



DOCUMENTATION TECHNIQUE

Romain SIEURIN

C# avril 2023



Table des matières

Préface.....	3
1- Début & Base du C#.....	3
A) Premières lignes en C#	3
B) Séquences d'échappement	4
C) Opérateurs.....	4
D) Type de Variables	4
E) Les instructions 'if' et 'else'	5
F) Les Tableaux	5
2- Le C# avec Visual Studio	6
A) Qu'est-ce qu'un IDE ?	6
B) Configurer un nouveau projet.....	7
C) Quelques raccourcis clavier sous Visual Studio 2019	9
3- La Programmation Orientée Objet en C#.....	10
A) Pourquoi utiliser la programmation orientée objet ?	10
B) Première approche de la POO.....	10
C) Différence déclaration et instanciation.....	13
4- Le concept des objets.....	13
A) Le constructeur	13
B) Les mots-clés public et private.....	14
C) Les Accesseurs	15
5- La relation d'association.....	16
A) Principe	16
B) Exemple.....	16
6- La relation d'héritage	17
A) Principe	17
B) Exemple.....	17
7- Projet Bibliothèque	18
A) Les objets	18
B) Le codage	20
C) Les relations d'associations.....	21

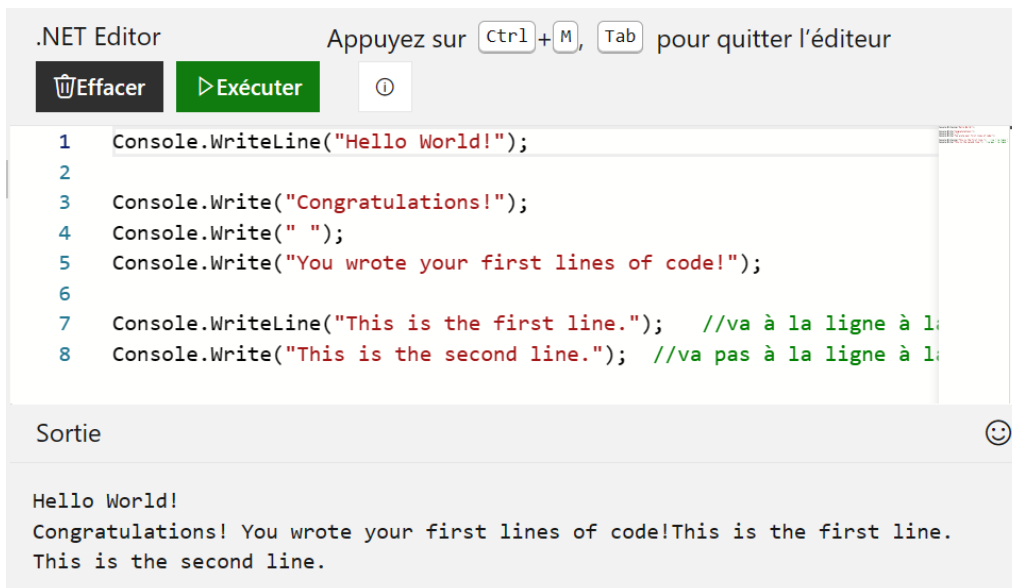
Préface

Nous avons commencé le C# en janvier 2023, c'est un langage de programmation orienté objet développé par Microsoft. Il est conçu pour être simple, sûr et moderne, tout en offrant une grande puissance et flexibilité aux développeurs. Le langage est basé sur le Framework .NET, qui fournit une infrastructure riche et cohérente pour la création d'applications pour différentes plateformes. Pour apprendre le C#, nous nous sommes familiariser avec les concepts de base tels que les variables, les types de données, les boucles et les conditions, les fonctions et les classes. Il a été également important de comprendre les principes de la programmation orientée objet, tels que l'encapsulation et l'héritage. Cette documentation technique permet de regrouper toute ces notions.

1- Début & Base du C#

A) Premières lignes en C#

Lors des premières séances nous avons écrit nos premières lignes de codes simple et compréhensibles.



The screenshot shows the .NET Editor interface. At the top, there's a title bar with the text ".NET Editor" and a hint "Appuyez sur Ctrl+M, Tab pour quitter l'éditeur". Below the title bar, there are three buttons: "Effacer" (with a trash icon), "Exécuter" (with a play icon), and a help icon. The main area contains C# code with line numbers 1 through 8. The code uses `Console.WriteLine` and `Console.Write` to output text. The output window at the bottom, titled "Sortie", shows the result of running the code: "Hello World!", "Congratulations! You wrote your first lines of code!", "This is the first line.", and "This is the second line.".

```
1 Console.WriteLine("Hello World!");
2
3 Console.Write("Congratulations!");
4 Console.Write(" ");
5 Console.Write("You wrote your first lines of code!");
6
7 Console.WriteLine("This is the first line."); //va à la ligne à l
8 Console.Write("This is the second line."); //va pas à la ligne à l
```

Sortie

```
Hello World!
Congratulations! You wrote your first lines of code!This is the first line.
This is the second line.
```

Le bouton vert **Exécuter** effectue deux tâches :

- Il compile votre code dans un format exécutable que l'ordinateur peut comprendre.
- Il exécute votre application compilée et génère la commande souhaitée.

Nous avons vu que la méthode `Console.WriteLine()` permet d'afficher une ligne de texte dans la console avec un retour à la ligne à la fin, tandis que la méthode `Console.Write()` permet d'afficher du texte sans retour à la ligne à la fin.

B) Séquences d'échappement

En C#, les séquences d'échappement sont utilisées pour insérer des caractères spéciaux dans des chaînes de caractères. Elles sont représentées par un caractère d'échappement qui est une barre oblique inverse "\" suivi d'un caractère spécial.

- `\n` : saut de ligne
- `\r` : retour chariot
- `\t` : tabulation horizontale
- `\'` : apostrophe
- `\"` : guillemets

Par exemple, pour créer une chaîne de caractères avec une tabulation suivie de la lettre "a", on peut utiliser la séquence d'échappement `\t` :

```
string maChaine = "Voici une \tleffre a";  
Console.WriteLine(maChaine);
```

Cela affiche :

```
Voici une      leffre a
```

C) Opérateurs

En C#, il existe plusieurs opérateurs qui permettent de réaliser des opérations entre des variables ou des valeurs. Voici une liste des principaux opérateurs en C# :

- Opérateurs arithmétiques : `+` (addition), `-` (soustraction), `*` (multiplication), `/` (division), `%` (modulo).
- Opérateurs de comparaison : `==` (égal à), `!=` (différent de), `<` (inférieur à), `<=` (inférieur ou égal à), `>` (supérieur à), `>=` (supérieur ou égal à).
- Opérateurs logiques : `&&` (et), `||` (ou).
- Opérateurs d'incrément et de décrémentation : `++` (incrément), `--` (décrément).

D) Type de Variables

Il existe plusieurs types de variables en C#, notamment :

- Les types numériques, tels que `int` pour les nombres entiers, `double` pour les nombres à virgule flottante, `decimal` pour les nombres décimaux, etc. Exemple : `int age = 30;`
- Les types booléens, qui ne peuvent avoir que deux valeurs : `true` ou `false`. Exemple : `bool estMajeur = true;`
- Les types de caractères, tels que `char` pour un seul caractère ou `string` pour une chaîne de caractères. Exemple : `char sexe = 'F';`
- Les types d'énumération, qui permettent de définir un ensemble de valeurs nommées. Exemple : `enum Couleur { Rouge, Vert, Bleu };`
- Les types de tableau, qui permettent de stocker plusieurs valeurs dans une même variable. Exemple : `int[] notes = { 12, 14, 16, 18 };`

Nous avons vu également que le C# est un langage sensible à la casse. Cela signifie que les noms de variables, de fonctions et d'autres éléments du code doivent être écrits avec la même casse lorsqu'ils sont appelés ou utilisés dans le code. Par exemple, si une variable est nommée "maVariable" avec une minuscule initiale, elle ne peut pas être appelée avec une majuscule initiale comme "MaVariable".

E) Les instructions 'if' et 'else'

Les instructions "if else" en C# permettent de définir un bloc de code à exécuter en fonction d'une condition booléenne. Si la condition est vraie, le code spécifié dans le bloc "if" est exécuté, sinon le code dans le bloc "else" est exécuté.

```
int a = 10;
int b = 20;

if (a > b)
{
    Console.WriteLine("a est plus grand que b");
}
else
{
    Console.WriteLine("b est plus grand que a");
}
```

Dans cet exemple, la condition `a > b` est fausse car `a` vaut 10 et `b` vaut 20, donc le code spécifié dans le bloc "else" sera exécuté et affichera "b est plus grand que a" dans la console.

F) Les Tableaux

Nous avons vu que pour utiliser un tableau, nous devons tout d'abord déclarer le tableau en spécifiant son type de données et sa taille. Par exemple, pour déclarer un tableau d'entiers de taille 5, on peut utiliser la syntaxe suivante :

```
int[] tableauEntiers = new int[5];
```

Pour accéder aux éléments du tableau, on peut utiliser l'index du tableau, qui commence à zéro. Par exemple, pour affecter la valeur 10 au premier élément du tableau, vous pouvez écrire :

```
tableauEntiers[0] = 10;
```

Pour récupérer la valeur de l'élément du tableau, nous pouvons utiliser l'instruction 'foreach'. Voici un exemple :

```
int[] tableau = { 1, 2, 3, 4, 5 };  
foreach (int element in tableau)  
{  
    Console.WriteLine(element);  
}
```

Dans cet exemple, le tableau contient les entiers de 1 à 5. Le mot-clé foreach permet de parcourir tous les éléments du tableau un par un. Pour chaque élément, la variable element prend la valeur de l'élément en cours de traitement. Le code à l'intérieur de la boucle foreach affiche simplement la valeur de l'élément courant à l'aide de la méthode Console.WriteLine().

Attention ! En C#, le point-virgule (;) est utilisé pour marquer la fin d'une instruction. Il est donc indispensable de le mettre à la fin de chaque instruction pour que le code soit valide et compilable. Toutefois, certains éditeurs de code peuvent ajouter automatiquement le point-virgule à la fin de chaque instruction, ce qui peut donner l'impression qu'il n'est pas nécessaire de le taper. Il est cependant recommandé de toujours mettre le point-virgule soi-même pour éviter des erreurs de syntaxe.

2- Le C# avec Visual Studio

A) Qu'est-ce qu'un IDE ?

Un IDE (Integrated Development Environment, AGL en Français pour Atelier de Génie Logiciel) est étudié pour faciliter le développement de programmes. Pour cela, il propose de nombreuses fonctionnalités étendues, comme la gestion d'un projet, la

compilation automatique, la gestion des liens, la gestion des dépendances, la gestion des erreurs, la gestion de la documentation, la gestion des tests unitaires, la gestion des bibliothèques, etc. Il ne font toutefois pas toujours l'unanimité.

Eclipse, Kdevelopp, PHPStorm, Netbeans sont des IDE.

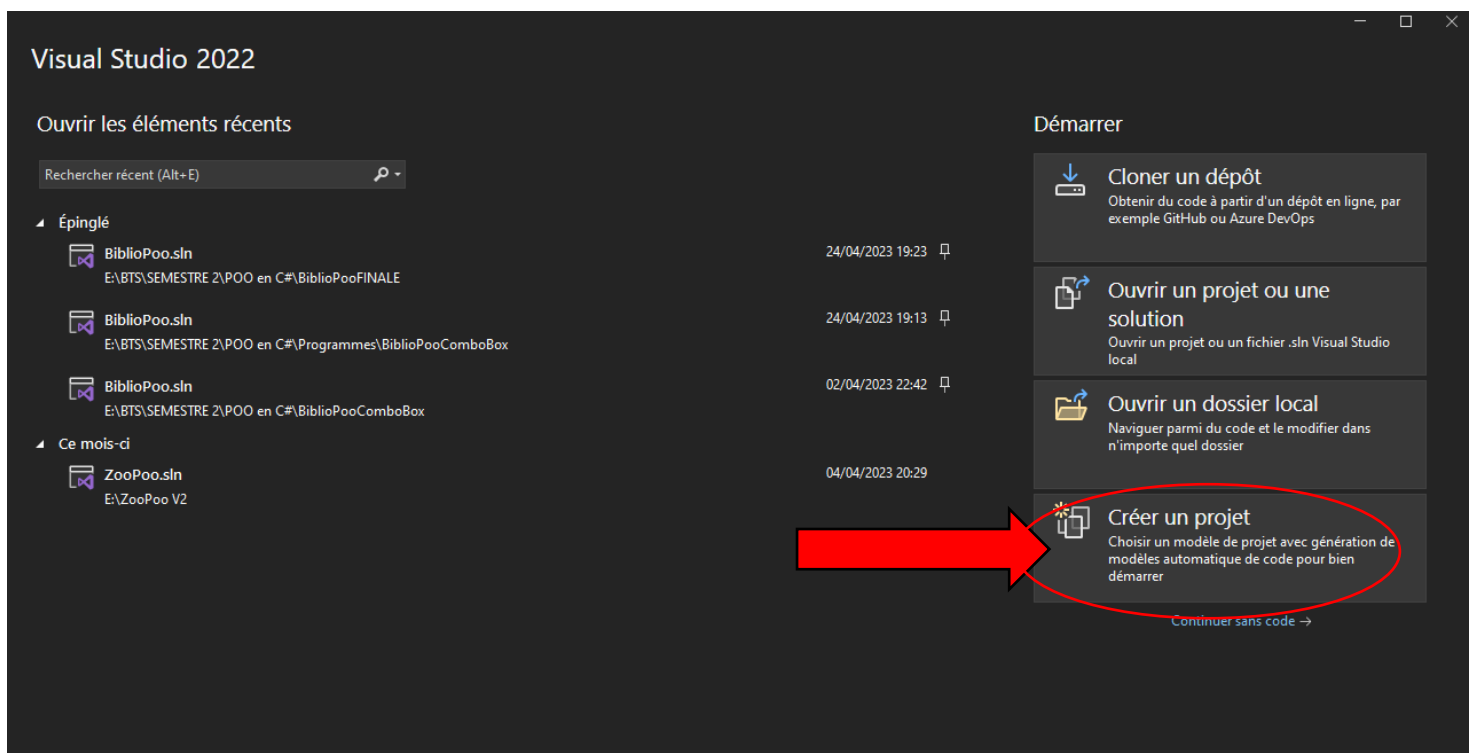
C# est un langage de programmation orienté objet développé par Microsoft. Visual Studio fournit des fonctionnalités telles que le débogage, la gestion de projet et la conception d'interface utilisateur pour simplifier et accélérer le processus de développement en C#.

B) Configurer un nouveau projet

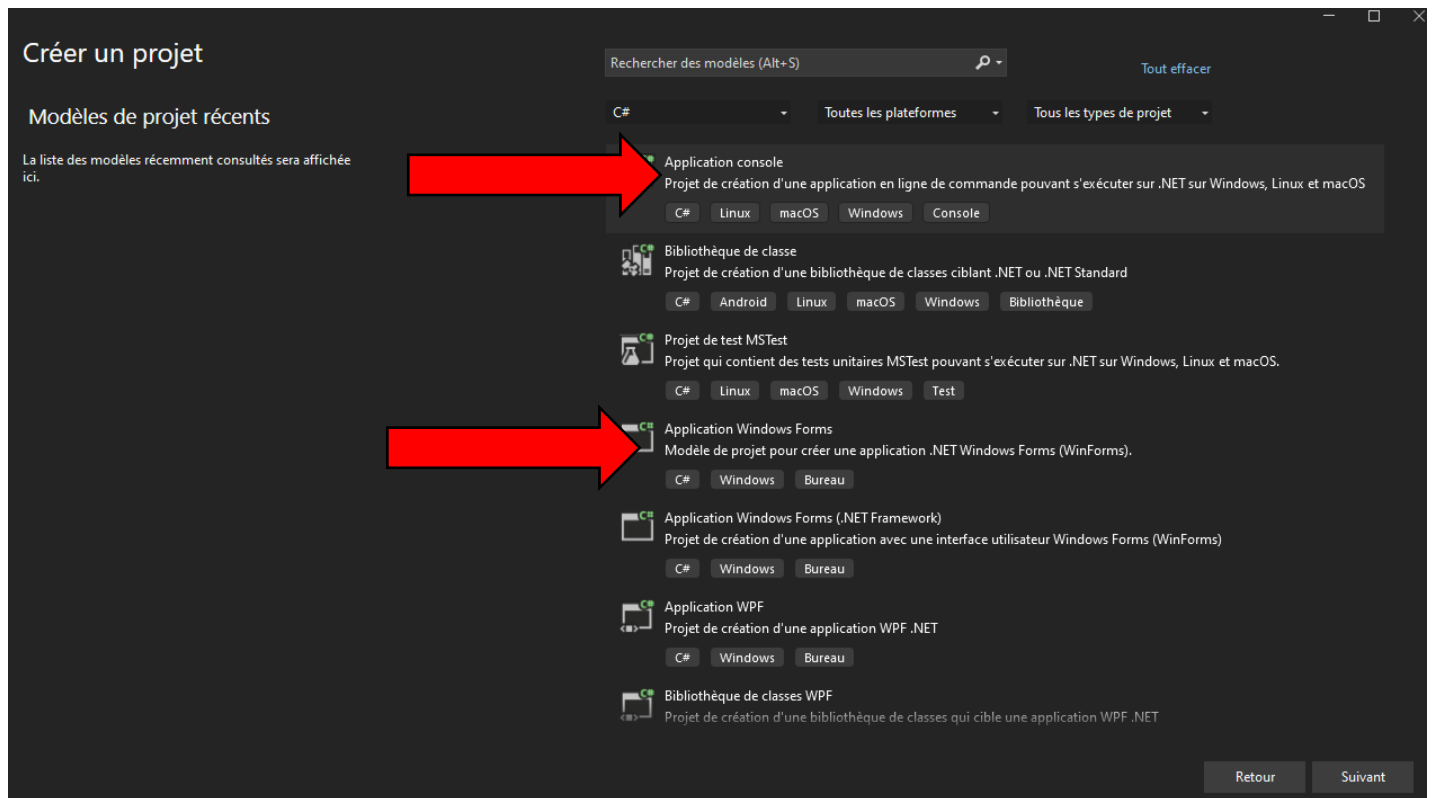
Un fois l'installation effectuer de l'IDE Visual Studio, nous avons la page qui suit:

Pour créer un nouveau projet dans Visual Studio :

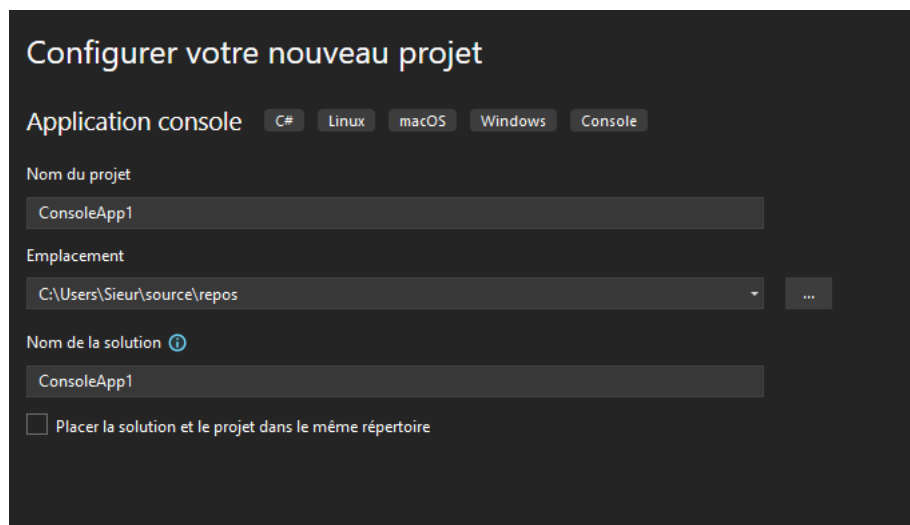
1. Cliquez sur « Créer un nouveau projet » (Indiquez par la flèche)



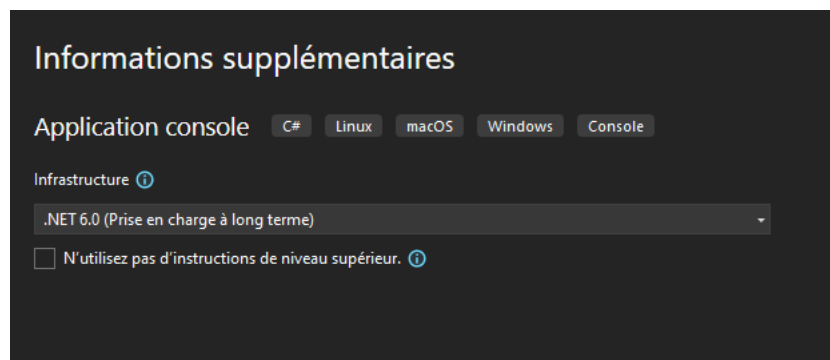
2. Puis sélectionnez le type de projet que vous souhaitez créer, par exemple "Application console" ou "Application Windows Forms".



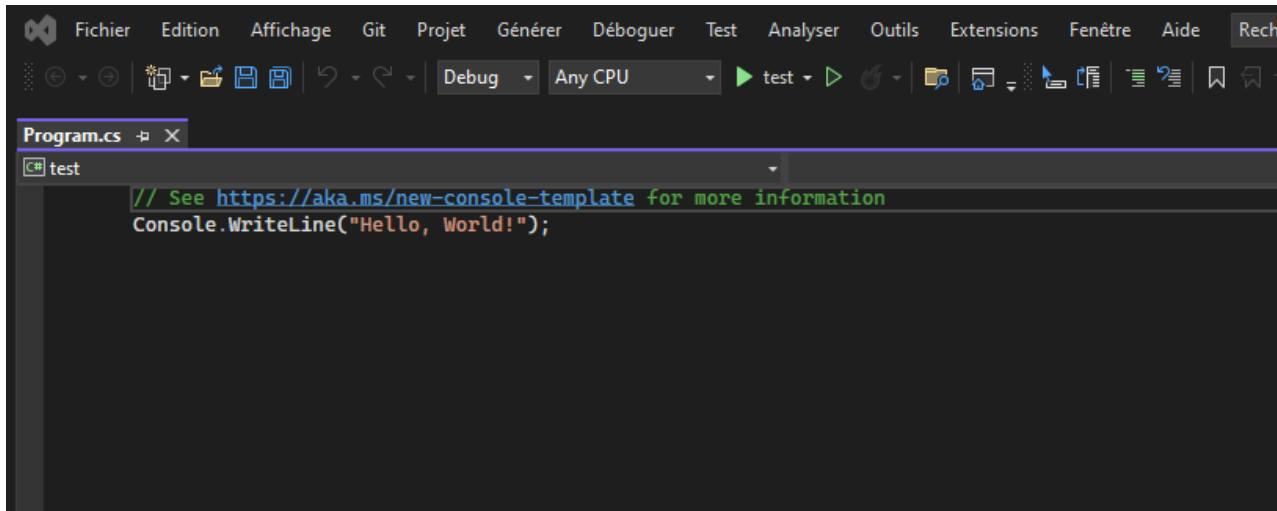
3. Ensuite, choisissez un nom et un emplacement pour votre projet, puis cliquez sur "Créer". **Attention !** Le nom du projet ne doit pas contenir d'accent.



4. Enfin, nous choisissons le Framework 6.0 (varie selon les MAJ de l'IDE.)



5. Le projet est maintenant créé. Avec, on trouve le squelette du projet.



Lors de la création d'un nouveau projet dans Visual Studio, l'IDE permet de sélectionner un modèle de projet prédéfini. En fonction du type de projet sélectionné, des fichiers de démarrage et des configurations par défaut seront générés automatiquement.

'**static void Main()**' est une méthode obligatoire dans une application console C#. C'est la méthode qui sert de point d'entrée de l'application. C'est à partir de cette méthode que le code de l'application commence à s'exécuter lorsqu'elle est exécutée.

La méthode **Main** doit être déclarée comme **static** car elle est appelée sans qu'il soit nécessaire d'instancier la classe. Elle doit être déclarée comme **void** car elle ne renvoie pas de valeur à l'appelant.

C) Quelques raccourcis clavier sous Visual Studio 2019

Atteindre la ligne → **Ctrl+G**,

Mettre le code en forme → **Ctrl+K, Ctrl+D**,

Plein écran → **Alt+Maj+Entrée**,

Exécuter avec débogage → **F5**,

Exécuter sans débogage → **Ctrl+F5**,

Rechercher dans les fichiers → **Ctrl+Maj+F**,

Sélection et passage en mode colonne → **Alt+Shift**,

Commenter un bloc de code → **Ctrl+K, Ctrl+C**,

Décommenter un bloc de code → **Ctrl+K, Ctrl+U**

3- La Programmation Orientée Objet en C#

A) Pourquoi utiliser la programmation orientée objet ?

La programmation orientée objet (POO) est utilisée pour plusieurs raisons, notamment :

1. Modélisation du monde réel : la POO permet de modéliser le monde réel en créant des objets qui ont des propriétés et des comportements similaires à ceux des objets physiques. Elle permet de concevoir des programmes plus intuitifs et plus faciles à comprendre.
2. Réutilisation du code : la POO permet de réutiliser le code en créant des classes et des objets qui peuvent être utilisés dans plusieurs parties du programme. Elle permet également de faciliter la maintenance et la mise à jour du code en cas de modification de l'application.
3. Encapsulation : la POO permet de cacher la complexité du code en encapsulant les données et les méthodes dans des classes. Elle permet également de protéger les données en limitant l'accès aux propriétés et méthodes des objets.

B) Première approche de la POO

LES OBJETS

Quand on utilise la POO, on cherche à représenter le domaine étudié sous la forme d'objets. C'est la phase de **modélisation orientée objet**.

Un **objet** est une entité qui représente (*modélise*) un élément du domaine étudié : une voiture, un compte bancaire, un nombre complexe, une facture, etc.

Objet = état + actions

Cette équation signifie qu'un objet rassemble à la fois :

- des **informations** (ou données) qui le caractérisent.
- des **actions** (ou traitements) qu'on peut exercer sur lui.

Imaginons qu'on souhaite modéliser des comptes bancaires pour un logiciel de gestion. On commence par réfléchir à ce qui caractérise un compte bancaire, puis on classe ces éléments en deux catégories :

- les informations liées à un compte bancaire.
- les actions réalisables sur un compte bancaire.

En première approche, on peut considérer qu'un compte bancaire est caractérisé par un **titulaire**, un **solde** (le montant disponible sur le compte) et utilise une certaine **devise** (euros, dollars, etc). Les actions réalisables sur un compte sont le dépôt d'argent (**crédit**) et le retrait (**débit**).

On peut regrouper les caractéristiques de notre objet "compte bancaire" dans ce tableau.

Informations	Actions
titulaire, solde, devise	créditer, débiter

Un **objet** se compose **d'informations** et **d'actions**. Les actions utilisent (et parfois modifient) les informations de l'objet.

- L'ensemble des informations d'un objet donné est appelée son **état**.
- L'ensemble des actions applicables à un objet représente son **comportement**.

REMARQUE : les actions associées à un objet s'expriment généralement sous la forme de verbes à l'infinitif (*créditer, débiter*).

A un instant donné, l'état d'un objet "compte bancaire" sera constitué par les valeurs de son titulaire, son solde et sa devise. L'état de l'objet "compte bancaire de Paul" sera, sauf exception, différent de l'état de l'objet "compte bancaire de Pierre".

LES CLASSES

Nous venons de voir que l'on pouvait représenter un compte bancaire sous la forme d'un objet. Imaginons que nous voulions gérer les différents comptes d'une banque. Chaque compte aura son propre titulaire, son solde particulier et sa devise. Mais tous les comptes auront un titulaire, un solde et une devise, et permettront d'effectuer les mêmes opérations de débit/crédit. Chaque objet "compte bancaire" sera construit sur le même modèle : ce modèle est appelée une **classe**.

Une **classe** est un **modèle d'objet**. C'est un nouveau type créé par le programmeur et qui sert de modèle pour tous les objets de cette classe. Une classe spécifie les informations et les actions qu'auront en commun tous les objets qui en sont issus.

Le compte en banque appartenant à Jean, dont le solde est de 450 euros, est un compte bancaire particulier. Il s'agit d'un objet de la classe "compte bancaire". En utilisant le vocabulaire de la POO, on dit que l'objet "compte bancaire de Jean" est une **instance** de la classe "compte bancaire".

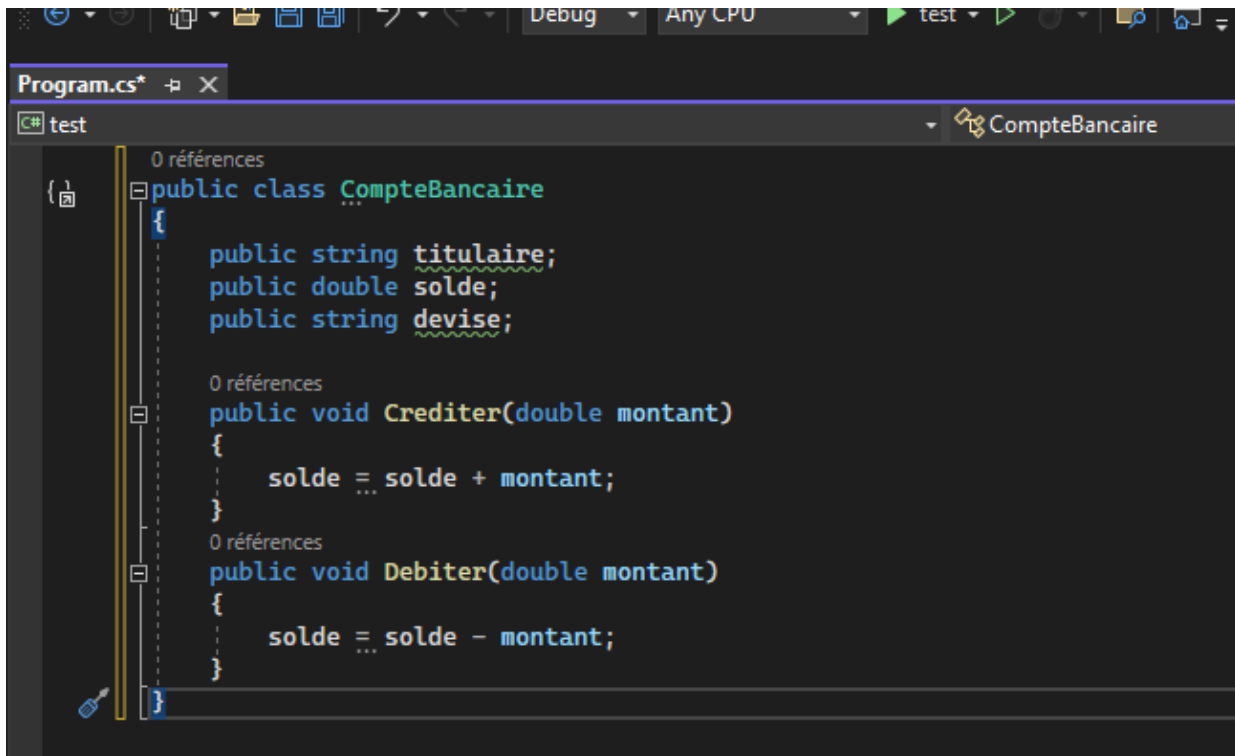
ATTENTION ! : ne pas confondre **objet** et **classe**. Une classe est un type abstrait (exemple : un compte bancaire en général), un objet est un exemplaire concret d'une classe (exemple : le compte bancaire de Jean).

Un objet est une variable particulière dont le type est une classe.

En résumé

- La POO consiste à programmer en utilisant des **objets**.
- Un objet modélise un élément du domaine étudié (exemples : un compte bancaire, une voiture, un satellite, etc).
- Un objet est une **instance** de **classe**. Une classe est un type abstrait, un objet est un exemplaire concret de cette classe.
- Une classe regroupe des **informations** et des **actions**.
- Les informations sont stockées sous la forme de **champs**. Les champs décrivent l'**état** d'un objet.
- Les actions réalisables sur un objet sont représentés par des **méthodes**. Elles expriment ce que les objets peuvent faire, leur **comportement**.

Voici la traduction en C# de la classe CompteBancaire :



C) Différence déclaration et instantiation

La déclaration d'une variable consiste à définir son type et son nom, tandis que l'instanciation consiste à allouer de l'espace en mémoire pour cette variable. En d'autres termes, la déclaration spécifie le type et le nom de la variable, tandis que l'instanciation crée une instance de cette variable et alloue de l'espace en mémoire pour stocker sa valeur.

Voici un exemple en C# pour illustrer la différence :

```
// Déclaration d'une variable de type int nommée "nombre"
int nombre;

// Instanciation de la variable "nombre" avec la valeur 5
nombre = 5;
```

Dans cet exemple, la première ligne déclare une variable de type int appelée "nombre". La deuxième ligne instancie cette variable en lui attribuant une valeur de 5.

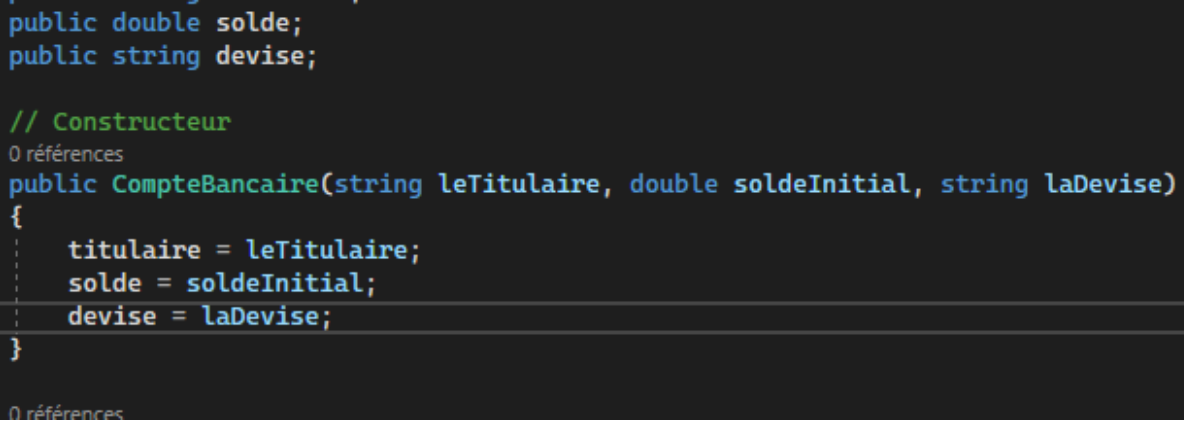
4- Le concept des objets

A) Le constructeur

En programmation orientée objet, un constructeur est une méthode spéciale appelée lors de la création d'un objet. Le constructeur est généralement utilisé pour initialiser les variables de l'objet lors de sa création. Il peut prendre des arguments pour spécifier les valeurs initiales de certaines variables et peut être surchargé pour fournir plusieurs façons de créer un objet. Le nom du constructeur est toujours identique au nom de la classe et il n'a pas de type de retour.

Reprenons l'exemple de la classe `CompteBancaire` du point précédent.

Tout compte a nécessairement un titulaire, un solde initial et une devise lors de sa création. On aimerait pouvoir instancier un objet de la classe `CompteBancaire` en définissant directement les valeurs de ses attributs. Pour cela, nous allons ajouter à notre classe une méthode particulière : le **constructeur**.



```
public double solde;
public string devise;

// Constructeur
0 références
public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)
{
    titulaire = leTitulaire;
    solde = soldeInitial;
    devise = laDevise;
}

0 références
```

Le **constructeur** est une méthode spécifique dont le rôle est de construire un objet, le plus souvent en initialisant ses attributs.

L'utilisation d'un constructeur se fait au moment de l'instanciation de l'objet (opérateur `new`), en passant en paramètres les futures valeurs des attributs de l'objet créé.

B) Les mots-clés `public` et `private`

En programmation orientée objet, les mots clés `public` et `private` sont utilisés pour spécifier le niveau d'accès aux variables et aux méthodes d'une classe. Ils permettent de modifier le **niveau d'encapsulation** (on parle aussi de **visibilité** ou **d'accessibilité**) des éléments de la classe (attributs et méthodes) :

- **public** : une variable ou une méthode déclarée avec le mot clé `public` peut être accédée depuis n'importe où dans le programme, y compris à l'extérieur de la classe.

- **private** : une variable ou une méthode déclarée avec le mot clé private ne peut être accédée qu'à l'intérieur de la classe. Les variables et les méthodes déclarées comme privées sont généralement utilisées pour encapsuler les données d'une classe et empêcher l'accès direct à ces données depuis l'extérieur de la classe.

Remarque ! Le niveau d'encapsulation intermédiaire **protected** est utilisé pour définir des membres (méthodes et variables) d'une classe qui ne sont accessibles qu'à la fois par la classe elle-même et par ses sous-classes (ou classes dérivées), mais qui ne sont pas accessibles depuis l'extérieur de la classe. Cela permet d'assurer que les données importantes d'une classe ne sont accessibles que par les méthodes de cette classe et ses sous-classes, tout en permettant aux sous-classes d'hériter des comportements de la classe parente et de les adapter à leurs propres besoins.

C) Les Accesseurs

En C#, les accesseurs sont des méthodes spéciales qui permettent d'accéder aux valeurs d'attributs d'une classe en offrant une interface de lecture et d'écriture.

Il existe deux types d'accesseurs : les accesseurs **get** et les accesseurs **set**. L'accesseur **get** permet de lire la valeur d'un attribut tandis que l'accesseur **set** permet de modifier la valeur de l'attribut.

Les accesseurs sont généralement utilisés pour contrôler l'accès aux attributs d'une classe et permettre une certaine encapsulation.

Voici la classe `CompteBancaire` modifiée pour intégrer des accesseurs vers ses attributs, ainsi que son nouveau diagramme de classe.

```
public string devise,  
  
    public string Titulaire { get; set; }  
  
    public double Solde { get; set; }  
  
    public string Devise { get; set; }
```

5- La relation d'association

A) Principe

Comme nous l'avons déjà vu, la programmation orientée objet consiste à concevoir une application sous la forme de "briques" logicielles appelées des objets. Chaque objet joue un rôle précis et peut communiquer avec les autres objets. Les interactions entre les différents objets vont permettre à l'application de réaliser les fonctionnalités attendues.

Nous savons qu'un **objet** est une entité qui représente (modélise) un élément du domaine étudié : une voiture, un compte bancaire, un nombre complexe, une facture, etc. Un objet est toujours créé d'après un modèle, qui est appelé sa **classe**.

Sauf dans les cas les plus simples, on ne pourra pas modéliser fidèlement le domaine étudié en se contentant de concevoir une seule classe. Il faudra définir plusieurs classes et donc instancier des objets de classes différentes. Cependant, ces objets doivent être mis en relation afin de pouvoir communiquer.

B) Exemple

Dans le cas des comptes bancaires, on peut imaginer une relation d'association entre **un client** et **son compte bancaire**, car un client peut posséder plusieurs comptes bancaires. On peut également imaginer une relation d'héritage entre différents types de comptes bancaires, par exemple un compte courant et un compte épargne qui héritent tous deux des caractéristiques d'un compte bancaire général.

Voici un exemple de classe représentant un client avec une liste de ses comptes bancaires :


```

1 référence
public class Client
{
    private string nom;
    private List<CompteBancaire> comptes;

    0 références
    public Client(string nom)
    {
        this.nom = nom;
        comptes = new List<CompteBancaire>();
    }

    0 références
    public void AjouterCompte(CompteBancaire compte)
    {
        comptes.Add(compte);
    }

    0 références
    public List<CompteBancaire> Comptes
    {
        get { return comptes; }
    }
}

```

```

9 références
public class CompteBancaire
{
    private double solde;

    0 références
    public CompteBancaire(double soldeInitial)
    {
        solde = soldeInitial;
    }

    0 références
    public void Depot(double montant)
    {
        solde += montant;
    }

    0 références
    public void Retrait(double montant)
    {
        solde -= montant;
    }

    0 références
    public double Solde
    {
        get { return solde; }
    }
}

```

6- La relation d'héritage

A) Principe

L'héritage est un mécanisme objet qui consiste à définir une classe à partir d'une classe existante. Une classe héritant d'une autre classe possède les caractéristiques de la classe initiale et peut définir ses propres éléments.

La nouvelle classe (ou classe **dérivée**) correspond à une **spécialisation** de la classe de base (appelée classe **parente** ou **superclasse**). On dit que l'héritage crée une relation de type **est un** entre les classes. Dans notre exemple, un compte épargne *est un* type particulier de compte bancaire.

B) Exemple

Cette nouvelle classe peut alors ajouter de nouvelles propriétés et méthodes ou modifier celles qui ont été héritées. L'héritage permet de réutiliser le code existant et de le rendre plus modulaire et évolutif.

Dans le cas des comptes bancaires, on pourrait utiliser l'héritage pour créer une classe de compte épargne qui hérite des propriétés et méthodes de la classe

CompteBancaire existante, mais qui ajoute des fonctionnalités spécifiques telles que des taux d'intérêt et des restrictions de retrait. Par exemple :

```
1 référence
public class CompteEpargne : CompteBancaire
{
    private double tauxInteret;

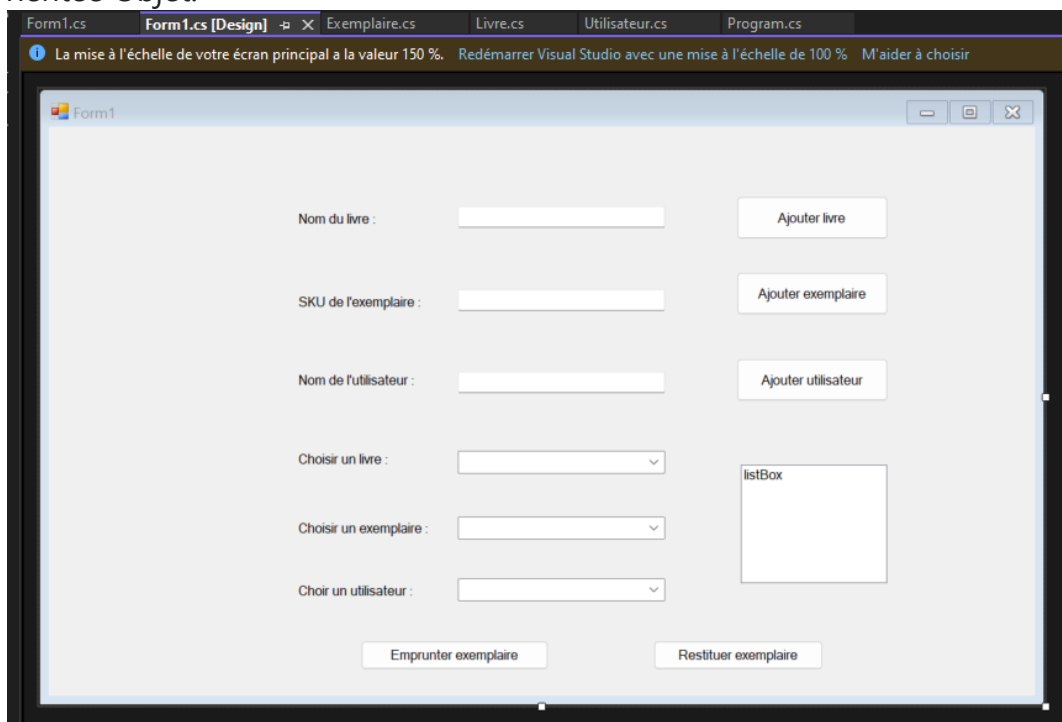
    0 références
    public CompteEpargne(double soldeInitial, double tauxInteret) : base(soldeInitial)
    {
        this.tauxInteret = tauxInteret;
    }

    0 références
    public void CalculerInteret()
    {
        double interet = Solde * tauxInteret;
        Depot(interet);
    }
}
```

7- Projet Bibliothèque

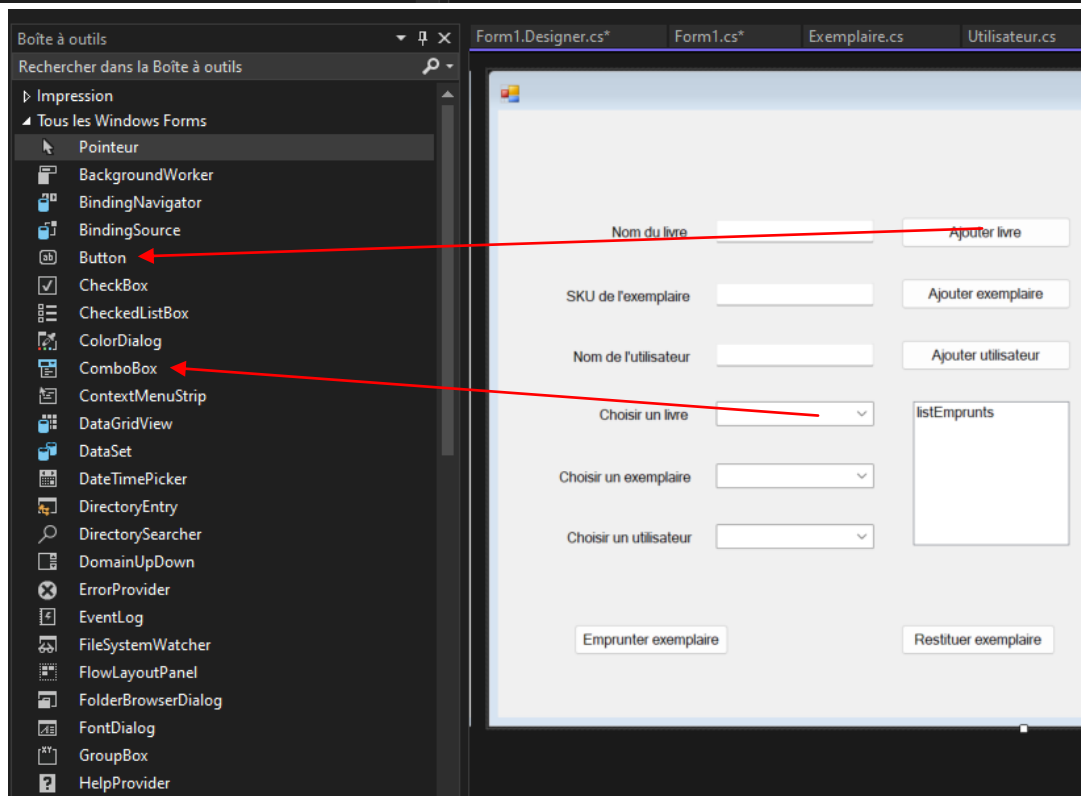
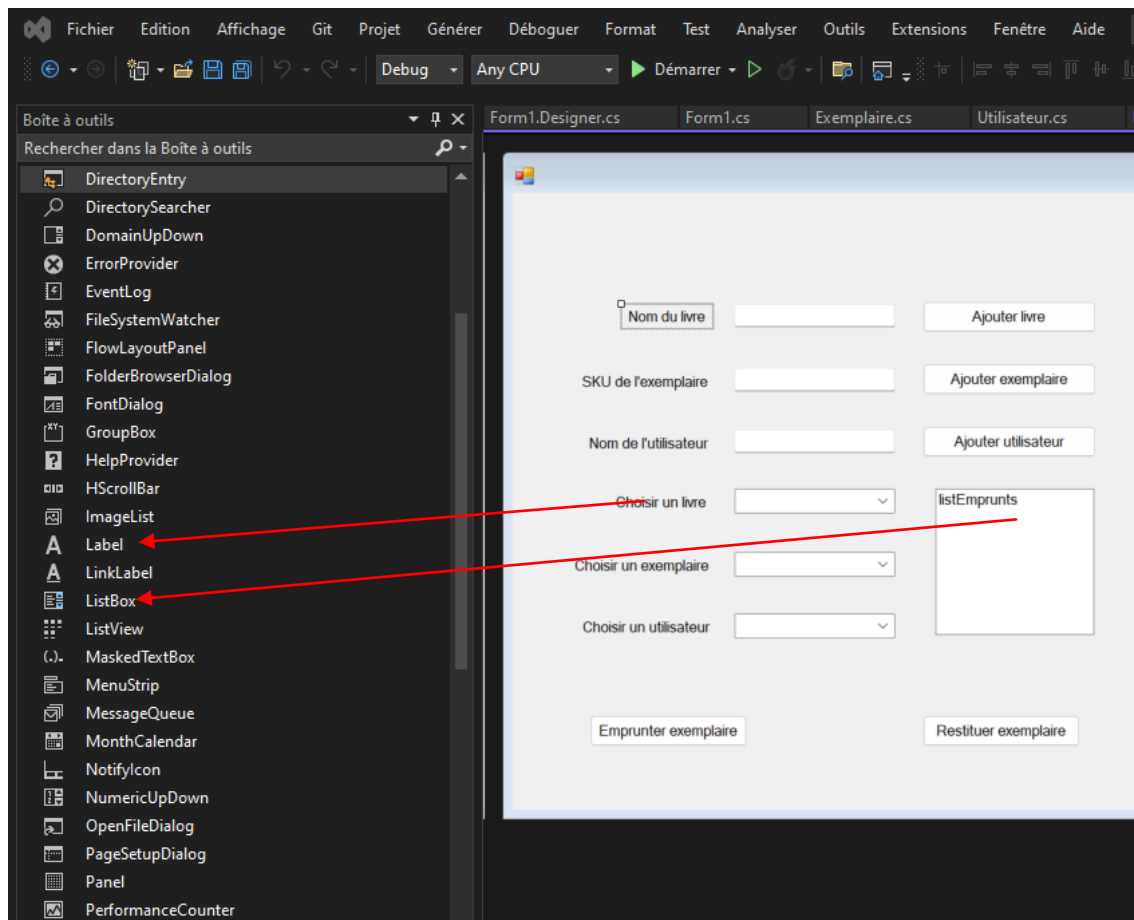
A) Les objets

A présent, nous allons détailler un projet qui regroupe tous les points de connaissances vu précédemment. Il s'agit du projet Bibliothèque en Programmation Orientée Objet.

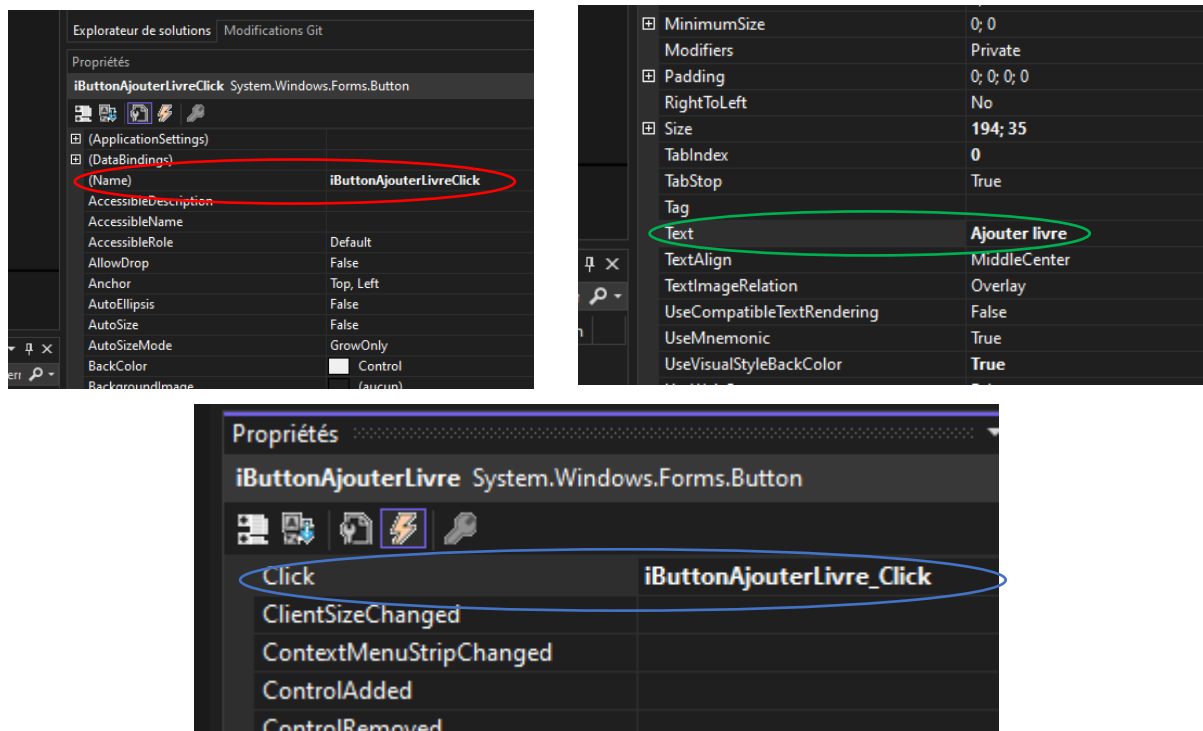


Nous avons donc commencé par aller chercher tous les objets dans la boîte à outils situé à droite de l'écran. (Si celle-ci n'apparaît pas, la fonctionnalité se trouve dans

affichage puis boîte à outil. Le raccourci clavier Ctrl + W, X permet également son affichage). Ensuite il ne nous reste plus qu'à sélectionner ce dont nous avons besoin.



Une fois les objets en place. Nous pouvons modifier les noms des objets (cercle rouge), leur contenu d'affichage (cercle vert) ainsi ainsi que les fonctionnalités lors du clique (cercle bleu) dans les propriétés de ceux-ci.



B) Le codage

Une fois nos éléments en place, nous avons la possibilité de modifier leurs actions (=ceux que vont faire les objets) lorsque l'on double clique dessus.

Commençons par le premier bouton : 'iButtonAjouterLivre_Clik'.

```
1 référence
private void iButtonAjouterLivre_Click(object sender, EventArgs e)
{
    // Ici on ajoute une instance de livre directement dans
    // notre tableau, on ajoute en incrémentant notre compteur
    if (textBoxTitreLivre.Text != "")
    {
        Livre tmpLivre = new Livre(textBoxTitreLivre.Text);
        mesLivres[nbLivres++] = new Livre(textBoxTitreLivre.Text);
        MessageBox.Show("Nous venons d'ajouter le livre : " + textBoxTitreLivre.Text + "!", "SUCCES");
        comboBoxLivre.Items.Add(tmpLivre);
    }
    else
    {
        MessageBox.Show("Merci de préciser un titre !", "Erreur !");
    }
    //Livre livre1 = new Livre(textBoxTitreLivre.Text);
    //livre1.afficherInfo1();
}
```

D'après ce code pour cet objet, nous ajoutons donc dans l'objet 'comboBoxLivre' (cadre orange) le livre que nous avons instancier dans l'objet 'textBoxTitreLivre' (cadre violet).

Il en va de même pour les exemplaires :

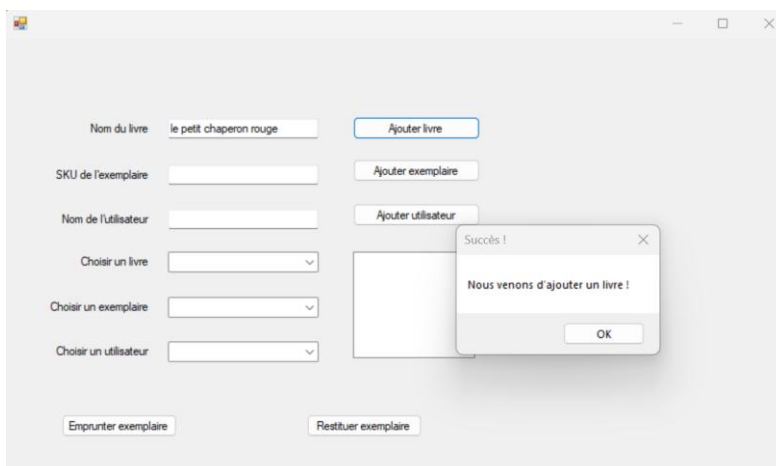
```
1 référence
private void iButtonAjouterExemplaireClick_Click(object sender, EventArgs e)
{
    if (!string.IsNullOrWhiteSpace(iTextBoxSkuAjouterExemplaire.Text))
    {
        Livre livreSelectionne = (Livre)comboBoxLivre.SelectedItem;
        Console.WriteLine(livreSelectionne);
        // Ajouter un exemplaire à son tableau d'exemplaires
        if (livreSelectionne != null)
        {
            Exemplaire exemplaire = new Exemplaire(livreSelectionne, iTextBoxSkuAjouterExemplaire.Text);
            Console.WriteLine(livreSelectionne.Exemplaires.Contains(exemplaire));
            if (livreSelectionne.Exemplaires.Contains(exemplaire))
            {
                MessageBox.Show("cet exemplaire existe déjà !", "attention");
            }
            else
            {
                livreSelectionne.AjouterExemplaire(exemplaire);
                MessageBox.Show("nous venons d'ajouter un exemplaire !", "succès !");
            }
        }
        else
        {
            MessageBox.Show("selectionner un livre !", "attention !");
        }
    }
    else
    {
        MessageBox.Show("merci de préciser un exemplaire !", "attention !");
    }
}
```

Ce code plus performant, vérifie si un nom a bien été donné, si l'exemplaire existe déjà et si un livre est bien associé à cet exemplaire. Nous allons rentrer dans les détails de ce code dans le point suivant.

C) Les relations d'associations

Maintenant les exemplaires. Pour eux cela va être différents. Il ne vont pas se contenter d'être créés puis mis dans une liste. Ils vont en fait être associés à un objet livre.

Prenons l'exemple de l'image ci-dessous. Nous instancions un livre 'Le petit chaperon rouge'.



Nousinstancions également un exemplaire 1 : « ex1 » :

The screenshot shows a web application interface for managing books and their examples. The form contains the following fields and buttons:

- Nom du livre:
- SKU de l'exemplaire:
- Nom de l'utilisateur:
- Choisir un livre:
- Choisir un exemplaire:
- Choisir un utilisateur:
-

A success dialog box is displayed over the form with the text: "succès ! nous venons d'ajouter un exemplaire !" and an "OK" button.

Notre exemplaire « ex1 » du livre « le petit chaperon rouge » est bien présent :

The screenshot shows the same web application interface as before, but the 'Choisir un exemplaire' dropdown now displays 'le petit chaperon rouge - ex'. The other fields and buttons remain the same.

Maintenant si nousinstancions un nouveau livre « Les 3 Mousquetaires », alors l'exemplaire « ex1 » n'existe pas pour ce livre.

The screenshot shows the web application interface for adding a new book. The 'Nom du livre' field is filled with 'Les trois Mousquetaires'. The other fields and buttons are the same as in the previous screenshots.

RECAP : Lorsqu'un livre est instancier, l'exemplaire est uniquement associé au livre en question (dans notre exemple c'était 'le petit chaperon rouge'). Si l'on prend maintenant un nouveau livre l'exemplaire n'existe plus puisque l'exemplaire est unique, et associé à un et un seul livre !