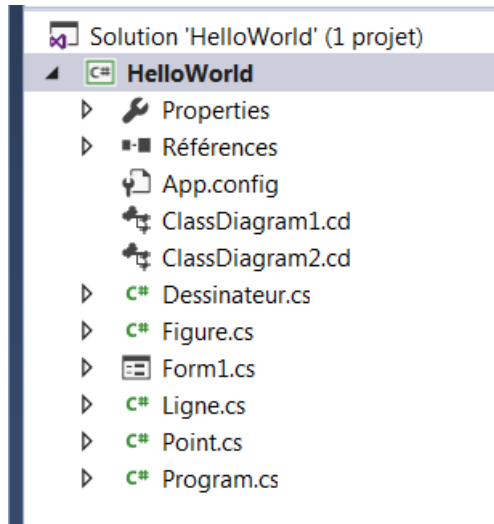
On constate encore quelques différences :

- Le mot clé **override** est nécessaire pour indiquer que la méthode remplace effectivement la méthode virtuelle de la classe parent.
- Dans le constructeur on utilise **base()** pour appeler le constructeur de la classe de base.

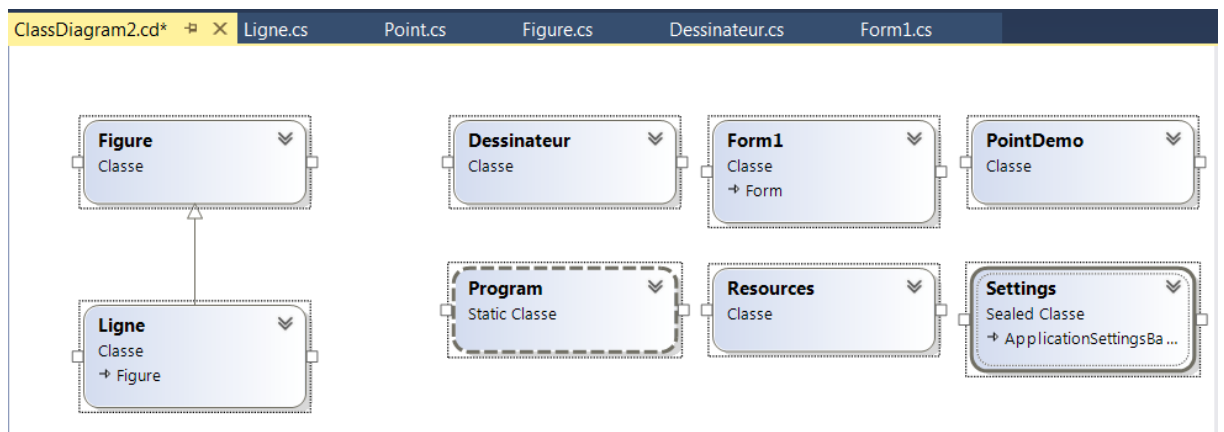
2.5.6. MODIFICATION DU PROJET HELLOWORLD

Pour tester la compilation des différents fichiers .cs qui ont été ajoutés au projet, un Dessinateur a été créé.

2.5.6.1. FICHIERS COMPOSANT LE PROJET



2.5.6.2. DIAGRAMME DE CLASSE



2.5.6.3. AJOUT DU DESSINATEUR

Voici l'ajout d'un membre dessinateur, sa construction et l'appel de la méthode ConstruireFigure.

```

// Dessinateur
Dessinateur MonDessinateur;

public Form1()
{
    InitializeComponent();
    MonDessinateur = new Dessinateur();
    MonDessinateur.ConstruireFigure(1);
}

```

2.6. CONCLUSION

A l'issue de ce chapitre, l'étudiant, aura découvert quelques différences entre le C++ et le C#. Il devra en tenir compte lors de la réalisation de projet avec le Visual Studio.

2.7. HISTORIQUE DES VERSIONS

2.7.1. VERSION 1.0 MARS 2016

Création du document à partir du même chapitre pour le Java.