Beuscart Benjamin

Lupa Valentin

Malbranque Louis

Verpoort Alexia

Découverte

Du

C#

Sommaire

I/ Introduction au C#:

Le C# est un langage créé en 2002 par Microsoft.

Il utilise le Frameworks.net et utilise donc la combinaison entre les frameworks et les spécificités du langage pour créer une application. Le frameworks.net étant un composant microsoft intégral prenant en charge la création et l'exécution d'applications et de services web. Il est basé sur le langage CIL (Common Intermediate Langage) et ne dépend donc pas du langage de programmation. Tous les langages compatibles ont donc accès aux bibliothèques installables dans l'environnement d'exécution. Le but étant de faciliter le développement des applications.

Bien qu'il existe des similitudes avec d'autres langages comme le C et le C++, le C# possède des avantages. En comparaison avec le C, il permet une meilleure comptabilité car l'IDE transforme le code en langage CIL ou MSIL plutôt qu'en binaire. Il est également plus simple que le C++.

Le C# est un langage programmé objet mais possède davantage de features que le langage Java par exemple. C'est également un langage orienté composant. C'est-à-dire une approche plus modulaire de l'architecture d'un projet pour obtenir une meilleure lisibilité et une meilleure maintenance.

Le C# est également un langage intégrant des fonctionnalités permettant d'obtenir des applications fiables. Il intègre en effet le **type-safe** permettant de vérifier qu'il n'y a pas d'erreurs propres aux variables (mauvaise indexation de tableaux, mauvaise opération sur un type…). Il intègre également la **Gestion des exceptions** pour permettre une récupération efficace des erreurs générées et enfin le **garbage Collection** qui permet de récupérer la mémoire occupée inutilement.

II/ Les bases du C# :

1) Rappels et Prérequis

Bon nombres de concepts sont similaires sous C# et sous de nombreux langages informatiques.Pour n'en citer que les principaux qui ne feront pas l'objet d'un développement, il y a par exemple les variables de bases : les entiers (**int**), les flottants (**float**), les booléens (**boolean**), les caractères (**char**) ou encore les chaines de caractères (**string**). Il en va de même pour l'opérateur d'affectation (Var1 **=** Var2) ayant un comportement similaire en java ou en C et permettant d'affecter une valeur à une variable. Pour finir le tour des prérequis, les opérations relationnelles (**!=, ==, <=, >= , <, >**), sont identiques au C, et permettent de comparer deux expressions ou deux variables, le résultat retourné sera donc vrai (**true**) ou faux (**false**). Les opérateurs booléens ET (**&&**) et OU (**||**) sont également identiques et permettent de générer une condition plus complexe.

Ce tutoriel n'abordera pas dans le détail les boucles ou les structures conditionnelles. Il est cependant important de rappeler la syntaxe des boucles de ce langage comme certaines différences peuvent être observées

- ***If/Else***

  Si une condition est vérifiée alors une instruction est exécutée, sinon une autre est exécutée.

```
Int var1=6 ;

if(3>5 && var1>3)
{
Console.WriteLine(« 3 ne peut être supérieur à 5 ») ;
}
else if(3>5 || var1>3)
Console.WriteLine(« 3 n'est pas supérieur à 5 mais var1>3 ») ;
}
else
{
Console.WriteLine(« 3 n'est pas supérieur à 5 et var1<3 ») ;
}
```

  Le résultat ici sera donc « 3 n'est pas supérieur à 5 mais var1>3 ».

- ***Switch Case***

  Le Switch permet en fonction de la valeur d'une variable d'effectuer telle ou telle opération.

```
Int var1=1 ;

switch(var1){
case(0) : Console.WriteLine(« var1 égale à 0 ») ;
case(1) : Console.WriteLine(« var1 égale à 1 ») ;
default : Console.WriteLine(« La valeur de var1 n'existe pas dans la condition ») ;
}
```

  Default peut être apparenté à else dans la structure du if. Il est exécuté lorsqu'un autre cas ne correspond.

  Ici, le programme renverra « var1 égale à 1 »

- ***Instruction for et instruction foreach***

  On effectue la boucle autant de fois qu'il est précisé. La structure for dépend en règle générale d'un entier qui est automatiquement incrémenté à chaque début de boucle et comparé pour savoir s'il faut continuer l'itération.

  La structure foreach prend en argument une collection d'objets énumérables, et une variable qui prend tout à tour chaque élément de la collection. Le nombre d'itérations dépend donc du nombre d'éléments dans la collection.

```
Var1=0;
for(int i=0 ; i<9 ; i++)
{
 Var1++;
}
```

  Dans cet exemple, la boucle est exécutée 9 fois et la valeur finale de Var1 est 9.

```
int[] joursAnniv={3,10,25,27};
int aujourdhui=10 ;
foreach(int jour in joursAnniv)
{
   If(aujourdhui == jour){
   Console.WriteLine(jour+ « est un jour d'anniversaire ») ;
}
}
```

La console affichera ici « 10 est un jour d'anniversaire », mais la boucle a bien été exécutée 4 fois.

- ***Instruction while et do while***

On effectue la boucle tant que la condition est vraie. La boucle **do…While** sera exécutée au moins une fois, tandis que la boucle **while** ne sera pas exécutée si la condition est immédiatement fausse.

```
var1=2 ;
var2=0 ;
while(var2<var1){
var2++ ;
}
```

```
var1=2 ;
var2=0 ;
do{
var2++ ;
} while(var2<var1);
```

La boucle ici est exécutée 2 fois.

2) *Les Flux d'entrée-sortie*

Avant d'entrer dans les détails plus techniques, il est important de différencier la syntaxe par rapport à d'autres langage permettant à l'utilisateur d'entrer des données ou au programme d'afficher une ligne dans un programme console, ce qui est fondamental afin d'avoir une interaction humaine avec le programme pour qu'il soit utilisable par tous sans nécessité d'ouvrir le code source.

a) Le flux de sortie

Le C#, comme le java, étant un langage orienté objet, les fonctions permettant l'affichage d'une donnée dans la console existe déjà.

Console.WriteLine(« Cette ligne sera affichée dans la console ») ;

Comme dans de nombreux langages, le ; est obligatoire à la fin d'une ligne d'instruction.

a) Le flux d'entrée

Concernant le flux d'entrée, il faut utiliser l'instruction suivante :

```
Int entier = Console.ReadLine() ;
```

Pour le flux d'entrée, il faut tout d'abord créer une variable en choisissant son type puis lui attribuer le résultat de l'instruction Console.ReadLine() ;

3) Les espaces de nom

Avant d'aborder la vraie programmation C#, nous allons faire un point sur les espaces de nom qui ne sont pas présents dans tous les langages et qui sont parfois présents dans les exemples qui suivent.

Un espace de nom regroupe plusieurs déclarations sous un même nom. Ces déclarations pouvant être des classes, interfaces…

On utilise le mot clé **namespace.**

```
namespace EspaceDeNom{
    public class premiereClasse(){
    }
}
```

Pour se référer aux méthodes à l'intérieur de chaque espace de nom, on fait l'appel suivant : EspaceDeNom.premiereClasse(). Un des espaces de nom le plus connu étant System, pour utiliser la console, la formulation exacte étant System.Console.Write(« »). Pour éviter d'utiliser continuellement l'espace de nom, le mot-clé **using** au début du programme permet d'écrire par exemple Console.Write(« ») grâce à un using System.

L'utilité des espaces de noms étant principalement l'organisation des projets grâce à des appellations communes.

4) Types en C#

La déclaration d'une variable s'effectue en déclarant le type de la variable directement (float, int…) ou en utilisant var qui permet au compilateur de déduire. Une variable déclarée ne peut pas être assigné à une variable d'un autre type sans utiliser le type-casting qui est une conversion explicite d'un type. Certains cas entrainent cependant une conversion implicite s'il n'y a pas de perte de donnée et cela est effectué par le compilateur.

Exemple de cast :
```
int a=10 ;
double b ;
b=(double)a ;
```

Le C# intègre le principe de **types unifiés**, c'est-à-dire que tous les types héritent du type object et donc il y a un partage d'opérations communes entre les types.

En C# on trouve deux grands types différents, les types valeurs et les types références.

L'utilisateur peut créer ses propres types à l'aide des constructions struct, classe, interface, enum.

a) Le type Valeur

Une variable de type valeur contient directement la donnée. Une opération sur une variable n'influence pas une autre variable de même type. Le type valeur regroupe les types simples, les struct et les enum. C'est un type scellé, il ne prend donc pas en charge l'héritage, c'est-à-dire qu'aucune classe ne peut hériter d'un struct par exemple.

➔ Les structures

Les structures peuvent contenir des données et des fonctions membres. Il s'agit d'un conteneur de variables liées.

```
struct cercle
    {
        public int rayon, centrex, centrey;
        public cercle(int rayon, int centrex, int centrey)
        {
            this.rayon = rayon;
            this.centrex = centrex;
            this.centrey = centrey;
        }
    }
```

Sur cet exemple nous définissons une structure de type cercle qui contient 3 variables : son rayon et les coordonnées selon x et y de son centre.

➔ Les énumérations

Une énumération sont des ensembles de constantes. Cela permet de définir un type avec des valeurs discrètes. Par exemple, pour une liste de 3 boissons, on peut créer une énumération de ces trois boissons, puisqu'il n'y en aura pas une infinité plutôt que de créer une classe Boisson et d'en instancier 3 différentes.

```
enum Vetement
    {
        Short,
        Jupe,
        Jean,
        Pull
           }
```

➔ Retour sur le type Valeur

D'après ce qui a été dit plus haut, le type Valeur contient directement la donnée, voyons ce que cela entraine sur notre exemple de structure.

```
cercle cercle1 = new cercle(2, 1, 1);
cercle cercle2 = cercle1;
cercle1.rayon = 1;
Console.WriteLine("cercle1 : Mon rayon est de " + cercle1.rayon);
Console.WriteLine("cercle2 : Mon rayon est de " + cercle2.rayon);
Console.ReadKey();
```

```
cercle1 : Mon rayon est de 1
cercle2 : Mon rayon est de 2
```

On observe bien que la modification du rayon du cercle1 n'a pas modifié le rayon du cercle 2.

b) Le type Référence

Une variable de type référence contient la référence à leur donnée qui est appelé objet. Une opération sur un variable peut influencer une autre variable de même type. Les types références sont des classes, interfaces, délégué ou tableaux. Ce type est initialisé à null et nécessite la création grâce à l'objet new.

➔ Les classes

Les classes sont des structures de données améliorées contenant à la fois les données et les méthodes liées.

```
public class cercle
    {
        public int rayon, centrex, centrey;
        public cercle(int rayon, int centrex, int centrey)
        {
            this.rayon = rayon;
            this.centrex = centrex;
            this.centrey = centrey;
        }
    }
```

➔ Les interfaces

Une interface ressemble à une classe de type abstraite. Elle ne peut être instanciée directement, c'est-à-dire qu'on ne peut créer un objet directement du type de l'interface. Les interfaces ne contiennent pas d'implémentation des méthodes.

Une classe peut implémenter plusieurs interfaces.

```
interface GérerFigure
{
    void Dessiner();
}
```

Cette interface pourrait être implémentée par la classe cercle créée plus haut.

➔ Les tableaux

Comme en C, les tableaux sont indexés sur zéro, avec n éléments allant de 0 à n-1. Les tableaux sont de tous types y compris du type tableau. Les valeurs par défaut des éléments de tableau sont définies sur zéro et les éléments de référence sont définis sur Null. Un tableau peut être unidimensionnel, multidimensionnel comme en C ou encore en escalier. La particularité de l'escalier est que les éléments du tableau principal n'ont pas les mêmes dimensions entre eux :

```
cercle[][] a = new cercle[3][];
a[0] = new cercle[3];
a[1] = new cercle[5];
a[2] = new cercle[1];
```

Tableau en escalier

➔ Les délégués

Les délégués sont des variables qui pointent vers des méthodes. Ils sont déclarés avec **delegate** et permettent la mise en place du principe d'ouvert/fermé, autrement dit, ils permettent de passer des méthodes comme argument lors des appels. Ils sont en quelque sorte un moyen de modifier le comportement d'une application sans modifier son code source et donc de réorganiser le code.
Leur effet est lié à celui de la catégorie de classes **abstract**.
Les délégués peuvent être déclarés dans la partie **namespace** ou alors au sein même d'une classe.

Il existe plusieurs méthodes permettant d'instancier un délégué, on peut passer par l'instanciation d'un délégué pointant vers une méthode qui respecte la signature du délégué, c'est-à-dire qu'un délégué prenant en argument un entier doit pointer vers une méthode qui prend en compte un entier également.

```
delegate string ActionOnString(string text);
 ActionOnString test1 = affichageClassique;
```

Dans l'exemple ci-dessus, affichageClassique doit prendre un string en argument et retourner un string pour respecter la signature du délégué et ainsi permettre à test1 de pointer vers affichageClassique.

On peut également passer par l'utilisation du mot clé **delegate** qui permet également d'instancier un délégué avec une méthode quelconque.

Il est également possible d'utiliser les expressions lambda pour instancier un délégué. Elles permettent notamment de simplifier l'écriture des délégués.

```
ActionOnString test3 = s => s + s;
```

Dans cet exemple, ce qu'il y a à gauche de la flèche après test3, correspond à l'argument du délégué. Le s correspond à un string, puisque le délégué ActionOnString prend en argument un string. Ce qu'il y a à droite, correspond à ce que retourne le délégué, on retourne donc une concaténation du même string. La signature est donc respectée, puisqu'on retourne également un string.

Exemple pratique permettant d'illustrer l'utilisation des délégués :

```
namespace UsingDelegate
{
  //Creation d'un délégué qui prend en paramètre un string et renvoie un string
  delegate string ActionOnString(string text);

  class Program
  {


    static string affichageClassique(string text)
    {
      return text;
    }

    static void Main(string[] args)
    {
      //Instancie un délégué pointant vers affichageClassique
      ActionOnString test1 = affichageClassique;

      //appel du délégué test1
      //affichageClassique
      string result1 = test1("Première phrase de test.");
      Console.WriteLine(result1 + "\n");

      //Instancie un délégué vers une méthode quelconque
      //All in uppercase
      ActionOnString test2 = delegate (string text)
      {
        string tmp = text.ToUpper();
        return tmp;
      };

      string result2 = test2("Deuxième phrase de test.");
      Console.WriteLine(result2 + "\n");

      //Instancie un délégué avec une expression lambda
      //qui concatène deux fois le même string
      ActionOnString test3 = s => s + s;
      string result3 = test3("Troisième phrase de test.");
      Console.WriteLine(result3 + "\n");

      Console.ReadKey();
    }
  }
```

Il existe également des délégués dits génériques qui sont les délégués Action et Func.

Action est un délégué qui permet de pointer vers une méthode qui ne renvoie rien et prend en argument un tableau de 16 types maximum.

Func est le délégué par défaut pour pointer vers un délégué qui retourne quelque chose.

➔ Retour sur le type Référence

D'après ce qui a été dit plus haut, le type Référence contient la référence de la donnée et non pas directement la donnée, voyons ce que cela entraine sur notre exemple de classe.

```
cercle cercle1 = new cercle(2, 1, 1);
cercle cercle2 = cercle1;
cercle1.rayon = 1;
Console.WriteLine("cercle1 : Mon rayon est de " + cercle1.rayon);
Console.WriteLine("cercle2 : Mon rayon est de " + cercle2.rayon);
Console.ReadKey();
```

```
cercle1 : Mon rayon est de 1
cercle2 : Mon rayon est de 1
```

La modification d'un champ de cercle1 a également modifié celui de cercle2 car leur référence est identique.

c) *Le type Générique*

En C#, il existe également un type appelé type Générique. Ce sont des types non spécifiés à l'implémentation et qui seront déterminé par le compilateur lors de l'utilisation des méthodes. L'utilité des génériques est donc la réutilisation du code et également l'amélioration de ses performances.

On déclare un type générique grâce à la syntaxe suivante : <nomVariable>. Les génériques peuvent être utilisés pour les classes, les méthodes ou encore les structures. Cette construction est notamment utilisée couramment lors de l'utilisation des listes : List<int>, la classe liste étant générique.

L'utilisation d'un générique peut être utile dans les méthodes qui prennent en argument un type object par exemple.

Voici un exemple très simple d'une méthode générique :

```
Public void verificationSiNul<nb>(nb nombre){
        If(nombre==0){
            Console.Write(« Le nombre vaut 0 ») ;
        }
}
```

Cette méthode peut donc être appelée avec des int, des floats, des doubles... Selon le type de nombre, une nouvelle version de la méthode sera créée.

On pourrait donc faire l'appel suivant :

```
Int chiffre=8 ;
verificationSiNul(chiffre) ;
```

On aura donc l'appel de la fonction pour des entiers.

Concernant les classes, l'appel d'une classe générique requiert la spécification du type, comme la liste. Pour appeler une classe générique, on procédera ainsi :

```
classeGen<int> appelclasse= new classeGen<int>
```

5) Utilisation de la programmation objet

La manière de gérer la programmation objet est semblable aux langages tels que le C++ ou le java, la majeure différence avec ce dernier étant par exemple la syntaxe.

Pour commencer, lors de la création d'une classe, l'héritage par rapport à une classe mère se déroule de la sorte :

```
public class Salade : Aliment
```

Ici notre classe Salade dérive de la classe aliment, elle détient donc les mêmes caractéristiques que la classe Aliment. Supposons que la classe aliment soit défini de la manière suivante :

```
public class Aliment
{
    protected string nom;
    protected int prix;
    protected int calorie;
    protected Boolean estVegetarien;
    3 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public Aliment(string nom, int prix, int calorie, Boolean estVegetarien)
    {
        this.nom = nom;
        this.prix = prix;
        this.calorie = calorie;
        this.estVegetarien = estVegetarien;
    }

    7 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public virtual void descriptionAliment()
    {
        Console.WriteLine("Cet aliment est " + nom + " il coute " + prix + " et contient " + calorie + " calories");
        if (estVegetarien == true)
        {
            Console.WriteLine("Il convient aux végétariens");
        }
        else
        {
            Console.WriteLine("Il ne convient pas aux végétariens");
        }
    }
}
```

La classe Salade possèdera donc également les variables nom, prix… et aura accès au constructeur et à la méthode descriptionAliment.

Pour utiliser le constructeur ou les méthodes, on passe par l'utilisation du mot-clé **base**. Ce mot clé permet notamment d'overrider les méthodes.

```
public class Salade : Aliment
{
    1 référence | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public Salade()
        : base("Salade", 1, 3, true)
    {
    }
}
```

Ici Salade a un constructeur qui reprend les précédents éléments du constructeur de Aliment en prenant pour argument « Salade » pour le nom, 1 pour le prix…

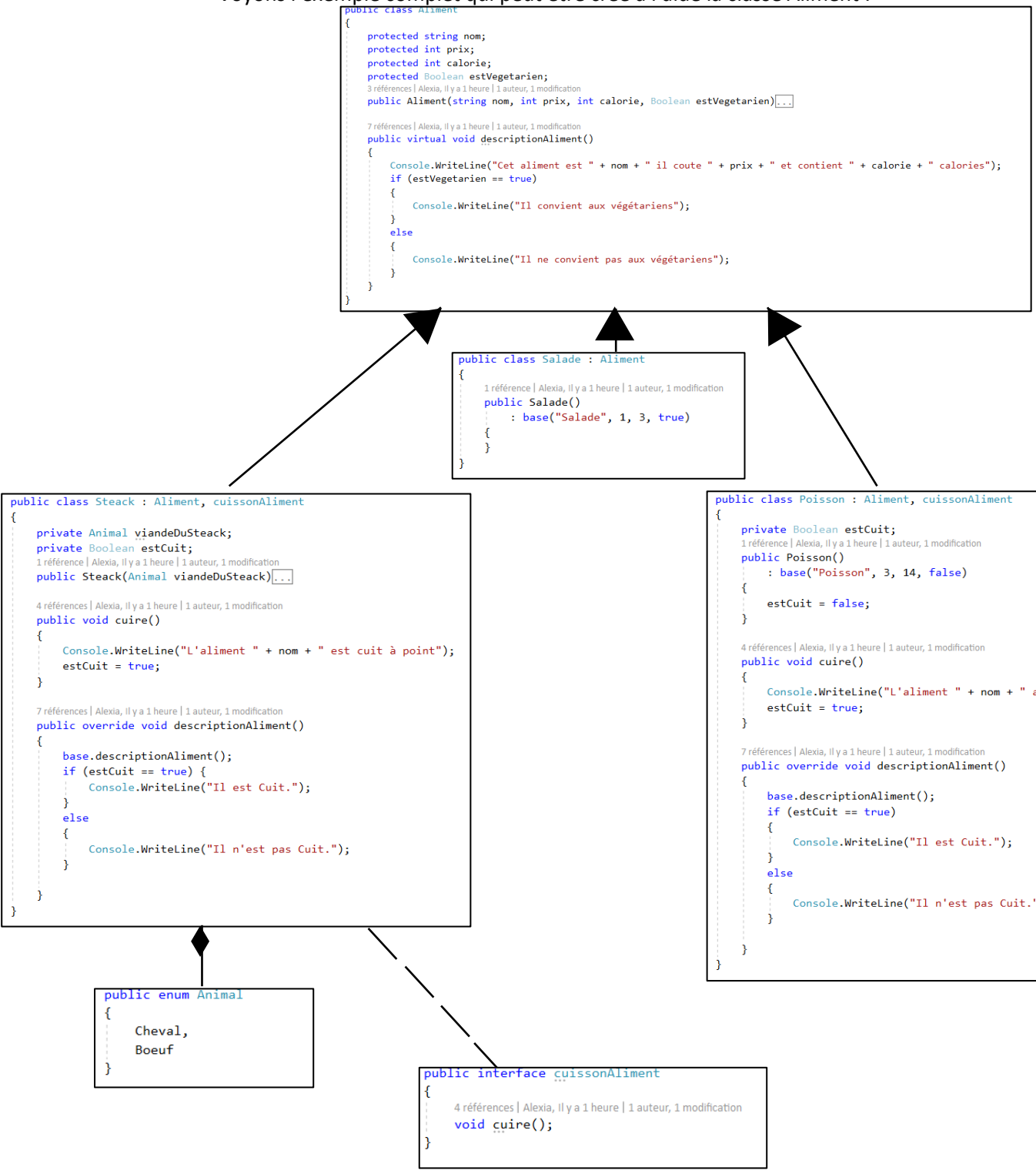
On peut ainsi créer plusieurs classes qui pourront potentiellement avoir chacune leurs caractéristiques mais qui posséderont chacune les caractéristiques de la classe aliment.

Il est également possible de créer une interface qui sera implémentée par les classes de la même manière que l'héritage. Les classes qui l'implémenteront devront alors implémenter les méthodes déclarées.

```
public interface cuissonAliment
{
    4 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    void cuire();
}
```

Cette interface pourrait être implémentée par un Aliment qui peut être cuit, ce n'est donc pas nécessaire que Salade implémente cette interface.

Voyons l'exemple complet qui peut être créé à l'aide la classe Aliment :



```
public class Aliment
{
    protected string nom;
    protected int prix;
    protected int calorie;
    protected Boolean estVegetarien;
    3 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public Aliment(string nom, int prix, int calorie, Boolean estVegetarien)...

    7 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public virtual void descriptionAliment()
    {
        Console.WriteLine("Cet aliment est " + nom + " il coute " + prix + " et contient " + calorie + " calories");
        if (estVegetarien == true)
        {
            Console.WriteLine("Il convient aux végétariens");
        }
        else
        {
            Console.WriteLine("Il ne convient pas aux végétariens");
        }
    }
}
```

```
public class Salade : Aliment
{
    1 référence | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public Salade()
        : base("Salade", 1, 3, true)
    {
    }
}
```

```
public class Steack : Aliment, cuissonAliment
{
    private Animal viandeDuSteack;
    private Boolean estCuit;
    1 référence | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public Steack(Animal viandeDuSteack)...

    4 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public void cuire()
    {
        Console.WriteLine("L'aliment " + nom + " est cuit à point");
        estCuit = true;
    }

    7 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public override void descriptionAliment()
    {
        base.descriptionAliment();
        if (estCuit == true) {
            Console.WriteLine("Il est Cuit.");
        }
        else
        {
            Console.WriteLine("Il n'est pas Cuit.");
        }

    }
}
```

```
public class Poisson : Aliment, cuissonAliment
{
    private Boolean estCuit;
    1 référence | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public Poisson()
        : base("Poisson", 3, 14, false)
    {
        estCuit = false;
    }

    4 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public void cuire()
    {
        Console.WriteLine("L'aliment " + nom + " a été cuit sur plancha");
        estCuit = true;
    }

    7 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    public override void descriptionAliment()
    {
        base.descriptionAliment();
        if (estCuit == true)
        {
            Console.WriteLine("Il est Cuit.");
        }
        else
        {
            Console.WriteLine("Il n'est pas Cuit.");
        }

    }
}
```

```
public enum Animal
{
    Cheval,
    Boeuf
}
```

```
public interface cuissonAliment
{
    4 références | Alexia, Il y a 1 heure | 1 auteur, 1 modification
    void cuire();
}
```

L'exemple précédent nous permet d'obtenir le résultat console suivant :

```
static void Main(string[] args)
{
    Salade salade = new Salade();
    salade.descriptionAliment();
    Steack steack = new Steack(Animal.Cheval);
    steack.descriptionAliment();
    steack.cuire();
    steack.descriptionAliment();
    Poisson poisson = new Poisson();
    poisson.cuire();
    Console.ReadKey();
}
```

```
Cet aliment est Salade il coute 1 et contient 3 calories
Il convient aux végétariens
Cet aliment est Steack il coute 5 et contient 25 calories
Il ne convient pas aux végétariens
Il n'est pas Cuit.
L'aliment Steack est cuit à point
Cet aliment est Steack il coute 5 et contient 25 calories
Il ne convient pas aux végétariens
Il est Cuit.
L'aliment Poisson a été cuit sur plancha
```

6) Gestion des exceptions

En C #, il faut être capable de savoir gérer les exceptions ou erreurs provoquées par une fonction ou une méthode. Comme dans de nombreux langages, certaines sont traitées automatiquement, c'est le cas lorsqu'il y a une erreur de types par exemple, et un message est affiché pour connaitre la source de l'erreur. Cependant, comme le java est un langage orienté objet, la création de certaines méthodes dans des classes sont susceptibles de générer des exceptions pour que celle-ci s'exécute telle qu'elle l'a été pensée, mais être correcte syntaxiquement.

Un exemple simple pourrait être un appel à l'utilisateur pour que celui-ci entre un entier et qu'il entre un nombre entier négatif. Techniquement c'est correct, la gestion automatique n'y verra pas de problème. Mais le traitement futur pourrait poser problème.

La structure **try…catch** permet de remédier à ce problème. La partie try contiendra la partie du code susceptible de faire planter le programme, c'est donc ici qu'on aura dans notre exemple la réception de l'entrée utilisateur. Catch permettra de traiter l'exception, s'il y en a sinon on passera à l'instruction suivante.

Catch prend en argument une Exception, qui peut être donc toute sorte d'erreur que le compilateur peut identifier ou bien des erreurs spécifiques, les exceptions pour la division par zéro : **DivideByZeroException** ou pour un problème d'indice : **IndexOutOfRangeException**, peuvent être utilisé directement. Catch(DivideByZeroException). L'utilisateur peut également créer ses propres exceptions comme ce serait le cas dans notre exemple, notamment avec l'utilisation du mot-clé **throw.**

Nous avons donc créé un exemple permettant de créer sa propre levée d'exception avec l'utilisateur qui doit entrer un entier positif.

```
static void Main(string[] args)
        {
            string entier;
            int entier2;
            Program essai = new Program();
            Console.WriteLine("Entrez un entier");
            entier = Console.ReadLine();
            entier2 = int.Parse(entier);
            essai.test(entier2);

            Console.WriteLine("Entrez un entier");
            entier = Console.ReadLine();
            entier2 = int.Parse(entier);
            essai.test(entier2) ;
            Console.ReadKey();
        }
        public void test(int entier)
        {
            try
            {
                if (entier < 0)
                {
                    throw (new EntierIsNegativeException("Entier est negatif"));
                }
                else {
                    Console.WriteLine("entier est positif"); }
                 Console.WriteLine("Essai termine\n");
            }
            catch(EntierIsNegativeException e)
            {
                Console.WriteLine(e.Message);

            }
        }
        public class EntierIsNegativeException : SystemException
        {
            public EntierIsNegativeException(string message) : base(message)
            {
            }
        }
```

On utilise donc un try…catch dans la méthode test pour vérifier le signe du nombre entier. Cependant l'exception en argument dans le catch n'est pas une exception définie comme préexistante comme la division par zéro. Et sans définir l'exception, le compilateur ne verrai pas ici une levée d'exception mais une simple condition du if vraie. Nous avons donc créé la classe EntierIsNegativeException qui hérite de la classe Exception et qui est notre exception personnalisée. Lorsque l'entier est négatif, à l'ide du mot-clé throw, on déclare qu'une exception a été trouvée, ce qui nous permet de passer dans le catch et quel message la caractérise.

La sortie console de ce programme est la suivante :

```
Entrez un entier
2
entier est positif
Essai termine

Entrez un entier
-3
Entier est negatif
```

Notre exception a bien été levée lorsque le programme a été lancé avec un nombre négatif en argument. On observe également que lorsque l'exception a été levée, la fin du try a bien été passée pour entrer dans le catch, essai terminé ne s'affichera donc pas.

III/ Concepts Clés du C# :

1) La gestion des évènements

Les évènements permettent à un objet qui est appelé publieur d'avertir à d'autres classes, les abonnés lorsqu'un événement important se produit.

Le choix du déclenchement est effectué par le publieur, c'est-à-dire que le publieur choisit quelle action génère un événement et les abonnés choisissent que faire à la réception de l'événement.

Un évènement nécessite un abonné pour pourvoir être déclenché.

Ce sont les event handler qui permettent d'avertir lorsque l'évènement a été déclenché en l'attachant à un évènement.

Pour utiliser un évènement, on passe généralement par l'utilisation du type délégué.

Méthode d'utilisation d'un évènement pas à pas avec un exemple concret :

- On commence par déclarer un délégué, dans l'exemple complet il prendra en argument un string.
```
public delegate void delegue1(string texte);
```

- On stocke ensuite ce délégué dans un event
```
public event delegue1 changementTexte;
```

- Il faut alors s'abonner à l'évènement en créant une méthode respectant la signature de l'évènement.
  La méthode associée ici à notre changement de texte serait la suivante :

```
private void changement_texte_de_presentation(string text)
{
    Console.WriteLine("Le texte a change");
    Console.Write(text);
}
```

Pour réaliser l'abonnement, on fait pointer le délégué vers cette méthode puis à l'aide de l'opérateur =+ on abonne le délégué à l'évènement.

```
delegue1 delegue=changement_texte_de_presentation;
changementTexte += delegue;
```

- Il reste encore à créer une méthode appelant le event handler.

Le but de notre exemple ici, est de s'abonner à un changement de texte. Nous avons donc bien créé un délégué et associé à ce délégué un évènement. La méthode exemple permet de s'abonner à l'évènement. Lorsque le texte est modifié grâce à la méthode modification, l'event handler est déclenché et appelle la méthode changement_texte_presentation qui possède la même signature que le délégué qui a été associé.

Voici l'exemple complet qui nous a permis de réaliser notre exemple pas à pas :

```
namespace ExEvent
{
   class Program
   {
      static void Main(string[] args)
      {
         Exemple exemple = new Exemple();
         exemple.exemple();
         exemple.Modification("Bienvenue au cours de C#");
         Console.ReadKey();
      }
   }
}
public class Exemple
{
   public string texte = "Bienvenue à tous";
   public delegate void delegue1(string texte);
   public event delegue1 changementTexte;
   public Exemple()
   {
      Console.WriteLine(texte);
      exemple()
   }
   public void exemple() {
      delegue1 delegue=changement_texte_de_presentation;
      changementTexte += delegue;
   }
   private void changement_texte_de_presentation(string text)
      {
         Console.WriteLine("Le texte a change");
         Console.Write(text);
            }
   public void Modification(string text)
   {
      texte = text;
      changementTexte(text);
   }
}
```

2) Les attributs

A la base une entité peut être qualifée de private, public, protected ou internal selon son degré d'accessibilité en dehors et dans la classe.

La majeure différence en C# est de pouvoir créer des attributs qui permettent eux aussi de qualifier l'entité, les attributs permettent de donner des informations complémentaires sur la classe ou les méthodes.

Pour créer un attribut, il faut créer une classe héritant de la classe Attribute.

Une fois la classe créée, il suffit de placer l'attribut entre crochet juste avant la déclaration de classe par exemple ou encore de méthode ou de variable. Il est possible de cantonner l'utilisation de l'attribut uniquement aux classes par exemple.

Certains attributs provoquent des messages qui s'écrivent directement dans le compilateur c'est le cas des attributs de type Obsolete, entre autres, qui sont préexistants dans le frameworks.NET. D'autres à l'image de certains attributs tels que les descriptions de classe sont utiles en utilisant la réflexion.

La réflexion permet en fait d'obtenir des informations sur les types en fournissant des objets. La réflexion peut se faire grâce à plusieurs méthodes, telle que getType. Concernant les Attributs, l'une des méthodes permettant l'accès à ses informations est getAttribute.

Le programme Attributs illustre une utilisation de la description des attributs. Il s'agit d'un exemple simple où les attributs permettent la description d'une classe

```
public class DescriptionAttribute : Attribute
    {
        public string Description { get; set; }

        public DescriptionAttribute()
        {
        }

        public DescriptionAttribute(string message)
        {
            Description = message;
        }
    }
```

```
[Description(Description = "Cette classe correspond à une personne")]
    public class Personne
    {
        string Nom;
        string Prenom;
        int Age;

        [Description(Description = "Ce constructeur permet d'assigner l'identité de la
personne")]
        public Personne(string nom, string prenom, int age)
        {
            Nom = nom;
            Prenom = prenom;
            Age = age;
        }

        [Description(Description = "Cette méthode permet d'écrire dans la console ce
que l'utilisateur dit")]
        public void Parler(string parole)
        {
            Console.WriteLine(parole);
        }

    }
```

```
static void Main(string[] args)
        {
            Personne Jean = new Personne("Mondu","Jean",30);   //On crée un objet de
type Personne

            //On utilise la réflexion pour récupérer les attributs
            Type type = typeof(Personne);
```

```
            Attribute[] lesAttributs = Attribute.GetCustomAttributes(type,
typeof(DescriptionAttribute));
            foreach (DescriptionAttribute attribut in lesAttributs)
            {
                Console.WriteLine("\t" + attribut.Description);
            }
            Console.ReadKey();
        }
```

La sortie console est la suivante :

```
Cette classe correspond à une personne
```

L'utilisation de la réflexion permet ici l'affichage des métadonnées de la classe Personne.

3) *Les propriétés et indexeurs*

Les propriétés sont des valeurs pouvant être lue mais également modifiée grâce à **get** (retourne la valeur) et **set** (modifie la valeur). Il s'agit donc de permettre à l'utilisateur d'accéder à ces valeurs sans pour autant qu'il puisse visualiser le code permettant d'accéder et modifier ces variables. Une propriété en lecture seule ne possède pas de set, une propriété en écriture seule ne possède pas de get. Le mot clé **value** représente la valeur de modification dans le set.

```
Private int nombre ;
Public int Nombre{
get{return nombre ;}
set{nombre=value ;}
}
```

Cet exemple nous permet simplement de récupérer et modifier les valeurs d'un champ. Les propriétés peuvent également retourner une valeur différente du champ, par exemple si l'on a les secondes, le get peut retourner les heures. De même la valeur modifiée peut dépendre d'un traitement à l'intérieur du set.

Les indexeurs sont des propriétés mais dont les accesseurs prennent un argument et qui permettent d'utiliser une instance d'une classe sous forme de tableau. Le mot clé **this** permet de définir l'indexeur.

La syntaxe d'un indexeur est la suivante :

```
Public int this[int index]{
Get(){}
Set(){}
}
```

```
Public class jourAnniv(){
Private int[] jourAnniversaire={3,10,25,27} ;
Public int this[int index]{
get(){return jourAnniversaire[index];}
set(){jourAnniversaire[index]=value;}
}
```

Le fonctionnement d'un indexeur est semblable à celui d'une propriété mais il permet une indexation sous forme de tableau virtuel qui peut s'avérer pratique.

IV/ Lexique :

FrameWorks	Ensemble cohérent de différents éléments de logiciel structurels pour créer les fondations d'un logiciel.	Introduction
Application	Programme utilisé pour réaliser une tâche.	Introduction
Composant	Elément constitutif d'un logiciel.	Introduction
Services WEB	Protocole permettant le communication et l'échange de données entre une application et un serveur.	Introduction
Bibliothèque	Collection de portion de codes effectuant un traitement spécifique indépendants du reste du programme et réutilisables, qui est prête à être utilisée par des programmes.	Introduction
Environnement d'exécution	Aussi appelé Runtime. Logiciel responsable de l'exécution des programmes informatiques. Il ne permet que l'exécution et non la programmation.	Introduction
IDE	Aussi appelé Environnement d'exécution. Ensemble d'outils permettant d'améliorer la productivité des programmeurs grâce notamment à un éditeur de texte pour la programmation, un débogueur, un compilateur…	Introduction
Débogueur	Logiciel permettant au programmeur d'identifier les défauts de conception d'un programme qui peuvent être à l'origine de dysfonctionnement.	Introduction
Common Intermediate Language	Langage de programmation de plus bas niveau lisible par l'humain. C'est un bytecode, soit un code intermédiaire entre les codes source et les instructions machines qui n'est pas exécutable. C'est celui dans lequel sont compilés les codes sources de haut niveau sur la plateforme .NET.	Introduction
Langage MSIL	MicroSoft Intermediate Language. Nom préalablement donné au CIL avant la standardisation du C#.	Introduction
Type-Safe	Permet d'éviter les erreurs de types. Ce qui permet d'éviter des différences de types au sein d'une même expression.	Introduction
Gestion des exception	Permet de gérer les conditions exceptionnelles ou erreurs pendant l'exécution du programme.	Introduction
Garbage Collection	Logiciel de gestion automatique de la mémoire. Permet de récupérée la mémoire qui n'est plus utilisée.	Introduction
Console	Périphérique de télécommunication des entrées-sorties d'un programme. Elle permet donc une interaction directe avec l'utilisateur.	I-

Variable	Permettent d'attribuer un nom unique à une valeur.	I-
Itération	Action de répéter un processus.	I-
Incrémenter		I-
Type	Permet de définir la nature des données.	I-
Booléen	Variable à 2 états, en général un état true et un état false qui correspondent aux 2 états logiques.	I-
Type-casting	Conversion d'une valeur de type en une autre, plus adaptée.	I-
Compilateur	Programme qui transforme un code source en un code objet qui est une série d'instructions en langage informatique souvent en langage binaire.	I-
Type object	Type spécifique s'appliquant aux objets. (voir plus bas). Ces objets auront des méthodes propres contrairement aux types primaires.	I-
Héritage	Permet de créer une classe qui possède les mêmes attributs qu'une autre classe et qui est susceptible d'avoir des caractéristiques propres. La classe héritée est appelée classe fille et elle hérite de la classe mère. Elle possède donc forcément les mêmes attributs et les mêmes méthodes que la classe mère.	I-
Attribut	Entité définissant les propriétés d'un objet ou d'une classe.	I-
Instancier	Créer un objet à partir d'un objet.	I-
Objet	Conteneur qui contient des informations et mécanismes manipulés dans un programme.	I-
Implémenter	Fournir une fonctionnalité	I-
Pointer	Attribuer à une donnée (un pointeur) une adresse mémoire.	I-
Argument	Données traitées dans une fonction.	I-
String	Chaine de caractères.	I-
Retourner	Ce que le programme renvoie à la fin de l'exécution.	I-
Concaténation	Action de mettre bout à bout 2 chaines de caractères.	I-
Signature	Définit les types de données qui sont acceptables pour une méthode.	I-
Méthode	Portion de code effectuant un traitement spécifique placé dans une classe.	I-
Classe mère	Classe de base de laquelle une classe a hérité toutes ses propriétés.	I-
Override	Possibilité de redéfinir un comportement spécifique à une classe fille. C'est-à-dire modifier ou compléter la méthode mère pour l'utiliser dans la classe fille. Sans override, la méthode dans la classe fille aura le même comportement que celle de la classe mère.	I-
Constructeur	Fonction appelée lors de l'instanciation qui permet d'voir la mémoire nécessaire pour l'objet créé et de l'initialiser.	I-
Event Handler	Programme qui s'active lorsqu'un évènement particulier survient au cours de l'exécution d'un autre	II-

	programme.	

V/ Référence :

Cette est non exhaustive, certaines références moins importantes n'ont pas été citées.

Introduction :

-https://openclassrooms.com/fr/courses/1526901-apprenez-a-developper-en-c

-https://docs.microsoft.com/fr-fr/dotnet/csharp/

II- Les Bases du C# - Les types en C#

- https://tahe.developpez.com/dotnet/csharp/?page=page_4#LIV

- https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/value-types

II- Les Bases du C# - Programmation Orientée Objet

-https://tahe.developpez.com/dotnet/csharp/?page=page_4#LIV

-https://openclassrooms.com/fr/courses/2818931-programmez-en-oriente-objet-avec-c

-https://fr.wikibooks.org/wiki/Programmation_C_sharp

II- Les Bases du C# - Gestion des exceptions

-https://www.tutorialspoint.com/csharp/csharp_exception_handling.htm

-https://tahe.developpez.com/dotnet/csharp/?page=page_4#LIV

III- Concepts Clé du C# - Gestion des évènements

- https://fr.wikibooks.org/wiki/Programmation_C_sharp/Delegates_et_events

- https://openclassrooms.com/fr/courses/2818931-programmez-en-oriente-objet-avec-c/2819111-delegues-evenements-et-expressions-lambdas

- https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/events/index

III- Concepts Clé du C# - Attributs

-https://ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf

- https://openclassrooms.com/fr/courses/2818931-programmez-en-oriente-objet-avec-c/2819186-attributs-et-reflexion

- https://fr.wikibooks.org/wiki/Programmation_C_sharp/Attributs

III- Concepts Clé du C# - Propriétés et indexeurs

-https://fr.wikibooks.org/wiki/Programmation_C_sharp/Propriétés_et_indexeurs

-https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/indexers/index

Lexique :

-Wikipédia