

Mise en commun de certaines questions

MISE EN COMMUN DE CERTAINES QUESTIONS.....	1
NEW – VIRTUAL – OVERRIDE (MIKA).....	1
INTERFACES, CLASSES ABSTRAITES ET MÉTHODES ABSTRAITES.....	4
A. CLASSES ABSTRAITES ET MÉTHODES ABSTRAITES (PIET).....	4
B. LES INTERFACES (ECHO).....	5
DÉLÉGUÉS (NONOFF).....	7
LES THREADS (PENCHU).....	8
A. CRÉER ET LANCER UN THREAD.....	8
B. PASSER DES PARAMÈTRES A UN THREAD.....	9
C. PROTÉGER DES RESSOURCES CRITIQUES.....	10
D. STOPPER DES THREADS.....	11
QUESTION 2 DE JANVIER 2004 (NIGHTS).....	11

New – virtual – override (mika)

new, virtual, override. Ces modificateurs concernent l'héritage. Lorsque le mot clef **virtual** est utilisé pour déclarer une méthode d'une classe de base, cela signifie qu'elle est surchargée dans les classes dérivées. Dans les classes dérivées, la surcharge de la méthode de la classe de base doit être déclarée en utilisant le mot clef **override**. Notons que pour qu'une méthode puisse surcharger une autre méthode, la méthode surchargée ne doit pas être de type static et qu'elle doit être déclarée en utilisant le mot clef **virtual**, **abstract** ou **override**.

L'opérateur **new** permet à une méthode d'une classe dérivée de cacher une méthode d'une classe de base sans pour autant nécessiter une déclaration particulière pour la méthode cachée.

Supposons une classe de base appelée rectangle et une classe dérivée appelée carré. Supposons dans la classe de base une méthode appelée surface qui doit être surchargée dans la classe dérivée.

- Utilisation des mots clefs virtual et override:

```

1     namespace ConsoleApp
2     {
3
4         class Rectangle
5         {
6             protected float longueur;
7             protected float largeur;
8             public Rectangle(float largeur, float longueur)
9             {
10                this.largeur = largeur;
11                this.longueur = longueur;
12            }
13            public float surface()
14            {
15                return largeur*longueur;
16            }
17        }
18        class Carre:Rectangle
19        {
20            public Carre(float largeur):base(largeur,0)

```

```

21         {
22
23         }
24     public float surface()
25     {
26         return largeur*largeur;
27     }
28 }
29
30 class Class1
31 {
32     static void Main(string[] args)
33     {
34         Carre ca = new Carre(10.0F);
35         Rectangle ra = ca;
36         Console.WriteLine("surface:"+ra.surface().
ToString());
37         Console.WriteLine("surface:"+ca.surface().
ToString());
38     }
39 }
40 }

```

Commentaires:

Pour rappel, en C++, un pointeur d'une classe de base peut contenir l'adresse d'une classe dérivée. En C#, une référence d'une classe de base peut pointer vers un objet d'une classe dérivée. C'est ce que l'on retrouve à la ligne 35. La question à se poser est de savoir quelle est la méthode qui sera appelée à la ligne 36: est ce la méthode liée au type de la référence ou au type de l'objet sur lequel la référence pointe.

L'exécution du code nous donne l'affichage suivant:

```

surface:0
surface:100
Press any key to continue

```

C'est bien le type de la référence et non pas le type de l'objet référencé. (On pourrait penser que comme on a la ligne 35, on utiliserait à chaque fois la méthode de la classe Carré mais il n'en est rien). La méthode exécutée est celle de la classe utilisée pour déclarer les objets ca et ra : avec ca, on obtient le carré de la largeur=100 ; avec ra, on obtient largeur*longueur=0 (voir constructeur de la classe dérivée). Si nous voulons que le choix de la méthode se fasse sur base du type de l'objet pointé, nous devons utiliser le mot clef *virtual* pour définir la méthode de la classe de base et obtenir la syntaxe suivante:

```
virtual public float surface()
```

L'analyse de l'affichage résultant de la compilation nous donne l'avertissement suivant:

```
warning CS0114: 'ConsoleApp.Carre.surface()' hides inherited member
'ConsoleApp.Rectangle.surface()'. To make the current member override that implementation,
add the override keyword. Otherwise add the new keyword.
```

Actuellement, ce message n'a pas beaucoup de signification car nous n'avons pas encore dévoilé les particularités de l'opérateur new dans la déclaration d'une méthode.

Si nous nous limitons à l'usage de l'opérateur *virtual* sur la méthode de la classe dérivée, nous obtenons à la compilation le message suivant:

```
warning CS0108: The keyword new is required on 'ConsoleApp.Carre.surface()' because it
hides inherited member 'ConsoleApp.Rectangle.surface()'
```

L'exécution du code donne le même résultat. Il faut donc ajouter le mot clef *override* dans la déclaration de la surcharge sous la forme:

```
override public float surface()
```

L'exécution de notre programme nous permet maintenant d'obtenir l'affichage suivant:

```
surface:100  
surface:100  
Press any key to continue
```

Allez, un autre exemple plus explicite avec Toto (regardez les commentaires pour mieux comprendre) :

```
class vertu {  
    public static void Main (String []args) {  
        Personne p = new Personne();  
        Toto t = new Toto();  
        Personne tp = new Toto();  
  
        Console.WriteLine ("p dit :" + p.TuEsQui()); //-> pers  
        Console.WriteLine ("t dit :" + t.TuEsQui()); //-> toto  
        Console.WriteLine ("tp dit :" + tp.TuEsQui()); //-> pers pas bon !!  
  
        Console.WriteLine ("p dit :" + p.WhoRU()); //-> pers  
        Console.WriteLine ("t dit :" + t.WhoRU()); //-> toto  
        Console.WriteLine ("tp dit :" + tp.WhoRU()); //-> toto ok !  
    }  
}  
  
class Personne {  
    public Personne() {}  
  
    public String TuEsQui() {  
        return "je suis une personne";  
    }  
    public virtual String WhoRU() {  
        return "I'm a person";  
    }  
}  
  
class Toto : Personne {  
    public String TuEsQui() {  
        return "je suis un Toto !!";  
    }  
    public override String WhoRU() {  
        return "I'm a Toto !!";  
    }  
}
```

Pour résumer l'utilisation de ces 2 mots-clé : Le mot clé **virtual** indique que la méthode est virtuelle et peut donc être surchargée. La surcharge est effective au moyen du mot clé **override**.

- **Utilisation du mot clef new**

Dans le premier exemple de notre classe Carre, la méthode surchargée cache par défaut la méthode de la classe de base. L'utilisation du mot clef *new* a le même effet excepté que le compilateur ne génère plus de message d'attention. Ce mot-clé permet donc d'être plus précis dans

son code, et de spécifier explicitement qu'on utilisera la méthode de la classe dérivée Carre lorsqu'on créera un objet Carre et qu'on appellera cette méthode.

Interfaces, classes abstraites et méthodes abstraites

A. Classes abstraites et méthodes abstraites (piet)

Les classes abstraites sont étroitement liées aux interfaces. Ces classes ne peuvent pas être instanciées et sont dans de nombreux cas soit partiellement implémentées soit pas implémentées du tout. Les classes abstraites et les interfaces se différencient principalement par le fait qu'une classe peut implémenter un nombre illimité d'interfaces, mais ne peut hériter que d'une seule classe abstraite (ou de toute autre type de classe). Une classe dérivée d'une classe abstraite peut toujours implémenter des interfaces. Les classes abstraites sont utiles lorsque vous créez des composants, parce qu'elles vous permettent de spécifier un niveau constant de fonctionnalités dans certaines méthodes, mais de différer l'implémentation des autres méthodes jusqu'à ce qu'une implémentation spécifique de cette classe soit nécessaire. Elles se prêtent également très bien à la création de versions, parce que si des fonctionnalités supplémentaires sont nécessaires dans les classes dérivées, elles peuvent être ajoutées à la classe de base sans impact sur le code.

Exemple :

```
using System;

class ClassePrincipale : Object
{
    public static void Main()
    {
        ClasseTest test=new ClasseTest();
        test.Méthode2();
    }
}

/*
** Une classe qui possède une méthode abstraite doit elle même
** être déclarée abstraite.
*/
abstract class ClasseAbstraite
{
    public void Méthode1()
    {
    }

    public abstract void Méthode2();
    /*
    ** Une sous-classe de "ClasseAbstraite" doit implémenter une
    ** méthode "Méthode2", sans argument et sans valeur de retour.
    */
}

class ClasseTest : ClasseAbstraite
{
    public override void Méthode2()
        //override sert à indiquer une redéfinition de méthode.
    {
    }
}
```

```
        Console.Out.WriteLine("Méthode2.");  
    }  
}
```

On donne aussi ceci dans la msdn :

```
abstract class WashingMachine  
{  
    public MachineALaver()  
    {  
        // le code pour initialiser la classe vient ici.  
    }  
  
    abstract public void Lave();  
    abstract public void Rince(int QuantiteChargement);  
    abstract public long Tourne(int vitesse);  
}  
  
class MaMachineALaver : MachineALaver  
{  
    public MaMachineALaver ()  
    {  
        // code d'initialisation ici.  
    }  
  
    override public void Lave()  
    {  
        // code pour laver.  
    }  
  
    override public void Rince(int QuantiteChargement)  
    {  
        // Code pour rince  
    }  
  
    override public long Tourne(int vitesse)  
    {  
        // code pour tourne.  
    }  
}
```

Lorsque vous implémentez une classe abstraite, vous devez implémenter chaque méthode abstraite qu'elle contient ; en outre, chaque méthode implémentée doit accueillir le même nombre et le même type d'arguments et avoir la même valeur de retour que la méthode spécifiée dans la classe abstraite.

B. Les interfaces (echo)

Il existe des types référence tel que string, int,... Objet. Objet est une classe de base de quoi dérive tout les autres. On peut cependant créer un nouveau type en utilisant interface (class ou delegate).

Une interface peut contenir des méthodes, des propriétés, des indexeurs et des évènements (car il n'y a rien dans l'interface même).

Les interfaces profitent du fait que l'objet représenté par l'interface héritera forcément de System.Object. C'est donc tout naturellement qu'une interface implémente les méthodes GetType, ToString etc... de System.Object

Exemple d'interface :

```
interface IExempleinterface
{
    //exemple de déclaration de propriété
    Int testproperty { get ;}
    //exemple de déclaration d'événements
    Event testevent Changed ;
    //exemple de déclaration d'indexeur
    String this[int index]
    { get ; set ; }
}
```

Etant donné qu'une interface est un « contrat », toute classe qui implémente une interface doit définir obligatoirement tous les éléments de cette interface.

Elles permettent aussi d'implémenter des comportements au sein d'une même classe. Vous pouvez dériver une classe qui comporte un nombre quelconque d'interfaces.

Le but des interfaces est de définir le contrat "minimum" à remplir des classes qui seront utilisées, pour que toi dans ton code tu n'ais jamais à faire un seul "cast" . Parce que pour toi, tu as uniquement besoin d'un IExempleinterface, le reste, le comment ça marche tu t'en balances.

La seule façon de mettre en œuvre un héritage multiple est de passer par les interfaces.

Il faut aussi savoir qu'une interface est une classe qui ne contient que des méthodes abstraites, qui rappelons le est une classe héritées d'une classe dérivée et pouvant avoir plusieurs instances.

C'est pourquoi si l'on reprend l'exemple du cours

```
interface Iformes
{
    double surface();
    double perimetre();
}

class carre:Iformes
{
    double cote;
    public carre(double cote)
    {
        this.cote=cote;
    }
    public double surface()
    {
        return (cote*cote);
    }
    public double perimetre()
    {
        return (cote*4);
    }
}

class Class1
{
    static void Main(string[] args)
    {
        carre ca = new carre(10.5);
    }
}
```

```
        Console.WriteLine (ca.surface());
        perimetre (ca);
    }
    static void perimetre (Iformes x)
    {
        Console.WriteLine (x.perimetre());
    }
}
```

Dans la class carre qui hérite de l'interface Iformes, on est OBLIGE de déclarer la méthode surface () et perimetre () .

L'avantage donc d'une interface est qu'elle peut créer des références sur n'importe qu'elle instance de la classe (hérité). Ici on aura donc « Iformes » en tant que paramètre du la fonction perimetre.

On peut forcer le faite qu'une interface soit appelé par une référence en mettant dans les methodes de la class hérité (ici carre :Iformes) une référence sur l'interface (exemple sur surface ()) ... ce qui donnerais donc « public double Iformes :surface() ». Le problème c'est que maintenant l'appel à la fonction carre génèrera une erreur car il y a une OBLIGATION de passé par référence... ce qui donnera que dans la classe principale (class1) il faudra mettre

```
carre ca = new carre(10.5);
Iformes test=ca;
Console.WriteLine (test.surface());
```

Si on reprend le Garbage Collector l'évenements *IDisposable* est une interface. Si on voudrais implémenter la méthode Dispose () il faudrait faire hériter notre classe de cette interface.

On pourrait donc dire en conclusion que l'intérêt de faire des interfaces, c'est d'avoir un type d'objet "générique", que tu vas pouvoir manipuler sans te soucier de comment il travaille en interne.

Délégués (nonoff)

```
// appel de methodes statiques et d'instances a partir des délégués
using System;

// declaration du délégué
delegate void MyDelegate();

public class MyClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("Message de la méthode d'instance.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("Message de la méthode statique.");
    }
}

public class MainClass
{
```

```

static public void Main()
{
    MyClass p = new MyClass();
    // on lie le délégué à la méthode d'instance
    MyDelegate d = new MyDelegate(p.InstanceMethod);
    d();

    // on lie le délégué à la méthode statique
    d = new MyDelegate(MyClass.StaticMethod);
    d();
    Console.ReadLine();
}
}

/* Aide dans la msdn :
http://msdn.microsoft.com/library/fre/default.asp?url=/library/FRE/csref/html/vcwlkdelegatetutorial.asp
* Une déclaration delegate définit un type qui encapsule une méthode avec un ensemble d'éléments et un type de retour particuliers.
* Pour les méthodes statiques, un objet delegate encapsule la méthode à appeler. Pour les méthodes d'instance, un objet delegate encapsule à la fois une instance et une méthode sur l'instance. Si vous avez un objet delegate et un ensemble d'arguments approprié, vous pouvez appeler le délégué avec les arguments.
* Un délégué possède une propriété intéressante et utile : il ignore la classe de l'objet qu'il référence et ne s'en soucie pas. N'importe quel objet peut convenir ; il faut simplement que les types d'argument et le type de retour de la méthode correspondent à ceux du délégué.
*/
* Les délégués sont utiles quand : *
*     une seule méthode est appelée ;
*     une classe peut avoir plusieurs implémentations de la spécification de la méthode ;
*     l'appelant n'a pas besoin de savoir ou d'obtenir l'objet sur lequel la méthode est définie ;
*/

```

Les threads (penchu)

A. Créer et lancer un thread

Le code suivant montre comment créer et lancer un thread managé :

```

using System;
using System.Threading;

class ThreadedApp
{
    public static void Main()
    {
        // Déclaration du thread
        Thread myThread;

        // Instanciation du thread, on spécifie dans le
        // délégué ThreadStart le nom de la méthode qui
        // sera exécutée lorsque l'on appelle la méthode
        // Start() de notre thread.
        myThread = new Thread(new ThreadStart(ThreadLoop));

        // Lancement du thread
        myThread.Start();
    }
}

```

```
// méthode appelée au lancement du thread
public static void ThreadLoop()
{
    // Tant que le thread n'est pas tué, on travaille
    while (Thread.CurrentThread.IsAlive)
    {
        Thread.Sleep(500);        // Attente de 500 ms
        Console.WriteLine("Je travaille...");    // Affichage
    }
}
}
```

Ce thread affiche " Je travaille " dans la console toutes les 500 millisecondes.

```
myThread = new Thread(new ThreadStart(ThreadLoop));
```

C'est ici que l'on crée notre objet qui va représenter notre tâche. Le constructeur de la classe *Thread* introduit les délégués. Pour résumer, considérons un délégué comme un pointeur de méthode. Le délégué *ThreadStart* prend comme argument le nom d'une méthode que le thread va exécuter lorsqu'il sera lancé. La méthode passée en argument sera appelée lorsque l'on invoquera la méthode *Start* de notre thread. C'est dans cette méthode **ThreadLoop** que le travail que l'on veut paralléliser devra être introduit.

```
// Tant que le thread n'est pas tué, on travaille
while (Thread.CurrentThread.IsAlive)
```

Généralement, on trouve une boucle qui tournera tant que le thread est en vie. Pour faire ce contrôle, on peut utiliser le membre **IsAlive** qui renvoie **vrai** si le thread est toujours en exécution. Le membre statique *Thread.CurrentThread* renvoie la référence sur le thread actuellement en exécution.

On utilise la méthode statique **Sleep** de la classe *Thread* pour ordonner au thread de passer en attente passive durant 500 ms. Un fois ce délai écoulé, on affiche le texte.

B. Passer des paramètres a un thread

La méthode de création de threads avec le délégué *ThreadStart* ne permet pas de passer directement des paramètres. Pour contourner ça, il suffit de créer une classe spécifique qui contient la méthode du thread (**ThreadLoop** dans l'exemple ci-dessus). On utilisera alors les membres de cette classe comme paramètres.

La classe suivante a le rôle de gérer le thread. On trouve dans cette classe la définition de la méthode utilisée par le thread ainsi que des champs membres utilisés comme paramètres. Il suffira alors de modifier ces champs avant de créer notre thread pour les utiliser par la suite dans notre méthode **ThreadLoop**.

```
//Classe de gestion de thread
public class MyThreadHandle
{
    int myParam;        // utilisé comme paramètre

    // Constructeur
    public MyThreadHandle (int myParam)
    {
        this.myParam = myParam;
    }
}
```

```

// Méthode de modification du paramètre
public void SetParam(int param)
{
    this.myParam = param;
}

// Méthode boucle du thread
public void ThreadLoop()
{
    switch (myParam)
    {
        // ...
    }
}
}

```

Dans l'exemple ci-dessus, le membre **myParam** sera utilisé comme paramètre du thread. C'est-à-dire que, via l'accessor **SetParam** ou le constructeur de la classe, il est possible de le modifier. La subtilité réside dans le fait que la méthode du thread **ThreadLoop** est membre de la même classe que **myParam**. Ainsi, notre méthode peut accéder aux différents champs membres que l'on peut créer et modifier à sa guise.

La création du thread est quelque peu modifiée avec cette technique. Voyons comment créer le thread avec un paramètre :

```

//Exemple de création d'un thread avec paramètre
// On crée notre 'manipulateur' de thread en y passant un
// paramètre classique
MyThreadHandle threadHandle = new MyThreadHandle(10);

// On crée notre thread en y donnant comme méthode boucle, une
// méthode membre de notre manipulateur
Thread t = new Thread(new ThreadStart(threadHandle.ThreadLoop));

// La méthode ThreadLoop de l'objet threadHandle est appelée, et myParam
est donc accessible!
t.Start();

```

En premier lieu, il faut créer une instance de la classe **MyThreadHandle**. Le constructeur de cette classe prend en paramètre un entier, qui sera attribué au membre **myParam**. Ensuite, on crée le thread en déléguant la méthode publique **ThreadLoop**, membre de la classe **MyThreadHandle** et appartenant à notre objet **threadHandle**.

En appelant la méthode **Start** de notre thread, la méthode **ThreadLoop** de l'objet **threadHandle** sera exécutée et aura accès au membre **myParam**. Ainsi, nous avons paramétré l'exécution de notre thread.

C. Protéger des ressources critiques

Une ressource est dite *critique* si sa modification ne doit pas être interrompue. Par exemple, considérons une zone de code critique de 5 lignes. Si la tâche se trouve actuellement à la 3ème ligne de cette zone et qu'elle perd le processeur, certaines données seront erronées, ou perdues, voire pire encore. Il faut donc préciser au `CommonLanguageRuntime` qu'une certaine zone de code ne doit pas être interrompue par d'autres tâches.

Pour réaliser ceci, C# propose un mécanisme extrêmement simple avec le mot-clé **lock**. La directive `lock` avant une portion de code ou une déclaration permettra d'éviter le problème de l'exclusion mutuelle (*mutex*). Imaginons que plusieurs tâches accèdent à une ressource critique pour y débiter un montant. La ressource critique se nomme *Solde* :

```
//Exemple de protection d'une zone de code critique simple
private void DebiterCompte(int Montant)
{
    lock (this)           // Le code dans le bloc sera protégé
    {
        Console.WriteLine("Solde avant transaction : " + Solde);
        Console.WriteLine("Montant à débiter : " + Montant);
        Solde = Solde - Montant;           // Code critique
        Console.WriteLine("Solde après transaction : " + Solde);
    }
}
```

Lorsqu'une tâche voudra débiter le compte, elle appellera la méthode **DebiterCompte** en lui passant un montant, et si aucune autre tâche n'est actuellement en train d'exécuter la partie protégée, elle pourra entrer dans la zone de code critique. Sinon, la tâche sera placée dans une file d'attente.

D. Stopper des threads

A tout moment, on peut être amené à vouloir explicitement détruire des threads, par exemple lors de la fermeture du programme, histoire de faire les choses proprement. La méthode la plus sèche pour stopper un thread se nomme **Abort**. Celle-ci tue le thread et lève une exception du type *ThreadAbortException*.

```
myThread.Abort(); // Détruit notre thread
myThread.Suspend(); // Suspend le thread
myThread.Resume(); // Le thread reprend son activité
```

Question 2 de janvier 2004 (nights)

Vous devez mettre en place la gestion d'une base de donnée sous la forme d'un objet de type `DataReader`. La base de données s'appelle `examen.mdb` tandis que le nom de la table à laquelle on désire accéder est `Clients`. Quelles sont les différentes lignes de code que vous devez mettre en place pour pouvoir créer votre objet de type `DataReader` et pouvoir ainsi accéder aux données si vous passez par l'intermédiaire d'un serveur `OleDb`.

Il faut d'abord mettre en place un `oleDbConnection` par méthode graphique ou non en faisant bien attention de paramétrer `ConnectionString` et `DataSource` (à vérifier si fait graphiquement).

```
private System.Data.OleDb.OleDbConnection oleDbConnection1;
this.oleDbConnection1 = new System.Data.OleDb.OleDbConnection();
this.oleDbConnection1.ConnectionString=@"Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=D:\examen.mdb;"
```

Ensuite il faut mettre en place un `oledbcommand`, par méthode graphique ou non.
La propriété `CommandText` contient la requête SQL permettant de recueillir les informations.

```
private System.Data.OleDb.OleDbCommand oledbCommand1;
this.oledbCommand1=new System.Data.OleDb.OleDbCommand()
this.oledbCommand1.Connection=oledbConnection1
//this.oledbCommand1=oledbConnection1.CreateCommand() peut remplacer les //
deux premières lignes
oledbCommand1.CommandText="SELECT * FROM Clients"
```

Il faut ensuite mettre en place le `datareader` qu'il faut instancier manuellement tel que suit :

```
private System.Data.OleDb.OleDbDataReader oledbDataReader1;
```

Ensuite nous pouvons lire les données avec le `datareader` tel que suit :

```
this.oledbConnection1.Open();
this.oledbDataReader1=this.oledbCommand1.ExecuteReader();
while(this.oledbDataReader1.Read())
{
    this.comboBox1.Items.Add(this.oledbDataReader1.GetString(0));
}
this.oledbDataReader1.Close();
this.oledbConnection1.Close();
```

Ici nous avons l'exemple de la lecture du champs de la première colonne de toutes lignes afin de remplir une `combobox` (;).

La ligne 1 permet d'ouvrir la connexion et de se relier à la base de donnée (en temps réel).

La ligne 2 permet au `datareader` d'accéder aux données grâce a notre requête sql placée dans `oledbcommand`.

Ligne 3 : `this.oledbDataReader1.Read()` permet de lire une ligne et de passer à la suivant. La boucle permet donc de lire les lignes une à une tant qu'il y a des données.

Ligne 6 : `this.oledbDataReader1.GetString(0)` récupère la donnée de la colonne 1 sur la ligne lue en cours.

Les deux dernières lignes servent à la fermeture de connexion.

Comment pouvez vous vous prémunir des erreurs provoquées par l'utilisation de la méthode `Read` et plus particulièrement `System.InvalidOperationException`. Que devez vous faire pour pouvoir également en plus de la technique déjà mise en place pour l'erreur évoquée précédemment et sans en modifier le code, assurer également la gestion des autres erreurs. Donner les explications complètes sur ce qui peut être fait dans ces techniques.

La capture des exceptions en C# peut s'effectuer en subdivisant la partie de votre code en trois blocs caractérisés par les mots clefs *try*, *catch* et *finally*.

Le bloc *try* correspond au bloc qui contient les opérations normales pouvant être à l'origine de l'erreur.

Le bloc *catch* correspond au bloc qui contient le code à exécuter en cas d'erreur.

Le bloc *finally* correspond au bloc contenant le code permettant de nettoyer les ressources ou d'effectuer toute action qu vous souhaitez effectuer à la fin d'un bloc *try* ou *catch*. Le contenu du bloc *finally* est exécuté qu'il y ait une condition d'erreur ou pas. Ce bloc est optionnel.

Nous pouvons spécifier des blocs *catch* multiples pour permettre des types différents d'exceptions. Une complication peut surgir du fait que les exceptions forment une hiérarchie d'objets et de ce fait, une exception particulière peut trouver une correspondance avec plus

qu'un seul bloc *catch*. Etant donné que c'est le premier bloc *catch* rencontré correspondant à l'exception générée qui est exécuté, il est important de placer en premier les blocs *catch* correspondant à des exceptions plus spécifiques et ensuite de placer les blocs *catch* correspondant aux exceptions plus générales.

```
try
{
    this.oleDbConnection1.Open();
    this.oleDbDataReader1=this.oleDbCommand1.ExecuteReader();
    while(this.oleDbDataReader1.Read())
    {
        this.comboBox1.Items.Add(this.oleDbDataReader1.GetString(0));
    }
    this.oleDbDataReader1.Close();
    this.oleDbConnection1.Close();
}
catch (System.InvalidOperationException e)
{
    this.textBox1.Text=e.Message;
    //e.message contient le code natif de l'erreur
    //possibilité d'exécution de code
}
catch
{
    this.textBox1.Text= « Erreur indéterminé »;
    //possibilité d'exécution de code
}

this.textBox1.Text= « Fermeture programme » ;
```

Le bloc try contient le bloc de code à exécuter, le premier catch s'exécutera si l'erreur **System.InvalidOperationException** a lieu. Sinon le dernier catch capturera toutes les erreurs autres que la précédente. Ainsi comme il y a une capture d'exception qui est effectuée si une erreur a lieu lors du remplissage de la combobox le programme ne se fermera pas brutalement et exécutera le code contenu dans le catch adéquat (une sauvegarde du travail en cours par exemple).

A noter que dans la gestion d'erreur il est possible de déclencher ses propres erreurs ceci grâce à la commande throw. Imaginons que je veuille provoquer l'erreur **InvalidOperationException** à la fin du remplissage de la combobox même si tout c'est bien passé. Il me suffira de rajouter dans le try lors de la fermeture de la connection :

```
try
{
    this.oleDbConnection1.Open();
    this.oleDbDataReader1=this.oleDbCommand1.ExecuteReader();
    while(this.oleDbDataReader1.Read())
    {
        this.comboBox1.Items.Add(this.oleDbDataReader1.GetString(0));
    }
    this.oleDbDataReader1.Close();
    this.oleDbConnection1.Close();
    throw new InvalidOperationException(« Quoi qu'il arrive ca ne marchera pas !!! ») ;
    //attention de ne pas mettre System.InvalidOperationException
    // ou encore throw new Exception(); dernier catch car pas d'arg.
}
}
```

Ainsi le catch avec `System.InvalidOperationException` attrapera l'erreur et exécutera le code qu'il contient.

Ce qui suit n'a pas été testé :

A noter également les classes d'exception définies par l'utilisateur doivent dériver de la classe de base `ApplicationException`.