CM2

- Rappel Concepts de base de la Programmation Orientée Objet
  - Les classes dérivées et le polymorphisme
- Les classes Object et Type
- Les classes partielles
- Les classes abstraites
- Les interfaces
- Les classes conteneurs
- La généricité (modèles de classes)
- Les délégués, la covariance et la contrevariance
- Le langage C# : de la version 1 à la version 4

Les concepts de base de la POO
Les classes dérivées

- Possibilité de définir de nouveaux objets à partir des objets existants
  - Hiérarchie de généralisation/ spécialisation
- Une classe descendante hérite des propriétés de sa classe mère (attributs et méthodes)
- Elle peut être spécifiée :
  - par **enrichissement** par rapport à sa classe mère (ajout de nouveaux champs ou ajout de nouvelles méthodes)
  - par **substitution** i.e. surcharge de certaines méthodes (et par héritage direct de certaines autres)

Les concepts de base de la POO
Les classes dérivées

- Une classe peut être qualifiée de :
  - public : on peut créer partout un objet de cette classe.
  - Rien du tout : on peut créer un objet de cette classe dans le même assemblage seulement
- Le qualificatif "sealed" peut également être appliqué pour empêcher toute dérivation à partir de cette classe.
  - sealed class Chat : Animal { …}
  - On ne peut plus définir de sous-classes de la classe "Chat".

Les concepts de base de la POO
Les classes dérivées

- La syntaxe de définition d'une classe dérivée :

```
class classe_dérivée :  classe_ancêtre
  {
   //définitions des nouveaux membres
   // définitions des nouvelles méthodes ou méthodes surchargées
   } ;
```

- L'héritage multiple n'est pas autorisé en C# (comme en java, il est possible en C++)

Les concepts de base de la POO
Les classes dérivées

- On souhaite définir une classe dérivée de la classe "Livre" pour décrire des livres anciens "LivreAncien".
  - LivreAncien : comporte deux attributs spécifiques : sa "cotation" et un attribut qui donne des informations sur l'histoire du livre "histoire".
    - Ajout de 2 attributs
  - Ajout des méthodes permettant de travailler sur ces nouveaux attributs et surcharge des méthodes de saisie et d'affichage de la classe "Livre", et de la méthode ToString.

Les concepts de base de la POO
Les classes dérivées

- La classe"LivreAncien" comporte :
  - 2 nouveaux attributs
  - 2 constructeurs
  - Les accesseurs sur les nouveaux attributs
  - La surcharge des méthodes de saisie et d'affichage.
  - La surcharge de la méthode "ToString"

```
class LivreAncien : Livre
{
private float cotation;   // attributs supplémentaires
private string histoire;
…
};
```

Les concepts de base de la POO
Les classes dérivées

```
public override void affiche()  // surcharge de la méthode affiche
    {
       base.affiche(); // appel de la méthode affiche de la classe Livre
       Console.WriteLine("\nCotation : " + cotation);
       Console.WriteLine("\nHistoire : " + histoire);
    }

public override void saisie()  // surcharge de la méthode saisie
    {    string s;
       base.saisie(); // appel de la méthode saisie de la classe Livre
       Console.WriteLine("Cotation ? "); s = Console.ReadLine();
       cotation = Single.Parse(s);
       Console.WriteLine("Histoire ? "); histoire = Console.ReadLine();
    }
```


Les concepts de base de la POO
Les classes dérivées

- Les méthodes "affiche" et "saisie" de la classe "Livre" sont également modifiées
  - pour signifier au compilateur que le choix de la méthode "affiche" (ou "saisie") doit être fait dynamiquement
  - selon le type de l'objet sur lequel on applique la méthode.
- Le mot-clé "virtual" est utilisé.

```
public virtual void saisie()
    { …}
public virtual void affiche()
    { …}
```


Les concepts de base de la POO
Les classes dérivées

- Constructeurs et héritage
- Lorsque le constructeur d'une classe dérivée est appelé, le constructeur de la classe mère doit également être appelé.
- On a successivement :
  - appel du constructeur de la classe mère
  - appel des constructeurs des membres de la classe dérivée
  - exécution des instructions du corps du constructeur de la classe dérivée
- Sans indication particulière, c'est le constructeur par défaut de la classe mère qui est appelé.

Les concepts de base de la POO
Les classes dérivées

- L'appel au constructeur par défaut de LivreAncien
  - Provoque tout d'abord l'appel du constructeur de Livre
  - Puis les autres instructions sont exécutées.
- Le même principe est appliqué pour le constructeur avec paramètres.
  - L'appel du constructeur de la classe mère est fait par : base(isb)

```
public LivreAncien() : base()
    {
    cotation=0;
    histoire="";
    }

public LivreAncien(string isb) : base(isb)
    {
    cotation=0;
    histoire="";
    }
```


Les concepts de base de la POO
Les classes dérivées

- Surcharge de la méthode virtuelle "ToString" (de la classe "Object")
- dans la classe "Livre":

```
public override string ToString()
    {
       string s = ""; s = "\nISBN " + ISBN;
       s = s + "\nDate de parution" + dateParution.ToString("d");
       s = s + "\nTitre : " + titre;
       s = s + "\nPrix: " + prix;
       s = s + "\nAuteur(s) : ";
       for (int i = 0; i < nbA; i++)
          s=s+"\n" + tabA[i];
       return s;
    }
```


Les concepts de base de la POO
Les classes dérivées

- Surcharge de la méthode "ToString" (de la classe "Livre")
- dans la classe "LivreAncien":

```
public override string ToString()
    {
       string s = base.ToString();
       s=s + "\nCotation : "+cotation;
       s = s + "\nHistoire : " + histoire;
       return s;
    }
```

Les concepts de base de la POO
Les classes dérivées

- Redéfinition de méthode.
- C# permet de redéfinir une méthode (même nom et mêmes paramètres) d'une classe, dans sa classe dérivée comme une "nouvelle" méthode.
- Par exemple, on souhaite définir
  - une méthode affiche avec 1 paramètre entier dans la classe LivreAncien,
  - mais qui est différente de celle définie dans Livre
  - et qui affiche la cotation du Livre si le paramètre vaut 1 et son histoire s'il vaut 2.

Les concepts de base de la POO
Les classes dérivées

- Rappel de la méthode "affiche" dans la classe Livre

```
class Livre { …
public void affiche(int i) { …}
}
```

- Et dans la classe "LivreAncien"

```
class LivreAncien { …
public new void affiche(int i) { ..}
}
```


Les concepts de base de la POO
Les classes dérivées

- Exemple d'appels des méthodes :

```
Livre liv1, liv3;
LivreAncien liv2;
liv1= new Livre();
liv1.affiche(1); // affiche de la classe Livre
liv2= new LivreAncien();
liv2.affiche(0); // affiche de la classe LivreAncien
liv3= new LivreAncien();
liv3.affiche(1); // affiche de la classe Livre
                 // avec la redéfinition avec new
```


Les concepts de base de la POO
Les classes dérivées/ Polymorphisme

- Le polymorphisme est un mécanisme puissant,
  - très utile dans la spécification d'une hiérarchie de classes.
- Il permet l'application d'une "même" opération (méthode surchargée) à des objets différents.
- Sa mise en œuvre est réalisée par deux mécanismes en C# :
  - la surcharge
  - l'édition de liens dynamique (ligature dynamique)

Les concepts de base de la POO
Les classes dérivées/ Polymorphisme

- On définit dans la classe "Livre" une méthode "SaisieAffiche" qui effectue
  - la saisie d'un livre
  - et son affichage
- En utilisant les méthodes "Saisie" et "Affiche" déjà définies dans la classe.

```
public void SaisieAffiche()   // dans la classe Livre
    {   saisie();
        affiche();
    }
```


Les concepts de base de la POO
Les classes dérivées/ Polymorphisme

- Exemple d'appels des méthodes

```
Livre liv1, liv3;        LivreAncien liv2;
liv1= new Livre();  liv1.SaisieAffiche();
   // méthode SaisieAffiche de la classe Livre, avec saisie et affiche de Livre
liv2= new LivreAncien();  liv2.SaisieAffiche();
   // méthode SaisieAffiche de la classe Livre,
   // avec saisie et affiche de LivreAncien
liv3= new LivreAncien();
liv3.SaisieAffiche();
   // méthode SaisieAffiche de la classe Livre,
   // avec saisie et affiche de LivreAncien
```

Les concepts de base de la POO
Les classes dérivées/ Polymorphisme

- Les méthodes "Affiche" et "Saisie" sont surchargée dans la classe "LivreAncien", et elles sont définies comme "virtual" dans la classe "Livre".

```
public override void affiche()
    { …
    }
public override void saisie()
    { …
    }
```

- La méthode "AfficheSaisie " de la classe "Livre", elle n'est pas surchargée dans la classe "LivreAncien" car elle doit faire le même traitement :
  - Saisie puis affichage mais d'un LivreAncien (pas d'un Livre).

Les concepts de base de la POO
Les classes dérivées/ Polymorphisme

- Le mot clé "virtual" sur les méthodes saisie et affiche de la classe Livre, indique au compilateur
  - Qu'il doit attendre de connaître dynamiquement le type de l'objet pour effectuer le choix de la méthode à appliquer
    - Ici affiche et saisie de Livre ou de LivreAncien
  - On réalise de l'édition de liens dynamique.

Les classes
La classe "Object"

- En C#, toutes les classes sont dérivées de la classe "Object",
  - Même un type valeur peut être considéré comme dérivé d'Object
- Les méthodes de la classe "Object"
  - Constructeur : Object()
  - Méthodes
    - public virtual bool Equals (Object obj); // comparaison
    - public Type GetType();
      - // renvoie des méta-données sur l'objet à l'aide de la classe Type
    - public virtual string ToString(); // méthode virtuelle à surcharger

Les classes
La classe "Type"

- La classe "Type" permet de connaître des méta-données sur un objet comme :
  - Son type de base, son nom complet avec l'espace de noms
    - TypeBase (Type);
  - Savoir s'il s'agit d'un type tableau, d'un type énuméré, d'un type valeur (int, float, …)
    - FullName (String), IsArray, IsEnum, IsPrimitive (bool)
  - Le nom du type (par exemple Int32 pour un int, etc.)
    - Name (string)
  - L'espace de noms
    - NameSpace (String)

(attributs de la classe Type)

Les classes
La classe "Type"

- La classe Type comporte un certain nombre de méthodes comme :
  - GetFields qui retourne un tableau des attributs accessibles par l'objet
    - Les éléments du tableau sont de type FieldInfo .
    - Ils permettent de connaître par exemple, le nom et le type de l'attribut, de tester ses droits, etc.
- La directive typeof un objet Type pour une classe
  - Type t = typeof(LivreAncien);

Les classes
La classe "Type"

Exemple :

```
using System.Reflection;
Livre l = new Livre();
Type t = l.GetType();
Console.WriteLine("méta-Informations)\n");
Console.WriteLine(t.BaseType + t.FullName + t.Name);
foreach (FieldInfo f in
   t.GetFields(BindingFlags.NonPublic|BindingFlags.NonPublic|BindingFl
   ags.Instance))
Console.WriteLine(f.Name + " de type+ f.FieldType.Name);
```

Méthode de la classe Object

Bilblio.Livre
(si Biblio est le NameSpace)

Livre

Etc.

ISBN String

code Int32

dateParution
DateTime

Les classes
Les classes partielles

- Il est possible de fractionner la définition d'une classe (d'une structure ou d'une interface) sur deux fichiers sources ou plus.
- Chaque fichier source contient une partie de la définition de classe,
  - et toutes les parties sont combinées lorsque l'application est compilée.

Les classes
Les classes partielles

- Il est souhaitable de fractionner une définition de classe :
  - Pour des projets volumineux, travail en parallèle sur une classe
  - Lors de la génération automatique de fichiers sources, du code peut être ajouté sans recréer le fichier source
- Pour fractionner une classe, le mot-clé "partial" est utilisé lors de la description de la classe

Les classes
Les classes partielles

- Cette approche est utilisée lors de la création d'applications avec Windows Forms sous Visual Studio.
- Pour la classe de la fenêtre principale du programme, deux fichiers sont créés :
  - Form1.Designer.cs
    - partial class Form1 { …}
  - Form1.cs
    - public partial class Form1 : Form { …}
- Le fichier "designer" contient les aspects interfaces graphiques et le fichier Form1, le code.

Les classes
Les classes partielles

- Mais ce principe peut s'appliquer pour toute classe :

```
public partial class Bibliotheque
    { …
        public Livre rechercheLivre(string);
    }
public partial class Bibliotheque
    { …
        public void ajoutLivre(Livre);
    }
```


Les classes
Classes abstraites

- Une classe abstraite est une classe dans laquelle on a défini au moins une méthode qui n'est pas implémentée
- L'implémentation de ces méthodes appelées méthodes abstraites est laissée à la charge des classes dérivées.
- On ne peut donc pas instancier des objets d'une classe abstraite, mais uniquement des classes dérivées qui auront implémenté ces méthodes virtuelles.

Les classes
Classes abstraites

- L'objectif d'une classe abstraite est de permettre la définition de profils de méthodes (une ou plusieurs).
  - Qui devront être implémentées dans toutes les classes dérivées.
  - Et donc de définir un « comportement » commun à un ensemble de classes dérivées.

Les classes
Classes abstraites

- On considère par exemple une classe Element qui possède une méthode de dessin (Dessiner) qui n'est pas implémentée.

```
public abstract class Element
{  ...
public abstract void Dessiner() ;
...
}
```

- Chaque sous-classe doit spécifier l'implémentation de la méthode "Dessiner" pour être instanciable.
- La création d'un objet de type Element est incorrecte
  - Element elem;
  - elem = new Element() ; // incorrect

Les classes
Classes abstraites

- Description d'une classe dérivée « Rectangle » qui implémente la méthode abstraite « Dessiner »

```
class Rectangle :  Element
{ ..........
  public  override void Dessiner() { ...}
}
```

- On peut alors déclarer des objets de type Ligne
  - Rectangle rect1 = new Rectangle();
  - Element rect2 = new Rectangle();
    - // déclarations correctes

Les classes
Classes abstraites

- Une propriété peut également être définie comme abstraite

```
public abstract class Element
{
  public abstract  double Perimetre { get;}
  ...
}
```

- La classe Rectangle doit implémenter cette propriété.

```
public class Rectangle : Element
{ private double largeur, longueur;
  ...
  public override  double Perimetre
        { get { return 2*largeur + 2* longueur};}
  ...
}
```


Les interfaces

- Une interface contient uniquement des signatures/profils de méthodes et propriétés
  - comparable à une classe "totalement" abstraite
- Une interface peut être implémentée par une classe en donnant la spécification des éléments décrits dans cette interface

Les interfaces

```
public interface ITransformation
{  public void transforme(int i, string s); }

public class Test : ITransformation
{ ...
    public void transforme (int i, string s)
    { // description du code
    }
}
```

La classe Test implémente l'interface ITransformation

Les interfaces

- Il existe des interfaces "prédéfinies" dans les bibliothèques C#.
- Les interfaces sont en général préfixées par "I"
  - *Exemple interface "IComparable"*

```
public interface IComparable
{ int CompareTo(Object); }
```

Les interfaces

- La méthode CompareTo doit renvoyer un entier :
  - <0, si l'objet courant est < l'objet en paramètre
  - >0, si l'objet courant est > l'objet en paramètre
  - = 0 si les objets sont égaux
- Une exception est déclenchée si les objets ne sont pas de même type (ArgumentException)

Les interfaces

- Implémentation de l'interface IComparable par la classe Livre

```
public class Livre : IComparable
{ …
  public int CompareTo (Object o)
    { /* code de la méthode qui indique comment
      comparer et classer 2 livres */
    }
}
```


Les interfaces

- La description des interfaces implémentées par une classe lors de sa définition
  - Permet de connaître les attributs et méthodes qui sont existantes dans une classe
  - Sans étudier le détail du code
- Utile pour l'utilisation de conteneurs prédéfinis ou définis par l'utilisateur.

Les classes conteneurs

- C# propose un ensemble de classes conteneurs d'objets qui permettent de gérer des collections d'objets sous différentes formes
  - Les tableaux dynamiques
  - La classe Stack
  - La classe Queue
  - Les listes triées
  - La classe Hashtable
- Il propose également des conteneurs génériques qui permettent de gérer des collections d'objets de mêmes types
- Il fournit des outils de parcours des collections à l'aide d'itérateurs
- Les interfaces et les classes sont décrites les espaces de noms
  - System.Collections, System.Collections.Generic

Les classes conteneurs

- Les collections prennent comme base les interfaces ICollection, IList et IDictionnary et leurs équivalents génériques.
- Aperçu de la hiérarchie des interfaces (non exhaustif)
  - IEnumerator
    - IDictionnaryEnumerator
    - IEnumerable
      - ICollection
        - IList
        - IDictonnary
      - IEnumerable<T> // interfaces génériques
        - ICollection<T>
          - IList<T>
          - IDictionnary<T>

Les classes conteneurs

- Les classes ArrayList, et List<T> sont des versions avancées d'un tableau (non générique et générique)
  - ArrayList : IList, ICollection, IEnumerable, …
  - List<T> : IList, ICollection, IEnumerable, IList<T>, ICollection<T>, …
- Les classe Stack et Queue représente des collections non génériques de type LIFO et FIFO
- La classe Dictionnary représente une collection de paires clé/valeur
- La classe SortedList représente une collection de paires clé/valeur triées et accessibles par clé ou par index
- La classe Hashtable représente une collection de paires clé/valeur qui sont organisées en fonction du code hashage de la clé.
- Les classes génériques Stack<T>, Queue<T> et Dictionnary<TKey,TValue> sont aussi proposées

La généricité

- Les classes, les structures, les interfaces et les fonctions peuvent être "paramétrées" par des objets de classe qui ne seront définis qu'au moment de l'instanciation.
  - Classe générique = patron = template
- Les collections génériques sont des exemples de classes génériques proposées dans les bibliothèques
- Mais il est possible de définir des classes « utilisateur » génériques.

Généricité : les modèles de classe

- Modèles de classe = classes génériques = patrons
- Utilisation de la notion de type formel pour définir une classe générique.
- Supposons par exemple, qu'on souhaite définir une classe "Ensemble" qui peut contenir n'importe quels types d'objets avec des fonctionnalités classiques
  - Ajout d'un élément
  - Suppression d'un élément
  - Test si un élément est dans l'ensemble
  - Union de 2 ensembles
  - Etc.
- On souhaite ensuite avoir un ensemble d'entiers ou un ensemble de chaînes de caractères, etc.

Généricité : les modèles de classe

- Pour une classe générique il faut :
  - Préciser le nom du paramètre formel utilisé pour la description de la méthode

```
class Ensemble<T> {
    T [] tab ;
    public const int max=100;
    int  nbe;
        ...
    }
```

Nom de type formel

- Ici, l'ensemble est implémenté par un tableau d'éléments de type T.
- La classe Ensemble est générique

Généricité : les modèles de classe

- Définition du constructeur de la classe

```
public Ensemble()
{ nbe=0; tab= new T [max];}
```

- Définition de la méthode « ajout »

```
public void ajout (T elem)
{  if (nbe  < max && ! appartient(elem))
     tab[nbe++]=elem;
}
```


Généricité : les modèles de classe

- Définition de la méthode « appartient »

```
public bool appartient (T elem)
        {bool trouve = false;  int i=0;
        while (i  < nbe && !trouve)
                if (tab[i].Equals(elem)) trouve=true;
                        else i++;
        return trouve;
        }
```

- Cette méthode utilise la méthode Equals pour comparer 2 objets du type formel T
  - Il faudra que cet opérateur soit défini pour le type effectif utilisé pour instancier la classe Ensemble
    - Type de base (int, float, …)
    - Classe définie par l'utilisateur mais avec une surcharge de la méthode Equals.

Généricité : les modèles de classe

- Utilisation de cette classe générique.
- Instanciation du type formel par un type effectif à la déclaration d'un objet de cette classe
  - Ensemble<int> ensEntier;
  - Instanciation du type T par le type int,
  - Déclaration de la variable ensEntier comme étant un ensemble d'entiers
    - Le type int possède l'opérateur Equals dans la classe associée Int32.

Généricité : les modèles de classe

- Ensemble<Livre> ensLivre = new Ensemble<Livre>();
  - Instanciation du type T par le type Livre,
  - Déclaration de la variable ensLivre.
- Cette instanciation est correcte seulement si l'opérateur "Equals" a été défini dans la classe Livre.

```
public override bool Equals(Object o)
    {  if (o == null) return false;
       Livre l = o as Livre;  // Livre l= (Livre) o;
       return (this.ISBN == l.ISBN);
    }
```


Généricité : les modèles de classe

- Une classe générique peut dépendre de plusieurs paramètres (types formels)

```
class Compose <X, Y>
{
      private X attribut1;
      private Y attribut2;
       public Compose() { …}
       …
};
```

- Compose<int, Livre> cl = new Compose<int, Livre>();

Généricité : les modèles de classe

- Supposons que l'on définisse une classe générique : ListeTriée

```
class ListeTriee<T>
   { …
   }
```

  - qui utilise une méthode CompareTo pour comparer des éléments de la liste

Généricité : les modèles de classe

- L'instanciation de la liste triée par une classe
  - Nécessite que la classe dispose d'une méthode « CompareTo » pour permettre la comparaison des éléments
- Dans ce cas, cela signifie que la classe doit implémenter l'interface IComparable.
- Il est possible de spécifier cette contraintes lors de la description du type générique

Généricité : les modèles de classe

- Contraintes appliquées aux classes génériques
- Elles permettent de spécifier :
  - quelles interfaces doivent implémenter une classe pour être susceptible d'instancier une classe générique.
- Si une méthode de la classe générique utilise un opérateur de comparaison,
  - Il faut que la classe utilisée pour l'instancier implémente la méthode CompareTo (de l'interface IComparable).

Généricité : les modèles de classe

- Il est possible d'expliciter cette contrainte en décrivant la classe générique

```
class ListeTriee<T> where T: IComparable
   { …
}
```

- Indique que le type qui va instancier la classe générique "ListeTriee" devra implémenter l'interface IComparable
  - Qui comporte la signature de la méthode CompareTo.

Les délégués, covariance et contrevariance

- Le mot-clé delegate est utilisé pour déclarer un type référence pour encapsuler une méthode nommée (ou anonyme)
  - Comparable aux pointeurs de fonction de C++
- La covariance et la contrevariance interviennent lors de l'appariement de la signature d'une méthode avec un type délégué.
  - La covariance autorise un type de retour plus dérivé que celui du délégué.
  - La contrevariance autorise qu'une méthode ait des types de paramètres moins dérivés que ceux du type délégué.

Les délégués, covariance et contrevariance

- public class Personne { … }
- public class Employe : Personne
- { …

```
private delegate Personne GestionnairePersonne (Employe e);
private Employe Licencier(Employe p);
private Personne Embaucher(Personne p);
private void demo (…)
{  GestionnairePersonne g1= Licencier; // type de retour plus spécifique
   GestionnairePersonne g2 = Embaucher; // paramètre plus général
      …
 // invocation d'un délégué
 Employe e1 = new Employe(…);
 Employé e2 = g1 (e2);
}
```

Le langage C# De la version 1 à la version 5

- Evolution du langage C#
- Version 1 /1.1 : bases du langage (types valeurs et références, délégués, structures algorithmiques, gestion des exceptions, etc…)
- Version 2 : apparition des types nullables, des collections génériques, des méthodes anonymes, des classes partielles …
- Version 3 : apparition des accesseurs simplifiées, des méthodes partielles, de l'inférence de type, des initialiseurs d'objets et de collections, des méthodes d'extension et des expressions lamba
- Version 4 : apparition des paramètres nommés et optionnels dans les méthodes, le typage dynamique, la covariance et la contrevariance, ainsi que de la programmation parallèle …
- Version 4.5 / 5 : programmation asynchrone (méthodes async, await)