

## 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:

**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 champ 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 à notre requête SQL placée dans OleDbCommand.

Ligne 3 : this.OleDbDataReader1.Read() permet de lire une ligne et de passer à la suivante. 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 que 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 toute 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 tester :

A noter également les classe d'exception définie par l'utilisateur doit dériver de al classe de base ApplicationException.

### **Question 3:**

**Qu'est ce qu'un délégué dans la philosophie de programmation C#? Comment pouvez vous mettre en place ces délégués? Donner des explications complètes en vous basant sur l'utilisation des threads.**

Les délégués peuvent être considérés comme un nouveau type d'objet de C# (en effet, ils nécessitent une déclaration de type, la création de référence et des instanciations).

Ils remplacent les pointeurs de fonctions que l'ont utilisait en C et C++. Ils ont certaines similitudes avec les classes. Ils interviennent lorsqu'on veut passer des méthodes à d'autres méthodes. Par exemple, la méthode de la classe thread :

En C#, un thread est démarré par la méthode *thread.start()*. Cette méthode a besoin d'un paramètre correspondant à la méthode qui doit être invoquée par le thread, c'est-à-dire que si on demande à l'ordinateur de démarrer un thread, on doit lui indiquer l'emplacement de départ de celui-ci.

Pour passer des méthodes, on doit envelopper les détails de ladite méthode dans un nouveau type d'objet, le délégué.

*Mise en place des délégués.*

On doit d'abord commencer par définir le délégué.

Exemple : définition d'un délégué appelé VoidOperation (avec une niveau de visibilité public). Chaque instance de ce délégué peut contenir une référence à une méthode qui accepte un paramètre uint et retourne void.

⇒ public delegate void VoidOperation (uint x) ;

Cette syntaxe ressemble fort à celle d'une définition de méthode sauf qu'il n'y a pas de corps de méthode et que la définition est précédée du mot clé delegate. On peut définir un délégué partout où on peut définir une classe.

Puis il faut créer une ou plusieurs instances de ce délégué.

```
=> VoidOperation mon_instanciation = new VoidOperation(mon_uint) ;
```

#### *Question 4:*

**Qu'est ce qu'un assemblage? Qu'est ce qui distingue un assemblage privé d'un assemblage partagé? Comment créer et mettre en place ces deux types d'assemblages? Comment utiliser un assemblage privé ou un assemblage partagé dans son projet?**

L'utilisation des bibliothèques partagées Microsoft est la cause de nombreux problèmes. Par exemple, la gestion des versions entre deux bibliothèques de même nom est impossible. Lorsqu'un logiciel installe la version 2 d'un DLL, il écrase la version 1 et impose aux autres logiciels l'utilisation de la nouvelle bibliothèque. Si celle-ci n'est pas pleinement compatible avec la version précédente, les logiciels moins récents ne fonctionneront probablement plus.

Pour pouvoir être accessible, les programmes et les composants doivent se décrire et s'enregistrer dans la base de registres. Les informations stockées deviennent rapidement obsolètes et incohérentes, et la base de registre finit par devenir une véritable usine à gaz souvent ingérable.

Pour résoudre ça, Microsoft a défini 5 fonctionnalités de base pour les bibliothèques .NET :

- **Les applications doivent être auto descriptives** : elles suppriment le lien de dépendance avec le registre, ce qui permet une installation sans impact sur le système.
- **Les informations de version doivent être enregistrées et mises en œuvre** : la prise en charge des versions doit être intégrée dans la plate-forme pour garantir que la version appropriée d'un lien de dépendance est chargée au moment de l'exécution.
- **Mémorisation des composants** : lorsqu'une application s'exécute bien, la plate-forme doit pouvoir mémoriser le jeu de composants qui ont fonctionné ensemble.
- **Prise en charge des composants côte à côte** : la possibilité d'installer plusieurs versions d'un composant sur la machine et de les exécuter simultanément permet aux utilisateurs de spécifier la version qu'ils souhaitent charger. Le Framework permet à plusieurs versions de la structure de travail de coexister sur une même machine.
- **Isolement des applications** : Le Framework doit faciliter l'écriture d'applications qui ne seront pas affectées par des modifications effectuées sur la machine par d'autres applications.

Les bibliothèques .NET, appelées assemblages, possèdent ces fonctionnalités grâce à leur manifeste. Spécifiquement, un manifeste inclut les données suivantes sur l'assemblage :

- **Identité** : comprend un nom, un numéro de version et une culture (facultative).
- **Liste de fichiers** : qui composent l'assemblage.
- **Assemblages référencés** : les liens de dépendance entre les assemblages sont stockés dans le manifeste de l'assemblage appelant. Ces liens incluent un numéro de version pour garantir que la version adéquate du lien de dépendance est chargée.

- **Ressources et types exportés** : Les options de visibilité disponibles pour les types et les ressources sont notamment "visibles uniquement dans mon assemblage" et "visibles pour les appelants externes à mon assemblage".
- **Demandes d'autorisation** : Les demandes d'autorisation pour un assemblage sont regroupées en trois catégories : 1) celles qui sont nécessaires pour que l'assemblage s'exécute, 2) celles qui sont souhaitables mais en l'absence desquelles l'assemblage aura tout de même accès à certaines fonctions, même si ces autorisations ne lui sont pas accordées, et 3) celles dont l'auteur ne veut absolument pas qu'elles soient accordées à l'assemblage.

En général, les assemblages sont composés de quatre éléments : les méta données de l'assemblage (manifeste), les méta données décrivant les types, le code en langage intermédiaire (IL) qui met en œuvre les types, et un ensemble de ressources. Tous ne sont pas présents dans chaque assemblage. Seul le manifeste est absolument obligatoire, mais ni les types ni les ressources ne sont indispensables pour attribuer à l'assemblage une quelconque fonctionnalité.

### Assemblages privés

Les assemblages privés sont normalement livrés avec un logiciel et destinés à être utilisés avec ce logiciel uniquement.

Comme un logiciel s'installe toujours dans son propre répertoire, il n'y a pas de risque d'écraser une dll d'un autre programme, ni de risque de collision de noms. Chaque application ne voit que les classes définies dans l'assemblage privé.

#### Pour mettre en place cet assemblage :

Il faut créer un nouveau projet « bibliothèque de classes ». Une classe est créée par défaut, mais on ajoutera cette méthode:

```
public string MyFunction()
{
    return "Ma fonction version 1.0";
}
```

On va modifier le nom du projet en « Myassembly » et de ce fait, notre fichier de sortie s'appellera Myassembly.dll.

On génère la solution (en mode release). La bibliothèque de classes Myassembly.dll se trouve alors dans le répertoire ../bin/release/.

#### Pour utiliser cet assemblage :

On crée une nouvelle application Windows dans laquelle on met un bouton et une zone de texte qui affichera le résultat de la méthode MyFunction qui se trouve dans l'assemblage.

Il faut copier Myassembly.dll qu'on vient de créer dans le dossier ../bin/debug/ de l'application windows qu'on est en train d'éditer.

Il faut ajouter la référence de notre dll (par le menu projet, ajouter une référence).

Dans le code de notre application, il faut rajouter la ligne suivante :

```
using Myassembly;
```

Voici le code appliqué à l'évènement click du bouton :

```
Class1 test = new Class1();
this.textBox1.Text=test.MyFunction();
```

## Assemblages partagés

Ce sont des bibliothèques communes utilisables par toute application. Il faut donc faire attention à certaines choses :

- la collision de noms, dans le cas où l'assemblage partagé d'un autre programme implémente des types portant les mêmes noms que les vôtres.
- L'écrasement d'un assemblage par une version différente du même assemblage, la nouvelle version étant incompatible avec une partie du code client.

Pour éviter ces risques, il suffit de placer ces assemblages dans une cache d'assemblage. Car à l'inverse des assemblages privés, ils ne peuvent pas simplement être copiés dans le dossier approprié.

Pour éviter les risques de collision de noms, les assemblages partagés se voient attribuer un nom basé sur un cryptage à clé privée.

Pour éviter l'écrasement d'un assemblage, il faut spécifier une information de version dans le manifeste d'assemblage et permettre les installations simultanées.

### Pour mettre en place cet assemblage :

On reprend la bibliothèque de classe qu'on vient de créer. On doit alors créer une paire de clefs en exécutant l'utilitaire « sn.exe » qui se trouve dans c:\program files\visual studio.net 2003\bin. En voici la syntaxe pour notre exemple : « sn -k Mykey.snk »

Il faut alors le placer dans le projet lié à la bibliothèque de classes dans le répertoire /obj/release ou /obj/debug.

Ensuite il faut éditer le fichier AssemblyInfo.cs pour y modifier le code suivant:

Ligne de code original: [assembly: AssemblyKeyFile("")]

Ligne de code modifié: [assembly: AssemblyKeyFile("Mykey.snk")]

On génère la solution, et on copie la dll dans \windows\microsoft.net\framework\v1.1.4322.

Dans le panneau de configuration, on ouvre « Outils d'administration\Microsoft.net framework 1.1 configuration ».

Il faut alors ajouter la dll dans la cache d'assemblage (clic droit sur « cache d'assemblies »).

### Pour utiliser cet assemblage :

On reprend le projet windows en ajoutant une référence à un assemblage partagé.

Il faut ajouter la référence de notre dll (par le menu projet, ajouter une référence).

Ici, la propriété 'copie locale' de la référence ajoutée à notre projet est à false.

Si nous renommons ou supprimons le fichier Myassembly.dll, notre code continue à être exécutable car cet assemblage s'effectue vers la cache globale des assemblages.

**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.