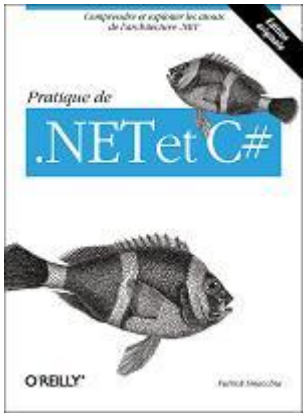
BAT 4 – Polytech'Nice



Le langage C#: Introduction aux Objets

D'après le cours de Patrick Smacchia - Microsoft

Introduction aux objets

- Le concept moteur de la programmation orienté-objet est issue de la création de nouveaux types complexes.

```
struct Simple
   {
            public int Position;
            public bool Exists;
            public double LastValue;
   };

   static void Main()
   {
            Simple s;
            s.Position = 1;
            s.Exists = false;
            s.LastValue = 5.5;
   }
```

Et si je veux ajouter des méthodes (procédures et fonctions) spécifiques au type Simple ?

Exemple :

```
Int deplacement ()
   {
return ( (int) Lastvalue – Position);
}
```

Et donc l'appeler avec s.deplacement()

Du Type à la Classe

- Un type complexe définit les propriétés (champs) d'une variable.
- Comme dans plusieurs langages OO et par extension quand on intègre des méthodes associées à un type (ex. en C#), ce nouveau « type étendu » est représenté par une classe.

```
using system;
class Car
   {
      string color;

      public Car(string color)
      { this.color = color; }

      string Describe()
      {
         return "This car is " + Color;
      }

    }
```

```
namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         Car car;
         car = new Car("Red");
         Console.WriteLine(car.Describe());
      }
   }
```

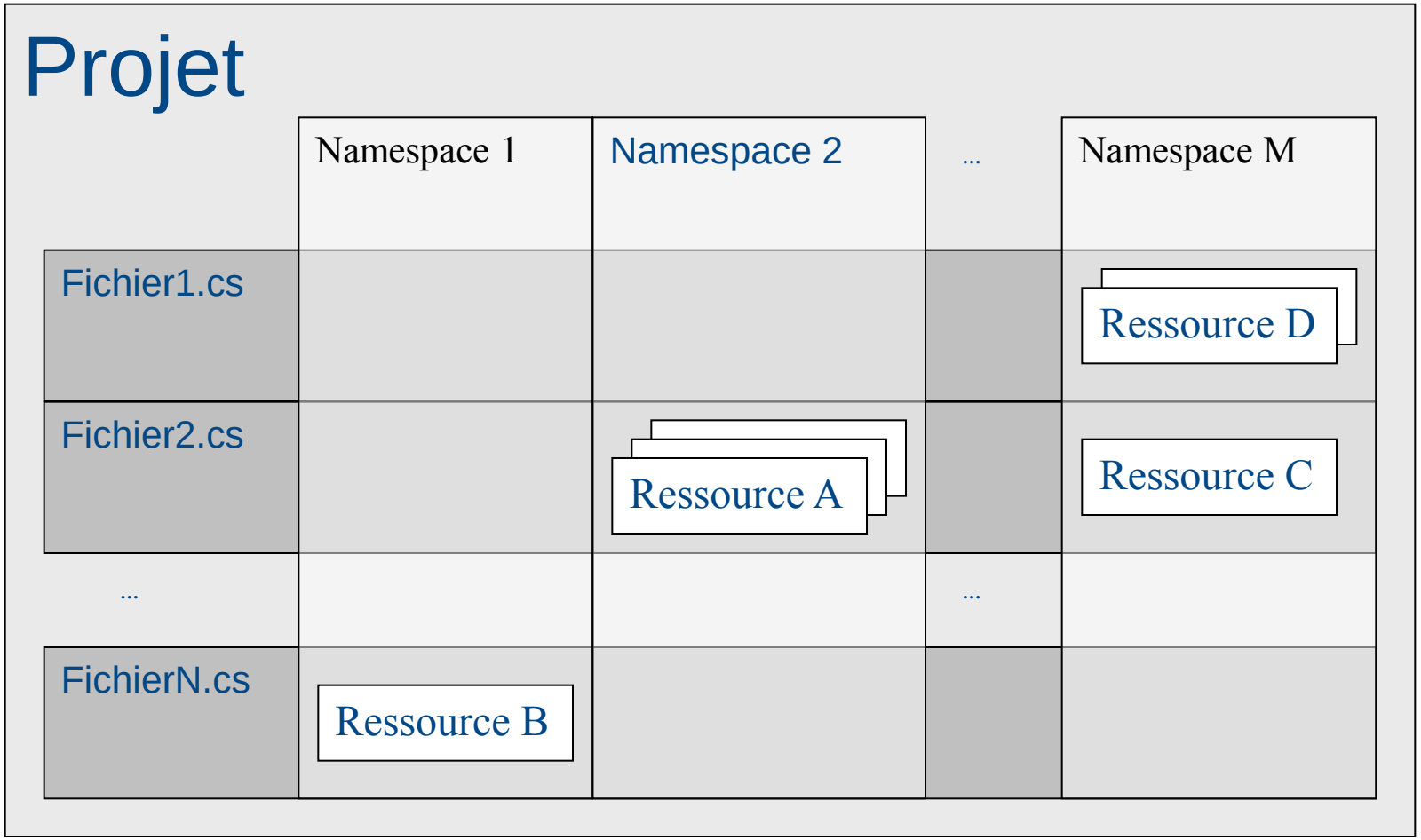
ELÉMENTS DE BASE

Les espaces de noms

- Une ressource dans le code source a la possibilité d'être déclarée et définie à l'intérieur d'un espace de noms.
- Si une ressource n'est déclarée dans aucun espace de noms elle fait partie d'un espace de noms global et anonyme.

```
using Foo1;
using Foo1.Foo2;
using System;
namespace Foo1
{
  // ici les ressources de l'espace de noms
    Foo1
  namespace Foo2
  {
    // ici les ressources de l'espace de noms
    Foo1.Foo2
  }
  // ici les ressources de l'espace de noms
    Foo1
}
```

Organisation du code source



La méthode Main() (1/2)

- Chaque assemblage exécutable a au moins une méthode statique Main() dans une de ses classes.
- Cette méthode représente le point d'entrée du programme, c'est-à-dire que le thread principal créé automatiquement lorsque le processus se lance va commencer par exécuter le code de cette méthode.
- Il peut y avoir éventuellement plusieurs méthodes Main() (chacune dans une classe différente) par programme. Le cas échéant, il faut préciser au compilateur quelle méthode Main() constitue le point d'entrée du programme.

La méthode Main() (2/2)

- Le programme suivant ajoute les nombres passés en arguments en ligne de commande, et affiche le résultat. S'il n'y a pas d'argument, le programme le signale :

```
using System;
class Prog
{
    static void Main(string[] args)
    {
          if (args.Length == 0)
                Console.WriteLine("Entrez des nombres à
    ajouter.");
          else
          {
                long result = 0;
                foreach( string s in args )
                      result += System.Int64.Parse(s);
                Console.WriteLine("Somme de ces nombres
    :{0}",result);
          }
    }
}
```

CLASSES & OBJETS

La Programmation Orientée Objet (POO ou OOP)

- Le concept de la programmation fonctionnelle est construit autour de la notion de fonction. Tout programme est un ensemble de fonctions s'appelant entre elles.
- Le concept de la programmation objet est construit autour des notions d'objet et de classe. Une classe est l'implémentation d'un type de données (au même titre que int ou une structure). Un objet est une instance d'une classe.
- L'avantage majeur de la POO sur la Programmation fonctionnelle est de réunir les données avec leurs traitements.

Classe et membre

- Exemple de déclaration.

```
class MaClasse
{
   // Ici sont placés les membres de la classe MaClasse.
   static void Main(string[] args)
   {
        // Ici sont placés les instructions de la méthode
   Main().
   }
   // Ici aussi, sont placés les membres de la classe
   MaClasse.
}
```

- Les membres d'une classe sont des entités déclarées dans la classe.

Structures vs. Classes

- La déclaration d'une structure se fait avec le mot clé struct à la place de class.
- La principale différence est qu'une structure définit un type valeur alors qu'une classe définit un type référence.
- Une autre différence est que les structures ne connaissent pas la notion d'héritage d'implémentation (voir plus loin).
- Une référence vers une instance d'une structure peut être obtenue avec l'opération de boxing.

Les membres d'une classe

- Six types de membres d'une classe:
  - Les champs : un champs d'une classe définit une donnée (type valeur) ou une référence vers une donnée (type référence) qui existe pour toute instance de la classe.
  - Les méthodes : peut être vu comme un traitement sur les données d'un objet.
  - Les propriétés : peut être vu comme un couple de méthode pour intercepter les accès (lecture/écriture) aux données d'un objet avec la syntaxe d'accès aux champs.
  - Les indexeurs : propriété spéciale utilisant la syntaxe d'accès à un tableau [].
  - Les événements : facilite l'implémentation de la notion de programmation événementielle.
  - Les types encapsulés dans la classe.

Encapsulation et niveau de visibilité

	public	internal protected	internal	protected	private
Méthodes de la classe A	OK	OK	OK	OK	OK
Méthodes d'une classe dérivée de A, dans le même assemblage	OK	OK	OK	OK	
Méthodes d'une autre classe dans le même assemblage	OK	OK	OK		
Méthodes d'une classe dérivée de A dans un assemblage différent	OK	OK			
Méthodes d'une autre classe dans un assemblage différent	OK				

Exemple: Propriétés, champs niveau de visibilité

```
public class UneClasse
{
   // Un champ privé de type int.
   private int m_ChampPrivé = 10;
   public int m_Prop  // Une propriété publique de type int.
   {
        get{ return m_ChampPrivé;}
        set{ m_ChampPrivé = value;}
   }
   static void Main()
   {
        UneClasse A = new UneClasse();
        A.m_Prop = 56;    // L'accesseur set est appelé.
        int i = A.m_Prop;// L'accesseur get est appelé.
   }
}
```

Les constructeurs

- Une méthode est automatiquement appelée lorsque l'objet est construit. On appelle ce type de méthode un constructeur (ctor en abrégé).
- En C#, syntaxiquement, une méthode constructeur porte le nom de la classe, et ne retourne rien (même pas le type void).

Les constructeurs

- Pour chaque classe il peut y avoir :
  - Aucun constructeur : dans ce cas le compilateur fournit automatiquement un constructeur par défaut, qui n'accepte aucun argument.
  - Un seul constructeur : dans ce cas, ce sera toujours lui qui sera appelé. Le compilateur ne fournit pas de constructeur par défaut.
  - Plusieurs constructeurs : dans ce cas ils diffèrent selon leurs signatures (i.e nombre et type des arguments). Le compilateur ne fournit pas de constructeur par défaut.
- Lorsqu'une méthode constructeur retourne, il est impératif que tous les champs de type valeur de la classe aient été initialisés.
- Un constructeur admet un niveau de visibilité.

Le destructeur

- On nomme destructeur la méthode de la classe qui est appelée par le ramasse-miettes juste avant qu'il désalloue la zone mémoire du tas occupée par l'objet.
- Il ne peut y avoir plus d'un destructeur dans une classe, et si le développeur n'en écrit pas, le compilateur crée un destructeur par défaut.
- Un destructeur est une méthode non statique avec le même nom que sa classe, et commençant par le caractère tilde '~'.

Le destructeur

- Syntaxiquement, un destructeur n'admet pas d'argument et ne retourne aucun type, même pas le type void.
- En interne, le compilateur C# fait en sorte qu'un destructeur est la réécriture de la méthode Finalize() de la classe Object.
- Un destructeur n'admet pas de niveau de visibilité.

Exemple: Construction et destruction des objets

```
public class Article
{
    private int Prix;
    public Article(int Prix)    { this.Prix = Prix; }          // ctor 1
    public Article(double Prix) { this.Prix = (int) Prix; }    // ctor 2
    public Article()            { Prix = 0; }                  // ctor 3
    ~Article()                  { /* rien à désallouer!*/ }    // dtor
}
class Prog
{
    static void Main(string[] args)
    {
          Article A = new Article();        // Appelle le ctor 3.
          Article B = new Article(6);       // Appelle le ctor 1.
          Article C = new Article(6.3);     // Appelle le ctor 2.
    }
   // le ramasse miettes appelle le destructeur sur les objets A B et C
   // avant la terminaison du programme
}
```

Les membres statiques

- Les champs, les propriétés, les méthodes (y compris les constructeurs mais pas le destructeur) et les événements d'une classe ont la possibilité d'être déclaré avec le mot-clé static.

Les membres statiques

- Ce mot-clé signifie que le membre appartient à la classe, et non aux objets, comme c'est le cas des membres non statiques.
  - Un membre statique a un niveau de visibilité.
  - Une méthode statique n'a pas accès aux méthodes, champs, propriétés et événements non statiques de la classe.
  - Une méthode statique n'est pas accessible à partir d'une référence vers une méthode de la classe.
  - Une méthode statique est accessible dans toutes les méthodes (statiques ou non) de la classe par son nom, et à l'extérieur de la classe (si son niveau de visibilité le permet) par la syntaxe :
  - NomDeLaClasse.NomDeLaMéthodeStatique()

Exemple: Membres statiques

```
public class Article
{
   static int NbArticles;  // Un champ statique.
   static Article()        // Le constructeur statique.
   {
        NbArticles = 0;
   }
   int Prix = 0; // Un champ privé non statique.
   public Article( int Prix )// ctor non statique
   {
        this.Prix = Prix;
        NbArticles++;
   }
   ~Article()  // dtor
   {
        NbArticles--;
   }
}
```

Le mot clé this

- Dans toutes les méthodes non statiques, C# permet d'utiliser le mot-clé this.
- Ce mot-clé est une référence vers l'objet sur lequel opère la méthode courante.
- L'utilisation de ce mot clé témoigne en général d'une architecture objet évoluée :

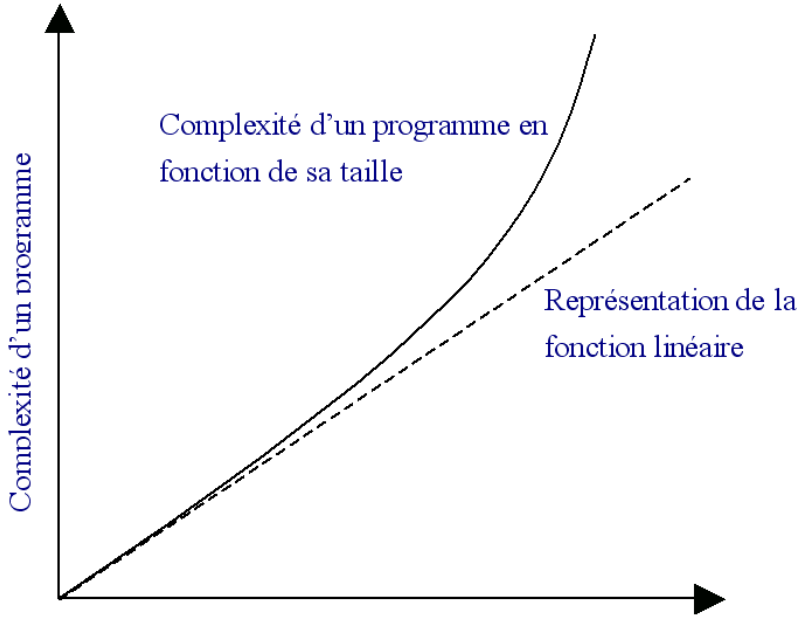
```
class UneClasse
{
    void fct()
// méthode non statique
    {
          fct2(this);
    }
    static void fct2(UneClasse A )
    {

// travaille avec l'objet référencé
// par A...
    }
}
```

HÉRITAGE, POLYMORPHISME & INTERFACES

La problèmatique (1/2)

- La complexité d'un programme en fonction de sa taille, grandit plus vite qu'une fonction linéaire de sa taille.



- Le travail de l'architecte (le *designer*) d'un programme consiste à ce que la complexité reste aussi proche que possible de la fonction linéaire.
- Le mécanisme de dérivation ou d'héritage, est un des points clés de la programmation objet.

La problèmatique (2/2)

- Souvent, dans une application, des classes ont des membres similaires car elles sont sémantiquement proche:
  - Les classes Secrétaire, Technicien et Cadre peuvent avoir en commun les champs Nom, Age, Adresse et Salaire et les méthodes Evalue() et Augmente().
  - Les classes Carré, Triangle et Cercle peuvent avoir en commun les champs CouleurDuTrait et TailleDuTrait et les méthodes Dessine(), Translation() et Rotation().
  - Les classes Button, ComboBox et EditBox peuvent avoir en commun les champs Position et Enabled et les méthodes OnClick() et OnMouseOver().

La problèmatique (2/2)

- On comprend bien que c'est parce que:
  - Une secrétaire, un technicien ou un cadre est un employé (i.e une spécialisation du concept d'employé).
  - Un carré, un triangle ou un cercle est une forme géométrique.
  - Un bouton, une combo-box ou une edit-box est un contrôle graphique.

La solution : L'héritage

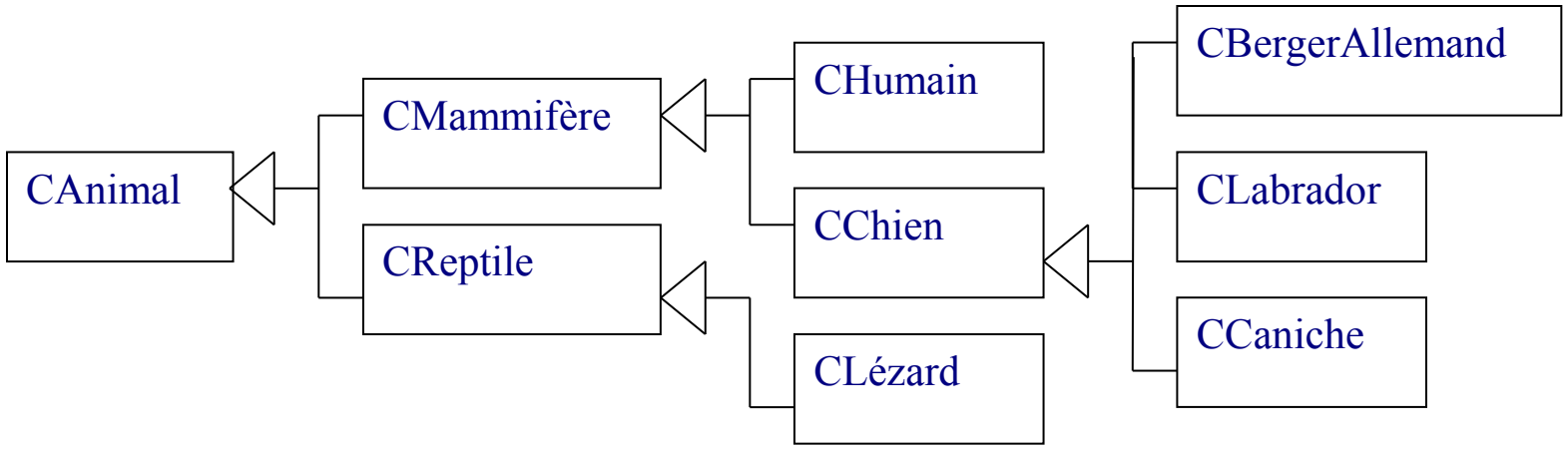
- L'idée de la réutilisation est de cerner ces similitudes, et de les encapsuler dans une classe appelée classe de base (par exemple Employé, FigureGéométrique ou Control).
- Les classes de base sont parfois appelées super classes.
- Une classe dérivée est une classe qui hérite des membres de la classe de base. Concrètement si Technicien hérite de Employé, Technicien hérite des champs Nom, Age, Adresse et Salaire et des méthodes Evalue() et Augmente().
- On dit aussi que la classe dérivée est une spécialisation de la clase de base.

Exemple : L'héritage

```
using System;
class Employé
{
    protected string Nom;
    protected Employé(string Nom)
    {this.Nom = Nom;}
}
class Technicien : Employé
{
    private string Spécialité;
    public Technicien(string Nom, string Spécialité): base(Nom)
    {this.Spécialité = Spécialité;}
}
class Prog
{
    public static void Main()
    {
          Employé E = new Technicien("Roger LaFleur","Electricien");
    }
}
```

Exemple : Schéma de dérivation

- Des normes telles que UML (Unified Modelling Language) permettent de faire des schéma pour représenter les liens entre les classes:



Méthodes virtuelles et polymorphisme (1/3)

- En POO on est souvent confronté au problème suivant :
- On crée des objets, instances de plusieurs classes dérivées d'une classe de base, puis on veut leur appliquer un traitement de base, c'est-à-dire, un traitement défini dans la classe de base.
- Le problème est que ce traitement diffère selon la classe dérivée.
- Par exemple on veut obtenir une description de tous les employés (traitement de base : obtenir une description d'un employé, quelle que soit sa catégorie).

Méthodes virtuelles et polymorphisme (2/3)

```
using System;
public class Employé {
    // m_Nom peut être accédé dans les méthodes des classes dérivées.
    protected string m_Nom;
    public Employé(string Nom) {m_Nom = Nom;}
    public virtual void GetDescription()
    {Console.Write("Nom: {0}",m_Nom);}
}
class Technicien : Employé{  // Technicien hérite de Employé.
    public Cechnicien(string Nom):base(Nom) {}
    public override void GetDescription(){
          // Appel de la méthode GetDescription() de Employé.
          base.GetDescription();
          Console.Write("  Fonction: Cechnicien\n");}
}
class Secrétaire : Employé{  // Secrétaire hérite de Employé.
    public Secrétaire(string Nom):base(Nom) {}
    public override void GetDescription(){
          // Appel de la méthode GetDescription() de Employé.
          base.GetDescription();
          Console.Write("  Fonction: Secrétaire\n");}
}
```

Méthodes virtuelles et polymorphisme (3/3)

- Ce programme affiche:

```
...
class Prog
{
    static void Main(string[] args)
    {
          Employé [] Tab = new Employé[3];
          Tab[0] = new Technicien("Roger");
          Tab[1] = new Secrétaire("Lise");
          Tab[2] = new Technicien("Raymond");
          foreach( CEmployé e in Tab )
                  e.GetDescription();
    }
}
```

```
Nom: Roger  Fonction: Technicien
Nom: Lise  Fonction: Secretaire
Nom: Raymond  Fonction: Technicien
```

La problématique de l'abstraction

- Il arrive que l'on n'ait pas de code à mettre dans la méthode virtuelle parce qu'il y a un manque d'information à ce niveau de l'arbre d'héritage.
- Par exemple pour la classe FigureGéométrique il n'y a rien à mettre dans la méthode virtuelle Dessine(). En effet, à ce niveau de l'héritage on ne sait pas quel type de figure géométrique on instancie.
- Une telle classe de base veut imposer des opérations à ces classes dérivées, alors qu'elle même n'a pas assez d'informations pour effectuer ces opérations, même en partie.

L’abstraction

- Une classe abstraite est une classe qui doit déléguer complètement l’implémentation de certaines de ces méthodes à ses classes dérivées.
- Ces méthodes qui ne peuvent être implémentées s’appellent des méthodes abstraites (ou virtuelles pures).
- La conséquence fondamentale: une classe abstraite n’est pas instanciable.
- Une autre conséquence est qu’une méthode abstraite ne doit pas avoir une visibilité privée.

Exemple: L'abstraction

```
abstract class FigureGéométrique
{
   public abstract void Dessine();
}
class Cercle : FigureGéométrique
{
   private       Point    Centre;
   private       double   Rayon;
   public Cercle( Point Centre, double Rayon)
   {this.Centre = Centre;this.Rayon  = Rayon;}
   public override void Dessine ()
   {
        // dessine un Cercle à partir de son centre et de son rayon
   }
}
...
// erreur de compilation !
// ne peut instancier une classe abstraite !
FigureGéométrique UneFigure = new CFigureGéométrique();
// OK
FigureGéométrique UneFigure = new Cercle(new Point(1,2),3);
...
```

Les interfaces

- Il existe des classes abstraites très particulières. Ce sont celles qui n'ont que des méthodes abstraites.
- En POO on les appelle les interfaces ou abstraction.
- On dit qu'une classe implémente une interface au lieu de 'dérive d'une interface'.
- Le point fondamentale: une classe peut implémenter plusieurs interfaces (et/ou dériver d'une seule classe de base).
- Dans la déclaration d'une interface, les méthodes ne peuvent avoir de niveau de visibilité. C'est aux classes qui l'implémentent d'en décider.

Exemple: Les interfaces

```
using System;

interface IA{void f(int i);}
interface IB{void g(double d);}
class C : IA, IB
{
   public void f(int i)    { Console.WriteLine("f de C {0}",i);}
   public void g(double d){ Console.WriteLine("g de C {0}",d);}
}
class Prog
{
   static void Main(string[] args)
   {
         // une référence interface sur un objet
         IA Obj1 = new C();
         // une référence interface sur un objet
         IB Obj2 = new C();
         // récupération de l'objet d'après une référence interface
         C  RefSurObj2 = (C) Obj2;
         Obj1.f(5);
   }
}
```

Les opérateurs is et as

- L'opérateur is sert à déterminer à l'exécution si une expression peut être transtypée (castée) dans un type donné. Cet opérateur retourne un booléen.
- Après avoir déterminé à l'exécution si une expression peut être transtypée (castée) dans un type donné avec l'opérateur is, on réalise effectivement le transtypage la plupart du temps. L'opérateur as permet d'effectuer ces deux étapes. Si le transtypage ne peut avoir lieu, la référence nulle est retournée.

Exemple: Les opérateurs is et as

```
...
static void Main(string[] args)
{
    IA              RefA              = new C();
    IB              RefB              = null;
    IEnumerator     RefE              = null;
    C               RefC              = null;

    // Ici, le transtypage peut se faire.
    if( RefA is C ){ RefC = (C)RefA; // utilise RefC... }
    // équivalent à
    RefC = RefA as C;
    if( RefC != null ){// utilise RefC...}

    // Ici, le transtypage ne peut pas se faire.
    if( RefA is IEnumerator ){ RefE = (IEnumerator)RefA; // utilise
    RefE... }
    // équivalent à
    RefE = RefA as IEnumerator ;
    if( RefE != null ){// utilise RefE...}
}
```

EXCEPTIONS

La problématique (1/2)

- Les applications doivent faire face à des situations exceptionnelles indépendantes du programmeur. Par exemple :
  - Accéder à un fichier qui n'existe pas ou plus.
  - Faire face à une demande d'allocation de mémoire alors qu'il n'y en a plus.
  - Accéder à un serveur qui n'est plus disponible.
  - Accéder à une ressource sans en avoir les droits.
  - Entrée d'un paramètre invalide par l'utilisateur (une date de naissance en l'an 3000 par exemple).
- Ces situations, qui ne sont pas des bugs mais que l'on peut appeler erreurs, engendrent cependant un arrêt du programme, à moins qu'elles ne soient traitées.

La problématique (2/2)

- Pour traiter ces situations on peut tester les codes d'erreur retournés par les fonctions critiques, mais ceci présente deux inconvénients :
  - Le code devient lourd, puisque chaque appel à une fonction critique est suivi de nombreux tests. Les tests ne sont pas centralisés.
  - Le programmeur doit prévoir toutes les situations possibles dès la conception du programme. Il doit aussi définir les réactions du programme et les traitements à effectuer pour chaque type d'erreur. Il ne peut pas simplement factoriser plusieurs types d'erreur en un seul traitement.
- En fait ces inconvénients sont majeurs.

La solution: les exceptions

- Voici les étapes dans la gestion d'une exception :
  - Une erreur survient ;
  - On construit un objet qui contient, éventuellement, des paramètres descriptifs de l'erreur. La classe d'un tel objet est obligatoirement fille de la classe System.Exception (directement ou indirectement) ;
  - Une exception est lancée. Elle est paramétrée par l'objet ;
  - Deux possibilités peuvent alors survenir à ce moment :
    - A) Un gestionnaire d'exception rattrape l'exception. Il l'analyse et a la possibilité d'exécuter du code, par exemple pour sauver des données ou avertir l'utilisateur.
    - B) Aucun gestionnaire d'exception ne rattrape l'exception. Le programme se termine.

Exemple: les exceptions

```
using System;

public class Prog
{
    public static void Main()
    {
          try
          {
            int i = 1;
            int j = 0;
            int k = i/j;
          }
          catch(System.DivideByZeroException)
          {
            Console.WriteLine("Une division entière par zéro a eu lieu!");
          }
    }
}
```

Le gestionnaire d'exception

- Un gestionnaire d'exception peut contenir une ou plusieurs clauses catch, et/ou une clause finally.
- La clause finally est toujours exécutée, quelle que soit l'issue.
- Une exception lancée, instance d'une classe dérivée D de la classe de base B, matche à la fois la clause catch(D d) et la clause catch(B b). D'où:
  - Dans le même gestionnaire d'exception, le compilateur interdit les clauses catch de classes dérivées de B, après la définition d'une clause catch(B b).
  - La clause catch(System.Exception) rattrape toutes les exceptions puisque toutes les classes d'exception dérivent de la classe System.Exception. Si elle est présente dans un gestionnaire d'exception, elle constitue forcément la dernière clause catch.

Exceptions propriétaires

- Le framework .NET définit de nombreuses exceptions mais vous pouvez définir vos propres classes d'exceptions (qui doivent elles aussi dériver de la classe System.Exception).

```
using System;
public class ExceptionArgEntierHorsLimite : Exception{
    public ExceptionArgEntierHorsLimite( int Entier , int Inf , int Sup ):
    base(string.Format("L'argument {0} est hors de l'intervalle [{1},{2}]",
                                        Entier,Inf,Sup)){}
}
class Prog{
    static void f(int i)
    {if( i<10 || i>50 )throw new ExceptionArgEntierHorsLimite(i,10,50);}
    public static void Main()
    {
      try{f(60);}
      catch(ExceptionArgEntierHorsLimite e)
      {Console.WriteLine("Exception: "+e.Message);}
    }
}
```

AUTRES ÉLÉMENTS DE C#

Les tableaux

- C# permet la création et l'utilisation de tableaux à une ou plusieurs dimensions.

```
int [] r1;   // r1 est une référence vers un tableau d'entiers de dimension 1
int [,] r2;  // r2 est une référence vers un tableau d'entiers de dimension 2
double [,,,] r4;//r4 est une référence vers un tableau de doubles dimension 4
```

- Les types de tableaux sont tous des types référence. En fait chaque type de tableau est une classe qui hérite de la classe abstraite System.Array.
- C# oblige tous les éléments d'un tableau à avoir le même type. Cette contrainte peut être facilement contournée. Il suffit de spécifier que les éléments sont des références vers une classe de base, pour qu'en fait, chaque élément puisse être une référence vers un objet de n'importe quelle classe dérivée.

Le mot clé lock

- Le langage C# présente le mot-clé lock qui remplace élégamment l'utilisation des méthode Enter() et Exit() de la classe System.Threading.Monitor qui définissent une section critique (i.e une portion de code accessible par un seul thread à la fois).

-

```
lock( typeof(Prog) )
{
    Compteur*=Compteur;
}
```

⇔

```
try
{
    Monitor.Enter( typeof(Prog) );
    Compteur*=2;
}
finally
{
    Monitor.Exit( typeof(Prog) );
}
```

Le préprocesseur

- Toute compilation d'un fichier source est précédée d'une phase de mise en forme du fichier. Celle-ci est effectuée par un préprocesseur.
- Il n'effectue que des traitements simples de manipulation textuelle. En aucun cas le préprocesseur n'est chargé de la compilation.
- Toutes les directives préprocesseur sont précédées par le caractère #.
- Le préprocesseur reconnaît les directives suivantes : #define #undef #if #elif #else #endif #error #warning #line #region #endregion

Les délégués et les évènements

- C# permet la création de classes particulières avec le mot-clé delegate en post-fixe. On appelle ces classes délégations. Les instances des délégations sont appelées délégués.
- Conceptuellement, un délégué est une référence vers une ou plusieurs méthodes (statiques ou non). On peut donc ‘appeler’ un délégué avec la même syntaxe que l’appel d’une méthode. Ceci provoque l’appel des méthodes référencées.
- La notion d’événement utilise les délégués pour stocker les méthodes à appeler lorsque l’événement est déclenché.

Surcharge des opérateurs

- C# permet la surcharge d'opérateurs dans des classes:
  - Les opérateurs arithmétiques:
  - unaires: + - ! ~ ++ --
  - binaires:+ - * / % & |^ << >>
  - Les opérateurs de conversion de type (appelés aussi opérateurs de transtypage.
  - Les opérateurs de comparaison:
  - == != < > <= >=

Mode non protégé (1/2)

- Le CLR s'occupe entièrement de la gestion de la mémoire (ramasse-miettes, allocations…)
- Mode non protégé = possibilité de désactiver momentanément la gestion de la mémoire par le CLR
- Une conséquence: En mode protégé on peut utiliser des pointeurs

Mode non protégé (2/2)
Exemple d'utilisation: traitement d'images

Facteur = x 97

Mode géré

```
...
for( int y=0 ; y<Y ; y++ )
  for( int x=0 ; x<X ; x++ ){
    cSrc = m_Bmp.GetPixel(x,y) ;
    cDest= Color.FromArgb(
 255-cSrc.R,255-cSrc.G,255-cSrc.B);
    m_Bmp.SetPixel(x,y,cDest) ;
  }
```

Mode non protégé

```
...
unsafe{
 StructPixel *pBmp
    =(StructPixel*)BmpData.Scan0;
 for( int y=0 ; y<Y ; y++ ){
  pCurrent = pBmp + y*X;
  for( int x=0 ; x<X ; x++ ){
   pCurrent->R=(byte)(255 - pCurrent->R);
   pCurrent->G=(byte)(255 - pCurrent->G);
   pCurrent->B=(byte)(255 - pCurrent->B);
   pCurrent++ ;
  }
 }
}
...
```