

## Examen de programmation (partie théorique)

### *Question 1:*

**Quelle(s) différence(s) y a-t-il entre une boîte de dialogue modale et une boîte de dialogue non modale. Quelles sont les différentes étapes que vous devrez suivre si vous souhaitez créer ces boîtes et les faire apparaître.**

**En supposant que ces boîtes doivent apparaître à la suite d'un clic sur un bouton et que l'on retrouve donc deux boutons dans notre application, l'un possédant comme nom Button1 et l'autre Button2, quelles sont les lignes de code que vous devrez insérer pour y arriver (déclaration des méthodes, gestion des événements...)**

### Différence

#### *Boîte modale (ShowDialog):*

Cette méthode renvoie une valeur d'énumération DialogResult. Elle bloque l'exécution et aucun code ne sera exécuté tant que l'utilisateur n'a pas fermé la boîte de dialogue.

Autrement dit, lors d'un appel d'une boîte de dialogue modale, le code qui la suit n'est pas exécuté tant que la boîte de dialogue n'est pas fermée. On ne peut donc pas retourner à la form1 tant qu'on n'a pas fermé la form2.

#### *Boîte non modale (Show):*

Cette méthode ne renvoie pas de valeur DialogResult et ne bloque pas l'exécution. La boîte est simplement affichée et le code continue son exécution.

Autrement dit, lors d'un appel d'une boîte de dialogue non modale, on peut atteindre la form1 et y exécuter du code même si la form2 est ouverte a été appelée.

### Etapes

Il faut d'abord créer un nouvel objet de type Form2. Une fois la référence initialisée, on appelle la méthode ShowDialog pour le formulaire Form2.

Il faut indiquer la fenêtre propriétaire de laquelle ce formulaire dépendra :

```
- this.AddOwnedForm(Form2);  
- Form2 test = new Form2();           //on instancie  
- test.ShowDialog(this);              //pour un formulaire modal  
- test.Show();                         //pour un formulaire non modal
```

### Déclaration des boutons (dans `public class Form1 : System.Windows.Forms.Form`)

```
private System.Windows.Forms.Button button1;  
private System.Windows.Forms.Button button2;
```

### Evènements

```
private void button1_Click(object sender, System.EventArgs e)  
{  
    Form2 test2 = new Form2();  
    test2.ShowDialog(this);  
}
```

```
private void button2_Click(object sender, System.EventArgs e)  
{
```

```

Form3 test3 = new Form3();
test3.Show();
}

```

### Question 2:

**Veillez par un exemple commenté, mettre en évidence les différences existant entre une méthode dite instanciée et une méthode dite de classe (déclaration, utilisation...).**

Les méthodes instanciées n'existent que si l'on a créé une instance de la classe et elles sont appelées par l'objet lui-même sous la forme : objet.méthode([paramètres]) ;

Les méthodes de classe déclarées avec le mot clef static peuvent être appelées même s'il n'existe aucune instance de la classe sous la syntaxe suivante : Nom\_de\_classe.Méthode ([paramètres]) ;

Prenons comme exemple une classe permettant la gestion des nombres complexes.

Sans mettre en œuvre la surcharge des opérateurs, nous allons intégrer dans notre classe une méthode permettant d'effectuer une opération d'addition sur deux complexes. En C++, nous pourrions utiliser une fonction amie de sorte que les objets passés en paramètre puisse accéder aux membres protégés de la classe mais en C#, cette solution n'est pas autorisée.

Nous devons donc nous orienter vers des méthodes statiques. Un peu dans le même esprit que pour les fonctions amies, l'ensemble des objets doit être passé en paramètre du fait que la méthode statique est appelée par le nom de la classe et pas par une instance.

```
using System;
```

```

namespace ConsoleApp
{
    class Complexe
    {
        public int reel;
        public int image;
        public Complexe(int reel, int image)
        {
            this.reel=reel;
            this.image=image;
        }
        public static Complexe MultComplexe(Complexe x, Complexe y)
        {
            int reel=x.reel*y.reel-x.image*y.image;
            int image=x.reel*y.image+x.image*y.reel;
            Complexe tmp = new Complexe (reel,image);
            return tmp;
        }
    }
    class Class1
    {
        static void Main(string[] args)
        {
            Complexe ca = new Complexe(10,20);
            Complexe cb = new Complexe(20,30);
            Complexe cc = Complexe.MultComplexe(ca,cb);
            Console.WriteLine("{0}+i{1}",ca.reel.ToString(),
                ca.image.ToString());
        }
    }
}

```

Résultat : 10+i20

### Question 3:

**Veillez par un exemple commenté, expliquer l'utilisation des indexeurs dans les classes.**

Un exemple classique de l'utilisation de la surcharge de l'opérateur [] en C++ était la possibilité de pouvoir accéder à un élément d'un vecteur étant lui-même un objet d'une classe. Reprenons cet exemple et adaptions le à la lumière du C#:

```
using System;

class vecteur
{
    int [] tab;
    int taille;
    public int this [int i]           //1
    {
        get
        {
            if (i<taille)
                return tab[i];
            else
                throw new IndexOutOfRangeException("Acces"); //2
        }
        set
        {
            if (i<taille)
                tab[i]= value;
            else
                throw new IndexOutOfRangeException("Acces"); //3
        }
    }
    public vecteur(int taille)
    {
        tab = new int[taille];
        this.taille=taille;
    }
}

class Class1
{
    static void Main(string[] args)
    {
        vecteur x = new vecteur(10);
        x[1]=10;
        x[9]=12;
        Console.WriteLine("x[1]="+x[1]);
    }
}
```

Résultat de l'affichage : X[1]=10 ;

Remarques:

L'utilisation d'un indexeur est fort semblable à la mise en place des propriétés accessibles par les accesseurs set et get (get pour la lecture, set pour l'écriture).

Comme pour les surcharges d'opérateurs, les indexeurs fonctionnent pareillement pour les structures et les classes.

//1 : déclaration de la méthode liée à l'indexeur sous la syntaxe suivante:  
public int this [int i]. L'opérateur this permet de référencer l'instance ayant servi à utiliser l'indexeur. Si nous avons envisagé une classe permettant la gestion d'une matrice avec l'accès à un tableau à deux dimensions, nous aurions eu la syntaxe public int this [int i, int j].

//2 et //3 : syntaxe permettant de générer une exception qui sera prise en charge par le runtime et indiquant qu'il y a eu une tentative d'accès en dehors des limites de notre vecteur. Si dans notre exemple, nous essayons d'entrer la ligne de code suivante: x[10]=20; nous obtiendrons lors de l'exécution le message d'erreur suivant:

« Une exception non gérée du type 'System.IndexOutOfRangeException' s'est produite dans ConsoleApplication1.exe. »

#### **Question 4:**

#### **Veillez par un exemple commenté, expliquer comment vous pouvez mettre en place la génération d'exceptions personnalisées**

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 contient les opérations normales pouvant être à l'origine de l'erreur.

Le bloc catch contient le code à exécuter en cas d'erreur.

Le bloc finally contient le code permettant de nettoyer les ressources ou d'effectuer toute action que vous souhaitez effectuer à la fin d'un bloc try ou catch.

Considérons l'exemple d'une division par zéro.

```
using System;
```

```
class Class1
{
    static void Main(string[] args)
    {
        int a=20;
        int b=0,c;
        try
        {
            c=a/b;
        }
        catch(System.DivideByZeroException e)
        {
            Console.WriteLine("Divison par zéro");
        }
        Console.WriteLine("Fin du programme");
        Console.ReadLine();
    }
}
```

On peut spécifier des blocs catch multiples pour permettre des types différents d'exceptions. Comme 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.

Nous pouvons également générer les exceptions avec l'emploi de throw. Reprenons le code précédent et adaptons le à la lumière de ce qui vient d'être dit.

```
using System;

class Class1
{
    static void Main(string[] args)
    {
        int a=20;
        int b=0,c;
        try
        {
            if (b == 0) throw new DivideByZeroException("division par zéro"); //1
            else c=a/b;
        }
        catch(System.DivideByZeroException e)
        {
            Console.WriteLine("Cause d'exception: "+e.Message); //3
        }
        catch (System.Exception e) //2
        {
            Console.WriteLine("Autre exception: "+e.Message);
        }
        Console.WriteLine("Fin du programme");
        Console.ReadLine();
    }
}
```

Commentaires:

//1 : permet de générer soi même une exception dans le code.

//2 : permet d'intercepter les autres exceptions qui ne sont pas prises en charge par un bloc catch spécifique précédent.

//3 : met en évidence l'accès à une propriété de la classe de base Exception qui est Message permettant d'afficher le texte natif qui décrit la condition d'erreur.

*Classes d'exceptions définies par l'utilisateur.*

Une classe d'exception définie par l'utilisateur doit dériver de la classe de base ApplicationException. Imaginons que l'on désire demander l'introduction au clavier d'un nombre compris entre 0 et 255 et que le non respect de cette restriction génère une exception que nous avons créée.

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        int valeur;
        string Valeur;
        try
        {
            Console.Write("Un nombre entre 0 et 255: ");
            Valeur = Console.ReadLine();
            valeur = Convert.ToInt32(Valeur);
            if ((valeur<0)||(valeur>255))
                throw new OutOfRange("Valeur hors limite");
        }
    }
}
```

```

        catch (OutOfRangeException e)
        {
            Console.WriteLine(e.Message);
        }
        catch (System.DivideByZeroException e)
        {
            Console.WriteLine("Cause d'exception: "+e.Message);
            Console.WriteLine(e.TargetSite);
        }
        catch (System.Exception e)
        {
            Console.WriteLine("Autre exception: "+e.Message);
        }
        Console.WriteLine("Fin du programme");
        Console.ReadLine();
    }
}
class OutOfRange:ApplicationException //1
{
    public OutOfRange(string Message):base(Message)
    {
    }
    public OutOfRange(string Message, Exception
        InnerException):base(Message,InnerException)
    {
    }
}

```

//1 : à partir d'ici, nous retrouvons la création de notre propre exception sous la forme d'une classe héritant de la classe de base ApplicationException. Dans cette classe, nous implémentons deux constructeurs chacun comprenant le message qui devra être fourni en cas d'erreur.

**Pour cet examen:** Aucune compilation ou génération de code par un outil quelconque n'est autorisée. Aucune nouvelle ligne de code ne peut être introduite dans vos applications existantes. Seule les applications existantes (non modifiables) peuvent être consultées ainsi que l'aide MSDN. Vous pouvez également avoir accès à vos notes de cours. Les différentes réponses aux questions doivent être manuscrites.