THEME 2
LES OBJETS

CONCEPT GENERAL

La complexité du monde réel nous conduit à définir, classifier, organiser et hiérarchiser les différents objets qui le composent.
Ces objets possèdent des propriétés et des comportements propres. Une même famille d'objets, avec des caractéristiques communes donc, peut être regroupée dans un ensemble appelé classe.
La programmation orientée objet reprend à son compte ces notions d'objet et de classe. Le développeur pourra à sa guise créer une classe, définissant les caractéristiques et les propriétés d'une même famille d'objet. Un objet, c'est-à-dire à un élément particulier d'une classe, est appelé instance d'une classe.

Exemples :

Chien
attributs :
âge, poids, couleur
comportements (Méthodes):
courir, aboyer, manger

Etudiant
attributs :
Nom, âge
comportements (Méthodes)
bavarder, écouter, dormir

L'objectif est de faciliter et de fiabiliser l'écriture de programmes complexes en décomposant l'analyse à l'aide de différentes entités distinctes qui coopérent.

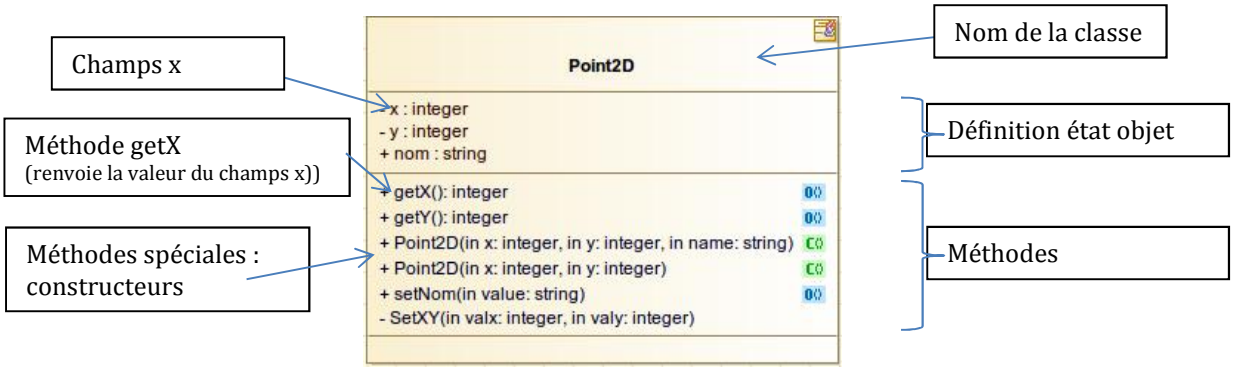
Du point de vue interne, l'objet est décrit en détail tant au niveau des propriétés que de ses méthodes d'action.

Du point de vue externe, l'objet est une boîte noire accessible au travers d'une certaine interface (les méthodes).

Avantages :

- réutilisation des objets
- structuration modulaire
- autonomie des objets
- maintenance et debuggage facilités

En adoptant le formalisme du langage de modélisation UML une représentation graphique des classes ainsi que leurs interactions est possibles :



TERMINOLOGIE

Vous devez absolument connaître la terminologie suivante :

Classe : défini un type d'objet
La classe définira en son sein l'état de l'objet (la zone des variables pour faire simple) et les méthodes (les fonctions) manipulant l'état.

Objet : instance d'une classe.
Concrétisation physique q'un type particulier d'objet. Exemple : La classe Voiture défini le concept général (définition) de ce qu'est une voiture. Ma porche Carrera est une concrétisation de la classe Voiture (Elle a une couleur qui lui est propre, un millésime, une cylindrée etc..).

Attribut ou Champ (d'une classe): une caractéristique de cette classe
Cette caractéristique sera codée en pratique par une variable d'un certain type.

Etat (d'un objet) : **valeur prise par l'ensemble des champs de d'un objet**

Méthode (d'instance) : assimilé à une fonction appartenant à un type d'objet particulier.
A la différence d'une fonction la méthode d'instance s'appelle au travers de l'objet. La méthode pourra modifié au besoin l'état de cet objet.
Ex : maPorcheCarrera.ContactMoteur() ;
On applique la méthode ContactMoteur () sur l'objet particulier maPorcheCarrera qui possède son propre état

Propriété : au sens général de l'informatique une propriété est assimilée à un attribut de la classe
En C#, le terme propriété est différent : il s'agit d'une méthode particulière dont l'unique but est d'accéder en lecture ou écriture à un champ précis

Accesseurs : Méthodes qui fournissent un point d'accès à un champs de l'objet depuis l'extérieur.

Encapsulation : Regroupement au sein d'une même entité d'un l'état et d'un comportement. Un mécanisme de protection peut-être (et doit être !!!) mis en place pour interdir tout accès non autorisé à l'état depuis l'extérieur. Certaines méthodes d'instances seront les seules autorisés à des accès.

UTILISATION DES CLASSES ET MANIPULATIONS D'OBJETS

DEFINITION D'UNE CLASSE

Une classe permet de regrouper au sein d'une même structure le format des données qui définissent l'objet et les fonctions destinées à manipuler ces objets. On dit qu'une classe permet d'encapsuler les différents éléments et les fonctions dans un ensemble appelé objet.

```
//Version 1 :1 seul fichier
using System;

class Chat
{
    uint _age;// par défaut Private!!
    private uint _poids;

    public String Crier()
    { return("Miaouuu"); }
    public void SetAge(uint var)
    { _age = var; }
}
```

Déclaration de la classe

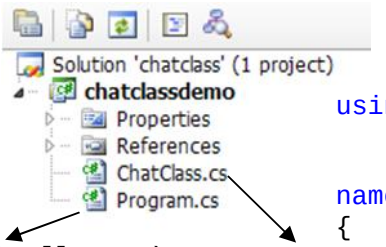
```
class Program
{
    static void Main(string[] args)
    {
        Chat unMatou;
        unMatou = new Chat();
        Console.Write(unMatou.Crier());
        unMatou.SetAge(10);
        Console.ReadKey();
    }
}
```

Utilisation de la classe

Commentaires

- ✓ _age et _poids correspondent aux champs de la classe (ie données, état interne) (rq : une convention de nommage possible est de préfixer les champs par un underscore _xxxxx)
- ✓ Crier() correspond à une méthode de la classe (remarquez la majuscule par convention qui est appliquée : attention, en java, la convention est « tout en minuscule sauf le nom de la classe »)
- ✓ `SetAge` () correspond aussi à une méthode de la classe mais dont l'unique rôle et de fournir une interface d'accès au champs privé `age`. `SetAge` est appelé accesseur.
- ✓ Le mot-clé **public** indique que les méthodes ou champs sont accessibles de l'extérieur.
- ✓ Le mot-clé **private** indique que les données ne sont accessibles que par les méthodes appartenant à la classe (non accessible de l'extérieur directement).
- ✓ Chat UnMatou; crée une réference sur Chat
- ✓ UnMatou = new Chat(); crée une instance de Chat (ou objet de type de Chat)
- ✓ UnMatou._age=10 ; ne compile pas car _age est private donc inacessible de l'extérieur !

// version 2 : 2 fichiers




```
using System;
using ProjetChat;

    class Program
    {
     static void Main(string[] args)
        {
            Chat UnMatou;
            UnMatou = new Chat();
Console.Write(UnMatou.Crier ());
            UnMatou. SetAge (10);
            Console.ReadKey();
        }

    }
```

```
using System;


namespace ProjetChat
{
    public class Chat
    {
        uint _age;// par défaut Private!!
        private uint poids;

        public String Crier()
        { return("Miaouuu"); }
        public void SetAge(uint var)
        { _age = var; }
    }
}
```

Commentaires :

- ✓ La classe est livrée dans un fichier séparé pour une meilleure lisibilité
- ✓ L'espace de nommage étant différent, la clause `using ProjetChat` est utilisée pour la visibilité de la classe Chat

De manière générale, et pour des raisons de sécurité, les champs sont maintenus private.
Seules les méthodes publiques seront accessibles depuis l'extérieur.

De cette manière, en cas de problème sur un objet, seules les méthodes appartenant à la classe incriminée peuvent être mises en cause puisque ce sont les seules qui puissent accéder aux données.

CREATION ET VIE DES OBJETS

L'opérateur new permet de créer dynamiquement (ie : en cours d'éxécution, au milieu de votre code) des objets qui seront stockés en RAM.
L'opérateur new renvoie une référence vers l'objet ainsi crée.
La manipulation des objets s'effectue de manière indirecte (on parle d'indirection) par le biais des références.

```
Chat Matou1,Matou2; // 2 référence nulles ( non initialisées)
Matou1= new Chat(); //1er objet Chat : référence Matou1 initialisée avec new
Matou2= new Chat(); //2ème objet Chat
….
Matou1=new Chat(); // Matou1 fait référence a un 3ème chat!
// A ce stade le 1er objet Chat n'est plus référencé donc éligible à une destruction (rôle du ramasse-miette)

//Chaque objet en mémoire possède ses propres propriétés _age et _poids;
```

Au moment de la création de l'objet, une méthode spéciale, le constructeur, est appelée. Ce mécanisme répond à la question : *comment créer et initialiser l'objet*.
Un constructeur par défaut existe : exemple d'appel new Chat();

La fin de vie de l'objet est décidée par le ramasse-miette qui désalloue l'espace mémoire. Il en résulte que ,non contrôlé par l'utilisateur, ce mécanisme est non déterministe (ne convient pas pour des systèmes temps réels au sens strict).

PRINCIPE

Les méthodes définies dans les classes ne sont ni plus ni moins que des fonctions qui permettent :

- ✓ de définir les comportements des objets
- ✓ d'accéder (interface publique) et de manipuler les champs attributs des objets

Nous distinguerons dans le cours 4 types de méthodes :

- les méthodes d'instances
- les accesseurs ou **propriétés**
- les constructeurs (et destructeurs)
- les méthodes de classes (méthodes statiques)

Les méthodes d'instances sont appelées au travers d'un objet : `unMatou.Crier();`

Lors de l'appel d'une méthode une référence surl'objet courant (dans l'exemple unMatou) est transmise implicitement par le biais d'un mot clé :**this**.

Exemple : appel de la méthode: `unMatou.SetPoids(5);`

```
 class Chat
{
 uint poids;
.....
public void setPoids (uint poids)
        { poids = poids; }
}
Commentaire :
Il y a ambiguité entre poids variable locale
transmise lors de l'appel et le champs poids
```

Solution: emploi de **this**

```
class Chat
{
uint poids
.....
public void setPoids (uint Poids)
        { this.poids = poids; }
}
Commentaire :
this :objet courant ( unMatou)
this.poids : accès à la variable Poids de l'objet
courant (unMatou)
```

Rq : la convention de nommage _poids aurait évité cette ambiguité. L'underscore permet du premier coup d'œil de réperer les champs des classes. Cependant avec l'évolution des environnements de developpements certains considèrent que ce nommage est inutile.

PASSAGE PAR VALEUR

Si un paramètre est déclaré pour une méthode sans ref ni out **le passage se fait par valeur** : une variable ou une référence locale est crée puis la valeur recopiée.

```
static void Method( int varlocal,MonInt test2)
        {
        test2.vartest = 10;
        varlocal = 10;
        }

       public class MonInt
        {
            public int vartest=5;
        }
        static void Main(string[] args)
        {

            MonInt testint = new MonInt();
            int var = 5;
            Method(var,testint);
            Console.WriteLine(testint.vartest);
            Console.WriteLine(var);
            Console.ReadKey();
        }
```

- ✓ MonInt est une classe donc testint est une variable de type référence.
- ✓ test2 est une copie locale de la référence (**mais pas de l'objet**)
- ✓ test2.vartest = 10; modifie la case mémoire
- ✓ var est crée dans la pile (objet valeur)
- ✓ varlocal est une copie locale=> pas de modification de var
- ✓ Le cas string est à part puisque bien que str soit une référence il y création d'un nouvel espace mémoire et recopie du contenu pointé par str. La classe string est dite immuable (en anglais immutable)=> une fois initialisé par son constructeur toute modification entraîne un nouvel objet de type string

```
10
5
```

Pour bien comprendre le passage de paramètre il est recommandé de tester par vous-même dans Visual Studio !

REMARQUE :Un tableau est toujours désigné par sa référence et est alloué par new (même si ce new n'apparaît pas quand un tableau est créé et initialisé dans la foulée). C'est à ce moment que la véritable zone de données du tableau est allouée. La référence « pointe » alors sur cette zone.

EN RESUME: UN PASSAGE PAR VALEUR CREE UNE COPIE TEMPORAIRE PAR CLONAGE:

- **dans le cas d'un type 'valeur' : copie d'un objet temporaire ayant même valeur;**
- **dans le cas d'un type référence: clonage des références => il n'y pas création d'un nouvel objet!**

PASSAGE PAR REFERENCE

Mot clé ref

Le mot clé **ref** fait en sorte que les arguments soient passés par référence. La conséquence est que toute modification apportée au paramètre dans la méthode est reflétée dans cette variable lorsque la méthode appelante récupère le contrôle. Pour utiliser un paramètre **ref**, l'appelant et l'appelé doivent tous deux utiliser le mot clé **ref** explicitement. Par exemple :

```
class RefExample1
{
   static void Method(ref int i)
   {
      i = 44;
   }
   static void Main()
   {
      int val = 0;
      Method(ref val);
      // val is now 44
   }
}
```

```
class RefRefExample2
{
   static void Method(ref string s)
   {
      s = "changed";
   }
   static void Main()
   {
      string str = "original";
      Method(ref str);
      // str is now "changed"
   }}
```

MOT CLÉ OUT

Le mot clé **out** fait en sorte que les arguments soient passés par référence. Il est semblable au mot clé ref, mais ref nécessite que la variable soit initialisée avant d'être passée. Pour utiliser un paramètre **out**, la définition de méthode et la méthode d'appel doivent toutes deux utiliser le mot clé **out** explicitement. Par exemple :

```
Class OutExample1
{
static void Method(out int i)
   {
      i = 44;
   }
   static void Main()
   {
      int value; // NON initialisée
      Method(out value);
      // value is now 44
   }
}
```

```
class OutReturnExample2
{
   static void Method(out int i, out string s1, out string s2)
   {
      i = 44;
      s1 = "I've been returned";
      s2 = null;
   }
   static void Main()
   {
      int value; // NON initialisée
      string str1, str2;
      Method(out value, out str1, out str2);
      // value is now 44
      // str1 is now "I've been returned"
      // str2 is (still) null]}
```

ACCES AUX CHAMPS DES CLASSES

Par les raisons déjà évoquées précédemment les attributs des classes seront déclarée *private.* Un accès direct dans votre programme est interdit : ~~UnMatou.Poids=10.~~
Des interfaces d'accès, nommées acesseurs (getter/setter en aanglais) sont donc nécessaires pour des lectures/écritures simples.
Deux façons de procéder sont possibles :

1- acceseurs standarts

this : référence sur l'objet en cours

Lors de l'appel unMatou.Crier(); this fait référénce à unMatou.
This.GetAge() renvoie l'age du chat sur lequel on travaile à l'instant t.

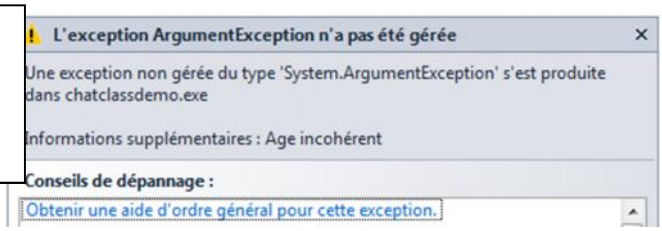
```
public class Chat
{
    uint _age; // par défaut Private!!
    private const uint AgeMaximum = 100;

    public String Crier()
    {
     return ("Miaouuu j'ai "+this.GetAge()+" ans");
    }

    // Accesseur
    public void SetAge(uint age)
    {
        if (age < AgeMaximum)
        {
            _age = age;
        }
        else
        {
            throw new ArgumentException("Age incohérent");
        }
    }
    public uint GetAge()
    { return (_age); }
}

class Program
{
    static void Main(string[] args)
    {
        Chat unMatou = new Chat();
        unMatou.SetAge(10);
        unMatou.Crier();
        Console.WriteLine("le matou a " + unMatou.GetAge() + " ans");
        Console.ReadKey();
    }
}
```

Exemple d'erreur si je mets 130 !
On remarquera que l'exception n'est pas rattrapée par un try/catch
Un exemple est donné ans les pages d'après



2- Propriétés

Les propriétés sont des membres de la classe offrant un mécanisme souple pour la lecture, l'écriture ou le calcul des valeurs de champs privés. Elles peuvent être utilisées comme si elles étaient des membres de données publiques, mais en fait, ce sont des méthodes spéciales appelées accesseurs. Elles facilitent l'accès aux données tout en préservant la sécurité et la souplesse des méthodes

- Avec des propriétés, une classe peut exposer de manière publique l'obtention et la définition de valeurs, tout en masquant le code d'implémentation ou de vérification.
- Un accesseur de propriété get permet de retourner la valeur de propriété, et un accesseur set est utilisé pour assigner une nouvelle valeur. Ces accesseurs peuvent avoir des niveaux d'accès différents. Pour plus d'informations, consultez Accessibilité de l'accesseur asymétrique (Guide de programmation C#).
- Le mot clé value sert à définir la valeur assignée par l'indexeur set.
- Les propriétés qui n'implémentent pas de méthode set sont en lecture seule.

```
using System;

public class Chat
{
    uint _age; // par défaut Private!!
    private const uint AgeMaximum = 100;

    public String Crier()
    {
        return ("Miaouuu j'ai " + AgeChat + " ans"); // ou this.AgeChat
    }

    // Propriétés :get/set
    public uint AgeChat
    {
        get { return this._age; }
        set
        {
            if (value > AgeMaximum)
            {
                throw new ArgumentException("Age du Chat (" + value + ") invalide");
            }
            else
            {
                this._age = value;
            }
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Chat UnMatou = new Chat();
        UnMatou.AgeChat = 10; // exception non rattrapée
        Console.WriteLine("le matou a " + UnMatou.AgeChat + " ans");
        //try-catch
        try
        {
            UnMatou.AgeChat = 135; // exception rattrapée
        }
        catch (Exception ex)
        {
            Console.Error.WriteLine(ex.Message);
        }
        Console.ReadKey();
    }
}
```

Remarque: La plupart des cas les "get et set" se bornent à des lectures/écritures simples. Le C# 3 introduit les propriétés automatiques (à déconseiller pour débuter).

Ex: Class MaClasseDemo

```
{ public int Age {get;set;}  //  création automatique d'un champ privé correspondant à l'age
 public string Nom {get;set} // création automatique d'un champ privé correspondant au nom
}
```

```
MaClasseDemo demo=new MaClasseDemo() ;
demo.Age=10; // la methode spéciale (c'est une propriété) Age est appellée.
```

DECLARATION DE CONSTRUCTEUR

Dans tout programme le développeur doit se préoccuper de l'initialisation des objets créés. Il peut comme dans les cas précédents incorporés des méthodes spécifiques. Cependant le langage C++ inclut une fonction membre spéciale appelée constructeur. Cette méthode, du même nom que la classe, est appelée automatiquement au moment de la création d'un objet.
De la même manière, au moment de la destruction de cet objet, une fonction spéciale du nom de la classe mais précédé d'un tilde ~, est invoqué.

```
public class Chat
    {
        private uint _age;
        private uint _poids;

        public Chat(uint age,uint poids)
        {
            _age = age;
            _poids = poids;
        }

        public override string ToString()
        {
            String affichage =
String.Format("j'ai {0} ans et je pèse {1} kg",
_age, _poids);
            return affichage;
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Chat chat1;
            chat1=new Chat(2,15);
            Console.WriteLine(chat1);
            Console.ReadKey();
        }

    }
```

SURCHARGE DE CONSTRUCTEUR

La surcharge de constructeur est utile lors l'utilisateur souhaite crée et initialiser des objets de manières différentes.
Le constructeur est récrit avec un prototype contenant les arguments utilisés pour initialiser l'objet. A l'éxécution le constructeur ayant la même signature que celui associé à l'opérateur new est invoqué.

Si un constructeur avec argument a été défini alors la déclaration d'objet non initialisé est impossible (ex Chat Frisky ; génrtère une erreur. Alors que : Chat Frisky(5); est autorisé)

Attention : veuillez à ce que l'état de l'objet soit toujours pertinent. La crétion d'un chat d'age 0 est'il pertinent ?

Ci-après un exemple de surchage du constructeur Chat.

```
public class Chat
    {
        public uint Age { get; set; }
        public String Nom { get; set; }


        // default ctor
        public Chat()
        {
            Age = 0; // Est ce bien pertinent?
            Nom = "NoName";
        }
        // default ctor 2
        public Chat(uint age)
        {
            Age = age;
            Nom = "NoName";
        }
        // default ctor 3
        public Chat(uint age, String nom)
        {
            Age = age;
            Nom = nom;
        }

        public override string ToString()
        {
            String affichage = String.Format("j'ai
{0} ans et je m'appelle {1}", Age, Nom);
            return affichage;
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Chat[] tableauChat;// référence vers un tableau de
Chat

            tableauChat=new Chat[3]; // crée le tableau et
initiale la référence
            tableauChat[0] = new Chat();// ctor par défaut
(BOF !)
            tableauChat[1] = new Chat(10); //ctor2
            tableauChat[2] = new Chat(1,"Frisk");// ctor3
            foreach (Chat unChat in tableauChat)
                Console.WriteLine(unChat);
            Console.ReadKey();
        }
    }
```

METHODES ET CHAMPS STATIQUES

Les méthodes et champs statiques ne s'appellent pas au travers d'un objet comme les exemples précédents mais à partir de leur type (classe de l'objet)

Méthode d'instance

MaClassObjet objet1=new MaClassObjet();
objet1.MaMethode;

Méthode statique

MaClassObjet.UneAutreMethode( )

Ci après des exemples de méthodes statiques déjà manipulées:

- Console.WriteLine() où WriteLine est la méthode statique de la classe Console.
- String.Parse()où Parse est la méthode statique de la classe string

L'attribut statique devant un champ d'une classe indique que la donnée (la variable) conserve sa valeur sur les différentes instances de la classe. Nous pouvons dire en quelque sorte que cette donnée sera partagée entre les différents objets issus de la même classe. Cette facilité permet notamment de compter le nombre d'instance d'une classe (ie nombre d'objets) et par exemple de contrôler une population donnée ou le nombre d'accès à une ressource.

```
public class Imprimante
    {   // champs
        private static uint nombreImprimantes;
        private string imprimanteNom;


        // accesseurs(get/set)
        public string ImprimanteNom // notez la majuscules
        {                           // qui différencie le champ
            // de la propriété
            get { return (imprimanteNom); }
            set { imprimanteNom = value; }
        }

        // PROP AUTO
        public int NumeroSalle { get; set; }


        //Constructeur
        public Imprimante(string name, int num)
        {
            nombreImprimantes++;  // incremente nbre total imprimante
            imprimanteNom = name; // mise a jour champ nom
            NumeroSalle = num;    // mise a jour numero salle
        }

        // methode statique
        public static uint NombreTotalImprimanteGEII()
        {
            return (nombreImprimantes);
        }

        // surchage de la methode ToString pour ce decrire
        public override string ToString()
        {
            return (String.Format("imprimante {0} localisée en salle {1}", imprimanteNom, NumeroSalle));
        }
    }
```

Résultat écran
imprimante brother localisée en salle 215
imprimante HP localisée en salle 208
salle pour hp2: 214
le nombre total est:3

```
static void Main(string[] args)
        {
            Imprimante brother = new Imprimante("brother", 215);
            Imprimante hp = new Imprimante("HP", 208);
            Imprimante hp2 = new Imprimante("HP", 214);
            Console.WriteLine(brother);
            Console.WriteLine("salle pour hp2: " + hp2.NumeroSalle);
            Console.WriteLine("Total imprimates:" + Imprimante.NombreTotalImprimanteGEII());
            Console.ReadKey();
        }
```

TRAVAIL PERSONNEL

1- Passage d'argument

Avant de tester ce programme pronostiquer l'affichage?(en justifiant!!)Le saisir et vérifier ***en pas à pas.***

```
namespace ConsoleApplication1
{
    class MonInt
    {
        public int Val { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MonInt refUnInt=new MonInt();
            int x = 3;
            refUnInt.Val=10;
            fct(x,refUnInt);
            Console.WriteLine( "x: {0} val: {1}",x,refUnInt.Val);
            Console.ReadLine();
        }


    static void fct(int y,MonInt test)
        {
        test.val=20;
        y = 30;        }
    }

}
```

2- Passage par référence d'un type valeur

Remplacer les xxxx de telle manière à réaliser un passage par référence sur la variable a initialisée. Pronostiquer la valuer prise par a après l'appel de f
Refaire maintenant sur a non initialisée.Apporter les modifications pour que cela fonctionne.

```
static void Main(string[] args)
        {
            int a = 3;
            f(xxxxxx); // passage par référence
            Console.WriteLine(a); // affiche
            Console.ReadKey();
        }
        static void f(xxxxx)
        {
            aa = 10; // modifie a du Main
        }
```

3- Passage par valeur d'un type référence

Saisir le programme suivant. Pronostiquer le résultat de l'affichage après l'appel de f. Expliquez(indice : la référence ref1 a-t-elle était modifié par f.)

```
class MonInt
    {
        public int Valeur { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MonInt ref1 = new MonInt();
            ref1.Valeur = 20;
            f(ref1);
            Console.WriteLine(ref1.Valeur);
            Console.ReadKey();

        }
        static void f(MonInt r)
        {
            r = new MonInt();
            r.Valeur = 30;
        }
    }
```

4- Passage par référence d'un type référence

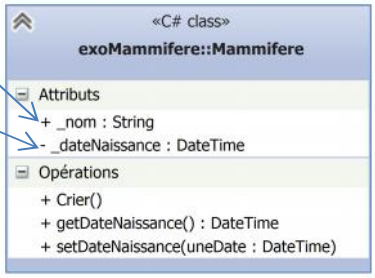
Modifier le programme précédent afin de passer par référence ref1.
Que se passe t'il . Expliquez (indice : la référence ref1 a-t-elle était modifié par f.)

5- Modéliser une classe simple

- Dans 1 fichier (clic droit ajouter>nouvel élément>classe) écrire la définition d'un Mammifère.
- Faire un programme qui crée un mammifère, modifie sa date de naissance et son nom. Faire crier le mammifère crée et afficher sa date de naissance
- Render privé le champ _nom. Que se passe-t-il ?
- Pour accéder à _nom écrire une propriété Nom (on commence par une majuscule puisque une propriété est assimilé à une méthode) en employant get et set. Vérifier grâce à votre programme que tout se passe bien
- Rajouter une propriété automatique Zoo de type String. Testez.

Le + indique un attribut public

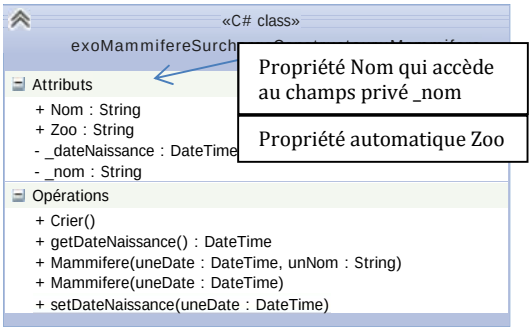
Le - indique un attribut privé



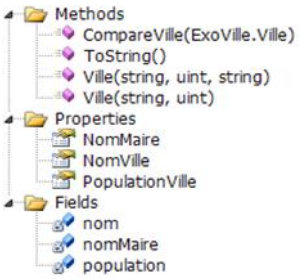
6- Surcharge de constructeur

Créer un nouveau projet.

- Rajouter à la classe précédente 2 constructeurs
- créer 2 mammifères en utilisant des constructeurs différents
- Essayer de créer un mammifère en utilisant le constructeur par défaut ? Que se passe t'il.



7- Créer une classe ville



ExoVille est l'espace de nommage (namespace)

CompareVille compare 2 villes (nom et population identique)

- ✓ Un argument de type Ville en entrée
- ✓ Un résultat en sortie de type bool (booléen)

ToString est la réécriture de la méthode ToString (voir exemple dans ce chapitre). Cette fonction renvoie un string .

```
public override string ToString()
    {
        string description = string.Format("Ville {0} , population {1}, maire {2}", nom, population, nomMaire);
        return (description);
    }
```

Pour afficher une ville : `Console.WriteLine(toulon);`
`En sous-main la méthode ToString() de l'objet toulon qui est du type Ville est appelée !`
Le programme principal sera simple et mettra en œuvre la classe Ville

- ✓ Création des objets "à la main" (pas de console.read) à l'aide des différents constructeurs
- ✓ Affichage des différents champs (mise en œuvre surcharge de ToString())
- ✓ Tester CompareVille(attention CompareVille est une **méthode d'instance**)
  Vous utiliserez par exemple des if/else pour tester différents cas.
- ✓ Rajouter à la classe une méthode statique. CompareVille ayant pour prototype
  `static public bool CompareVille(Ville nomVille1,Ville nomVille2)`
  Tester
- ✓ Rajouter à la classe une méthode **d'instance** qui ajoute deux populations de deux villes (population de la ville sur laquelle s'applique la .méthode + population de la ville référencée par `nomVille2.`
  `public void AdditionnePopulation(Ville nomVille2, ??? uint resultat).`
  Cette méthode doit pouvoir s'utiliser avec une variable non initialisée. Par quoi faut-il remplacer ???? pour que le résultat soit l'argument de sortie. Tester