Visual Studio 2013/2015 & Langage C#

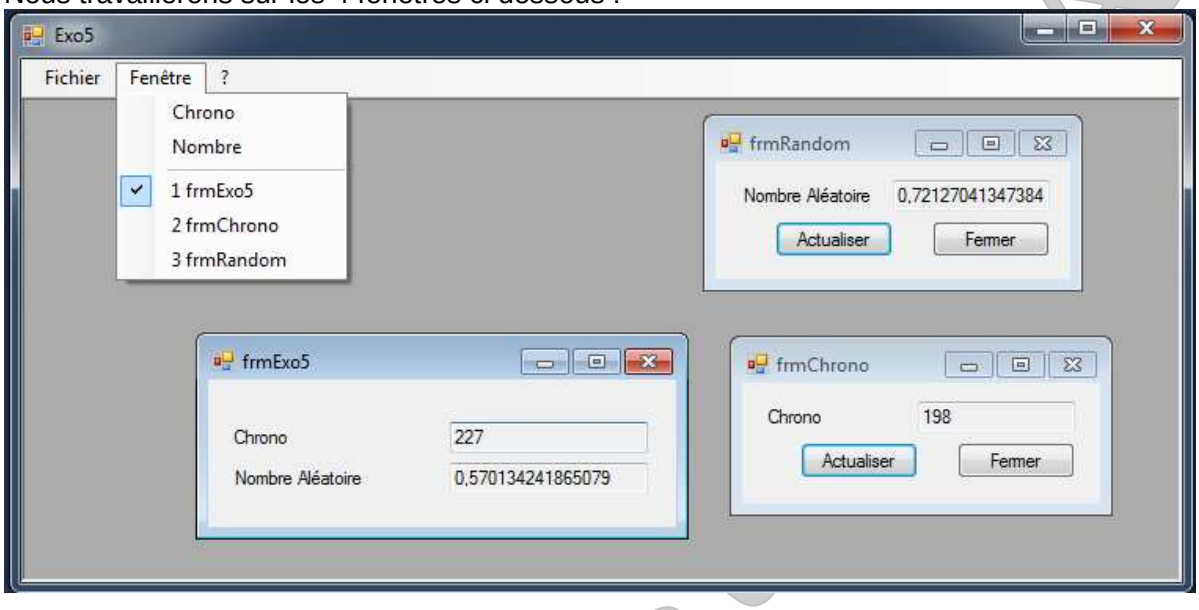
Progression exercice 5

Thèmes abordés:

- **Création de menus.**
- **Application multi-fenêtres.**
- **Fenêtres modales / non modales.**
- **Relations entre classes.**
- **Constructeur.**
- **Visibilité des variables.**
- **Nombres aléatoires.**

La plupart des applications Windows comportent plusieurs fenêtres, nous allons donc apprendre à créer plusieurs fenêtres et à les gérer correctement.

Nous travaillerons sur les 4 fenêtres ci dessous :



La fenêtre principale est celle qui contient toutes les autres. Nous l'appellerons `frmMDI` car c'est une en rapport avec ce type d'application Windows *'Multiple Document Interface'*. C'est elle qui doit supporter le menu de l'application ; elle ne contient rien d'autre que des fenêtres et elle se reconnaît à son fond grisé plus sombre (obtenu automatiquement dès lors que l'on place à `True` sa propriété `IsMdiContainer`).
Les menus sont réalisés assez facilement et intuitivement grâce à l'outil `MenuStrip`.
Le menu Fenêtres/Chrono ouvre la feuille `frmChrono`. Le menu Fenêtres/Nombre ouvre la feuille `frmRandom`. La liste des fenêtres ouvertes est obtenue automatiquement en ajustant la propriété `MdiWindowList` de l'objet `MenuStrip` de la fenêtre conteneur (voir l'aide en ligne sur les fenêtres MDI).

`frmMDI` s'ouvre au démarrage de l'application et ouvre automatiquement la fenêtre `frmExo5` ; c'est là où tout se passe car chaque seconde :

- Le champ `chrono` est incrémenté de 1.
- Un nouveau `nombre` aléatoire est généré.

A l'ouverture du form `frmChrono` la `TextBox` « Chrono » est initialisé avec la valeur actuelle du chrono, elle ne bouge plus ensuite. Un clic sur le bouton Actualiser met à jour la valeur du chrono.
Il en est de même pour le form `frmRandom` et sa valeur du nombre aléatoire.

Avant tout codage, dessiner et bien nommer les différents objets (form, boutons…), ajuster les boutons par défaut de chacun des form… et commencer le code C# par les habituels cartouches de commentaires.

1. Le form conteneur : gestion des fenêtres

Pour commencer, vérifier que `frmMDI` est bien le form de démarrage (voir `Program.cs`). En plus des options du menu Fenêtre, prévoyez une option 'A propos de…' dans le menu d'aide (?) et une option Quitter dans le menu Fichier ; ainsi l'application sera conforme aux standards Windows.

La création et l'ouverture automatiques de la fenêtre frmExo5 s'obtiennent en plaçant le code suivant sur l'événement Load de la fenêtre frmMDI ou, mieux, dans le code de son *constructeur* (car cette portion de code est exécutée dès la création du form par l'instruction new) :

```
/// <summary>
/// constructeur du form MDI : ouverture de la fenêtre principale
/// </summary>
public frmMDI()
{
      InitializeComponent();
      frmExo5 frmPrinc ;  // déclare une instance du form principal
      frmPrinc = new frmExo5(); // instancie le form principal
      frmPrinc.MdiParent = this; // précise le conteneur MDI du form principal
      frmPrinc.Show(); // affiche le form (dans son form conteneur)
}
```

La création et l'ouverture des fenêtres Chrono et Nombre seront programmées dans la procédure événementielle « Click » de l'option correspondante du menu :

```
/// <summary>
/// menu Fenêtres / Chrono ==> instancier un form frmTime
/// </summary>
private void chronoToolStripMenuItem_Click(object sender, EventArgs e)
{
      frmChrono frmC ; // déclare une instance du form
      frmC = new frmChrono(); // instancie le form
      frmC.MdiParent = this;
      frmC.Show(); // affiche le form dans son conteneur
}
```

Vous savez déjà comment coder pour quitter l'application (`Applicatioin.Exit()`) ; pour fermer une fenêtre (boutons Fermer), utilisez tout simplement `this.close()`.

Tester le bon fonctionnement : la fenêtre principale doit s'ouvrir dès le démarrage, les fenêtres secondaires, à la demande et tout cela doit pouvoir se fermer à la demande (bien entendu les Textbox restent vides).

NB : Ici nous avons programmé une ouverture de fenêtre en « mode non modal », c'est-à-dire que l'opérateur peut accéder à chaque fenêtre selon son bon plaisir ; elles sont indépendantes les unes de autres (toutefois la fermeture de la fenêtre MDI entraîne la fermeture des fenêtres ouvertes contenues – essayez-donc !). Nous réaliserons plus tard des fenêtres « modales » ; dans une fenêtre en mode « modal » l'opérateur ne peut agir que sur la fenêtre en cours, il sera obligé de la fermer pour accéder aux autres fenêtres (principe des boîtes de dialogue).

2. Form principal : nombre aléatoire et chrono

Le nombre aléatoire est obtenu par la méthode `NextDouble()` d'un objet `System.Random` ; `NextDouble()` retourne un nombre décimal compris entre 0 et 1, en format `Double`. Nous le déclarerons une seule fois pour toute la classe.

```
public partial class frmExo5 : Form
{
      /// <summary>
      /// générateur de nombre aléatoire
      /// </summary>
      private System.Random aleat; // variable de niveau classe
      ///<summary>
      /// nombre aléatoire tiré
      /// </summary>
      private Double nombre ;
```

Et il sera instancié par le constructeur de la classe frmExo5 :

```
/// <summary>
/// Constructeur par défaut : génère un nombre aléatoire
/// et affiche les valeurs courantes (stockées dans Donnees)
/// </summary>
public frmExo5()
{
      InitializeComponent();
      this.aleat = new System.Random(); // instancie un objet générateur
aléatoire
      this.affiche();  // calcul nouvelles valeurs et affichages
}
```

Pour compter le temps, rien ne vaut un objet `Timer` (disponible en boite à outils) : sa propriété `.Enabled` permet de l'activer ou de le désactiver à tout moment ; un `Timer` déclenche l'événement `Tick` périodiquement, plus précisément tous les `.Interval` millisecondes (voir l'aide Visual Studio). Prévoir une variable de niveau classe, `chrono`, dans `frmExo5`, pour stocker cette valeur.

```
/// <summary>
/// Déclenchement timer : affiche les valeurs courantes
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void timer_Tick(object sender, EventArgs e)
{
      this.affiche(); // calcul nouvelles valeurs et affichages
}
```

Reste à définir la procédure d'affichage :

```
/// <summary>
/// calcul nouvelles valeurs
/// puis les affiche sur ce form
/// </summary>
private void affiche()
{
      // génère un nombre entre 0 et 1 et l'affiche en textbox
      this.nombre = this.aleat.NextDouble() ;
      this.txtNombre.Text = this.nombre.ToString() ;
      // incrémente chrono et l'affiche en textbox
      this.chrono ++ ;
      this.txtChrono.Text = this.chrono.ToString();
}
```

Nous avions bien entendu besoin d'une variable privée de niveau classe (ici, '`chrono`', de type `Int32`) initialisée à 0 au démarrage du form `frmExo5`.

Sauvegardez ; testez et rectifiez jusqu'à ce que cette partie de l'application soit opérationnelle, sans vous occuper de la fermeture des fenêtres ni des boutons Actualiser. Le form principal doit compter les secondes et tirer des nombres aléatoires. Ne faudrait-il pas protéger les TextBox contre la saisie directe par l'utilisateur ?...

*NB : par la suite, vous allez effectuer plusieurs améliorations successives de votre projet. Pour vous aider à vous y retrouver vous pouvez travailler sur des **copies** afin de conserver les différentes versions ; pour cela quand un projet est satisfaisant, **à la fin d'une amélioration**, il est préférable de **recopier sous un autre nom** , **avec l'Explorateur Windows, en dehors de Visual Studio,** l'ensemble des dossiers et sous-dossiers constituant le projet, puis de travailler normalement sur ce nouveau projet.*

*Rappel : en standard, avec Visual Studio 2013, pour chaque projet créé vous devez retrouver dans le dossier **documents\Visual Studio 2013\Projects** :*

- *un premier dossier portant le nom de la 'solution Visual Studio', par défaut du même nom que le projet, et contenant essentiellement un fichier d'extension **.sln***
- *un deuxième dossier portant le nom du 'projet Visual Studio' et contenant le fichier projet d'extension **.csproj**, tous les composants du projet (forms, Program.cs,…) ainsi que des sous-dossiers '**bin**', '**obj**' et '**Properties**' pour la compilation*

Il s'agit bien ici de recopier le dossier de la 'solution Visual Studio' et tous ses sous-dossiers. Le seul inconvénient à cette manipulation est que rien ne différencie les projets dans Visual Studio car ils portent tous le même nom ; attention de bien vous organiser !.

3. Gestion fine des fenêtres

Amélioration 1, à réaliser sur une copie de votre première 'solution Visual Studio' :
SI vous cliquez plusieurs fois sur l'option de menu, vous allez ouvrir plusieurs *instances* de la même fenêtre. Il faut donc faire un contrôle préalable que la fenêtre voulue n'a pas déjà été instanciée.
Pour cela il faut mémoriser que les fenêtres ont été ouvertes, nous allons donc déclarer les *références de fenêtre* comme membres de la feuille/classe `frmMDI` :

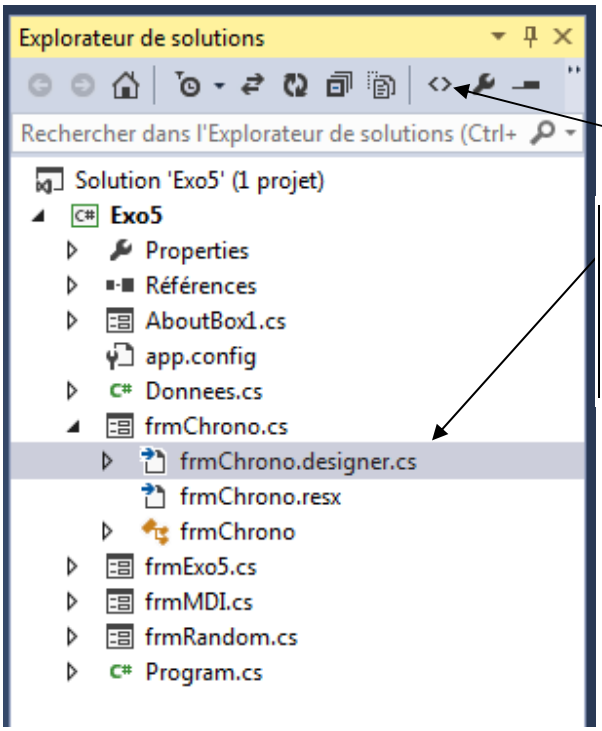
```
public partial class frmMDI : Form
{
      // Amélioration 1: 3 variables de niveau classe (accessibles dans toute
la classe)
      private frmExo5 frmPrinc ; // déclare une instance du form ppal
      private frmChrono frmC; // déclare une instance du form Chrono
      private frmRandom frmR;// déclare une instance du form Nombre
```

Puis nous modifions le code des fonctions événementielles permettant d'ouvrir les fenêtres :

```
/// <summary>
/// menu Fenêtres / Chrono ==> instancier un form frmChrono ou le réactiver
/// </summary>
private void chronoToolStripMenuItem_Click(object sender, EventArgs e)
{
      // instancie le form - amélioration 1
      if (this.frmC == null) // le form n'est pas/plus affiché
      {
            this.frmC = new frmChrono(); // instancie le form
            this.frmC.MdiParent = this;
            this.frmC.Show(); // affiche le form
      }
      else // le form secondaire est affiché
      {
            this.frmC.Activate(); // réactive le form (lui donne le focus)
      }
      // recopie la valeur courante
      this.frmC.txtChrono.Text = this.frmPrinc.chrono.ToString();
}
```

NB : même s'il n'est pas nécessaire dans notre cas, le mot-clé **this** ajouté en préfixe des noms de variables précise que cette variable est de *niveau classe*.

Il reste un petit problème de visibilité de la TextBox `txtChrono` : IntelliSense de Visual Studio ne vous l'a pas proposée lors de la frappe du code et il refuse de compiler. Pas grave !
Tous les contrôles graphiques que vous posez sur les forms grâce à la boite à outils de Visual Studio sont déclarés `private`, c'est à dire ***accessibles uniquement par ce form*** (cette classe). En mode Concepteur de Vues sur le form `frmChrono`, modifiez la propriété `Modifiers` de la valeur `private` vers la valeur `internal`, ou bien ouvrez la partie de code généré par le Concepteur de Vues de Visual Studio, pour le form `frmChrono`, et modifiez la portée de la TextBox, depuis `private` vers `internal` ; cette manipulation rend la TextBox ***accessible par tous dans le même projet*** (la même « assembly » - voir l'aide de Visual Studio).



Accès au code « partiel » du form généré par le Concepteur de Vues

NB : le mot-clé « `partial` » dans la définition des classes prend maintenant tout son sens : le code d'une classe peut être réparti entre plusieurs fichiers source ; Visual Studio en profite pour écrire le code généré par le Concepteur de Vues dans un fichier source séparé (et donc -un peu- protégé des erreurs du développeur).

```
Code généré par le Concepteur Windows Form
        // portée modifiée pour accès depuis les autres forms du projet
        internal System.Windows.Forms.TextBox txtChrono;

        private System.Windows.Forms.Label lblChrono;
        private System.Windows.Forms.Button btnActualiser;
        private System.Windows.Forms.Button btnFermer;
```

Sur le même principe, rendez la variable chrono du form `frmPrinc` accessible à tous dans le projet.

Faites de même pour ce qui concerne l'affichage de la fenêtre `frmRamdom`.

*NB : le form principal « connait » les forms secondaires grâce à leur « **référence** » ; **chaque référence d'objet est fournie lors de l'instanciation par l'instruction new et permet d'identifier chaque objet instancié sans ambigüité.** Ce mécanisme d'échanges de références entre objets qui doivent se « connaître » s'applique à tous les types de classes, en particulier, cela permet de représenter les relations entre objets 'Métier' d'une application. Ainsi, par exemple, pour représenter l'association entre des objets « GroupeDeStagiaires » et « Stagiaire », chaque objet « Stagiaire » instancié disposerait d'une référence à l'objet « GroupeDeStagiaires » correspondant et chaque objet « GroupeDeStagiaires » disposerait d'une collection des références des objets « Stagiaires » qui le concernent.*

Notez bien que ce mécanisme typiquement « orienté objet », utilisant les références des objets, est assez différent du mécanisme des « clés étrangères » en vigueur pour représenter les relations entre enregistrements dans les bases de données.

Sauvegardez ; testez et rectifiez jusqu'à ce que cette partie de l'application soit opérationnelle, sans vous occuper de la fermeture des fenêtres ni des boutons Actualiser. *Les fenêtres Chrono et Random ne sont maintenant plus dupliquées à chaque appel depuis le menu Fenêtre.*

4. Actualisation des fenêtres secondaires

Amélioration 2 ; *pensez à travailler sur une copie de votre réalisation précédente !*

Pour que les fenêtres Chrono et Random puissent réactualiser leurs valeurs, il est nécessaire qu'elles « connaissent » le formulaire qui les a appelées ; certains langages permettent de déclarer des variables « globales », accessibles par tous dans le même projet. Mais ce n'est pas le cas du langage C#, résolument rigoureux et « orienté objet ». Nous allons donc effectuer quelques adaptations de notre code de manière à ce que les différents forms (qui sont des *objets instanciés* en mémoire) se connaissent par communication de leur « *référence* ».

Commençons par les boutons Actualiser.
Nous avons défini des variables « `nombre` » et « `chrono` » comme membres publiques/internes de la classe « frmExo5 » :

```
public partial class frmExo5 : Form
{
      // 3 variables de niveau classe ==> accessibles à tous dans la classe
      // 2 public ==> accessibles aussi depuis d'autres forms

      /// <summary>
      /// générateur de nombre aléatoire
      /// </summary>
      private System.Random aleat; // variable de niveau classe

      ///<summary>
      /// nombre aléatoire tiré
      /// </summary>
      public Double nombre ;

      ///<summary>
      /// valeur du chrono
      /// </summary>
      public Int32 chrono ;
```

Pour pouvoir atteindre ces membres depuis les feuilles frmChrono et frmRandom, il faut qu'elles connaissent la **référence de la fenêtre principale frmExo5** (qui est elle-même un objet distinct). Actuellement, seule la fenêtre MDI `frmMDI` connaît la référence à ce form.

Un principe de la manipulation des objets est **qu'un objet connaît toujours sa propre référence** ; en langage C#, le mot-clé `this` représente la référence à l'objet courant (nous l'avons déjà utilisé pour nommer les contrôles ou variables d'un form). En conséquence un objet peut transmettre sa référence (ou une référence connue de sa référence) aux autres objets au moment de leur instanciation :

```
/// <summary>
/// menu Fenêtres / Chrono ==> instancier un form frmChrono
/// </summary>
private void chronoToolStripMenuItem_Click(object sender, EventArgs e)
        {
            // instancie le form - amélioration 1 et 2
            if (this.frmC == null) { this.frmC = new frmChrono(this.frmPrinc);
… }
```

Reste maintenant à récupérer cette référence d'objet dans les autres forms.
La manœuvre est plus délicate, voici une (bonne) solution :

Déclarer dans la classe frmChrono un membre privé pour stocker cette référence :

```
public partial class frmChrono : Form
{
    private frmExo5 frmPrinc; // amélioration 2 - référence au form appelant
```

Modifier ensuite le « constructeur » de la classe frmChrono généré par Visual Studio :

```
public partial class frmChrono : Form
{
    private frmExo5 frmPrinc; // amélioration 2 - référence au form appelant
    public frmChrono(frmExo5 f) // amélioration 2 - constructeur paramétré
    {
        InitializeComponent();
        this. frmPrinc = f; // amélioration 2 - mémorise la ref au form appelant
```

Comme vous le constatez nous avons ajouté un paramètre au *constructeur* (`frmChrono(…)`. Ce paramètre est une variable *locale à la procédure constructeur* (elle est détruite dès la fin de l'exécution du constructeur) et nous avons dû la recopier dans une *variable de niveau classe* pour que la procédure liée au bouton Actualiser puisse l'utiliser ultérieurement.

Ecrivez enfin cette procédure d'actualisation :

```
        /// <summary>
        /// bouton Actualiser : recopier chrono depuis form appelant
        /// </summary>
        private void btnActualiser_Click(object sender, EventArgs e)
        {
            // recopie depuis le form appelant (le chemin est long…)
            this.txtChrono.Text = this.frmPrinc.chrono.ToString();
        }
```

Testez. Reproduisez toutes ces adaptations sur le form frmRamdom

Pour terminer, il serait bon de libérer la référence à la fenêtre après usage (ce qui évitera aussi des « plantages » quand on tente de ré-ouvrir une fenêtre fermée – essayez-donc !) ; pour cela il suffit de créer des procédures dans `frmMDI` qui affectent la valeur **`null`** aux références de fenêtres :

```
        /// <summary>
        /// libère la ref au frmChrono
        /// </summary>
        public void fermeChrono()
        {
            this.frmC = null; // libère la réf au from chrono
        }
```

Il restera à appeler cette procédure de la fenêtre père lors de la fermeture de la fenêtre détail ; voici un exemple pour le form `frmChrono` :

```
        /// <summary>
        /// bouton Fermer : libère la réf au from courant et le ferme
        /// </summary>
        private void btnFermer_Click(object sender, EventArgs e)
        {
            ((frmMDI)(this.frmPrinc.MdiParent)).fermeChrono(); // libère la
référence à ce form - nécessite de caster MdiParent
            this.Close(); // ferme ce form
        }
```

La syntaxe devient compliquée (mais on fera mieux par la suite) : `frmChrono` connaît une référence de `frmExo5` qui connait une référence de `frmMDI` grâce à sa propriété `MdiParent` mais elle est « vue » comme une variable de type `Form` ; il faut donc la « caster » comme un `frmMDI` de manière à y voir sa procédure `fermeChrono()`.
Faites de même pour le form `frmRandom`.

Sauvegardez, compilez, testez en ouvrant et fermant plusieurs fois les fenêtres ; actualisez les fenêtres détail ; tout devrait fonctionner…

Encore une piste d'amélioration :

NB : on se limite ici à poser le problème sans réaliser la solution ; on mettra cela en œuvre par la suite dans l'amélioration 3.

Notre application tourne correctement et conformément aux spécifications fonctionnelles énoncées en début de ce cahier. Il reste une petite imperfection : quand l'utilisateur ferme un form `frmChrono` ou `frmRandom` à l'aide de la case de fermeture, la fenêtre se ferme bien (c'est un mécanisme hérité de le classe `Form`) mais le « déréférencement » dans le form principal `frmMDI` n'est pas réalisé (puisque la méthode `fermexxx()` est appelée uniquement par la procédure associée au bouton Fermer. Pour résoudre ce problème, il nous faut analyser le fonctionnement événementiel des objets `Form` (voir l'aide) ; on constate qu'un `Form` qui se ferme déclenche les événements *`FormClosing`* puis *`FormClosed`*. Il nous suffit donc de déclencher l'appel de notre procédure de déréférencement non plus lors du clic sur le bouton Fermer mais lors de la fermeture du form (déclenchée aussi bien par le bouton Fermer - méthode `Close()` - que par la case de fermeture). Par conséquent la procédure événementielle associée au bouton Fermer se limiterait à fermer le form courant :

```
/// <summary>
/// méthode déclenchée par le bouton fermer ==> ferme ce form
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnFermer_Click(object sender, EventArgs e)
{
   this.Close(); // ferme ce form ==> déclenche FromClosing
}
```

Il resterait alors à ajouter la procédure événementielle associée à l'événement **`FormClosing`** ; le code demanderait l'exécution de la procédure correspondante contenue dans le form MDI `frmMDI` :

```
/// <summary>
/// méthode appelée lors de la fermeture du form ==> déréférencer la fenêtre chrono
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void frmChrono_FormClosing(object sender, FormClosingEventArgs e)
{
            ((frmMDI)(this.frmPrinc.MdiParent)).fermeChrono(); // libère la
référence à ce form - nécessite de caster MdiParent
}
```

Avec cette dernière amélioration, notre application donnerait satisfaction et elle commencerait à être conçue et programmée en « orienté objet ».
Mais les puristes du monde objet reprocheront à notre construction la *trop forte spécialisation* des form `frmChrono` et `frmRandom` ; en effet, ces forms *doivent être appelés par un form de type* `frmMDI` qui *doit contenir des membres bien précis* tels la méthode `fermeChrono()` ou la propriété `nombre`, ce qui *limite les possibilités de réutilisation* de nos forms dans d'autres applications (mais ce n'est pas notre soucis principal aujourd'hui). Et l'orientation objet a pour but de faciliter la réutilisation des composants développés.

Comment faire alors ? *L'idéal sera de créer, par le form appelant* `frmMDI` *lors de la création d'un form secondaire, une procédure événementielle, implémentée dans le form secondaire, mais qui déclenche une action écrite dans le form principal*. Ainsi, le form secondaire serait totalement indépendant du form principal et donc réutilisable dans un autre contexte. Nous allons maintenant mettre en œuvre cette dernière construction.

5. Vers une meilleure conception objet : rendre les form plus indépendants et réutilisables

Amélioration 3 (à réaliser sur une nouvelle copie !)

Notre application comporte plusieurs fenêtres en interactions les unes avec les autres ; c'est bien le principe d'une application « Windows ». Chaque fenêtre étant programmée dans une classe et les classes étant indépendantes les unes des autres, nous avons dû *communiquer des informations entre classes*. Le moment opportun est lors de l'instanciation d'une fenêtre secondaire (*passation de paramètre au constructeur* de la classe). Le paramètre passé est la *référence de l'instance de classe* correspondant à la fenêtre principale.

Notez que la fenêtre MDI qui gère les fenêtres de l'application connaît les références de ces fenêtres mais ne sait rien des données à y afficher ; de même la fenêtre principale mémorise et expose les données à afficher puisque c'est elle qui les gère. On a déjà un début de partage des rôles entre les classes de l'application.

Cette construction opérationnelle et bien faite présente plusieurs limites :

- Transmettre la référence à un objet **donne accès à tous les membres exposés par cet objet** ce qui peut être potentiellement « dangereux » (comme quand une personne communique l'adresse de son domicile et la clé de la porte d'entrée) ;
- Avec un langage fortement typé comme C#, recevoir **la référence à un objet nécessite de connaître le type de cet objet**, ce qui rend les fenêtres spécifiques à l'application en ce sens qu'elles sont conçues pour être utilisées dans un contexte bien particulier ; cela limite la réutilisabilité des composants écrits (ici, un form de type `frmRandom` *doit* être appelé par un form de type `frmMDI`).

On peut construire cette application de manière différente, encore mieux « orientée objet », au prix de quelques adaptations selon les principes suivants :

- *Pas de passation de paramètres aux constructeurs* des fenêtres secondaires ; la fenêtre '*mère' connaît* ses fenêtres '*filles'* car elle les instancie, mais les fenêtres '*filles'* ne '*savent rien'* de leur '*génitrice'* ;
- Puisque les différentes classes doivent se partager certaines données, on va les exposer (en `public static`) dans une *classe particulière* conçue à cet effet, sorte de « dépôt de données » (ainsi la fenêtre principale pourra y stocker les valeurs courantes des variables `nombre` et `chrono` et les fenêtres secondaires n'auront plus qu'à y piocher ce qui les concerne) ;
- Puisque un événement d'une fenêtre secondaire doit provoquer un traitement dans une fenêtre principale (bouton `Fermer` qui « libère » la référence à la fenêtre secondaire), *la fenêtre principale va implémenter, lors de son instanciation, un événement dans la fenêtre secondaire, pour réaliser elle-même le traitement en réponse à l'événement survenu dans la fenêtre 'fille' (qui ne 'sait rien' de sa 'mère').*

Avec cette construction, les différentes classes resteront *indépendantes* les unes des autres, *réutilisables* et *évolutives* indépendamment les unes des autres ; elles devront simplement connaître les noms et types des *données communes*, stockées et partagées par une autre classe (le dépôt de données). Bien entendu, tout ceci ne concerne que **l'organisation** de la programmation des traitements et n'a aucune incidence sur l'interface homme-machine, sur le design graphique et le fonctionnement de l'application. Allons-y !

Pensez à effectuer tout d'abord une copie de la solution sous un autre nom !

Ajoutez au projet une classe contenant des données déclarées en public ou internal et static, c'est-à-dire partagées et disponibles pour tous dans le projet sans nécessiter d'instanciation :

```
namespace Exo5
{
    /// <summary>
    /// classe de données statiques partagées par les form de l'application Exo5
    /// </summary>
    public class Donnees
    {
        // nom de variables avec une capitale
        // par convention, car la portée est publique ou interne au projet
        internal static Int32 LeTemps = 0;
        internal static Double LeNombre = 0;
    }
}
```

Dans frmExo5, adapter la méthode affiche() de manière à ce qu'elle ***affecte*** ces données partagées, et supprimer l'exposition par la classe des variables nombre et chrono réalisée précédemment (elles redeviennent private) :

```
        /// <summary>
        /// calcul nouvelles valeurs et les stocke dans Donnees
        /// puis les affiche sur ce form
        /// </summary>
        private void affiche()
        {
            // génère un nombre entre 0 et 1 et l'affiche en textbox
            Donnees.LeNombre = this.aleat.NextDouble(); // exporte en dépôt de
données
            this.txtNombre.Text = Donnees.LeNombre.ToString();
            // incrémente chrono et l'affiche en textbox
            Donnees.LeTemps ++; // exporte en dépôt de données
            this.txtChrono.Text = Donnees.LeTemps.ToString();
        }
```

Modifier l'instanciation des forms secondaires en supprimant la passation de paramètres aux constructeurs de classes ; par exemple, pour le form frmChrono :

```
        /// <summary>
        /// menu Fenêtres / Chrono ==> recopier les valeurs
        /// en nouvelle fenêtre ou fenêtre en cours si déjà ouverte
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void chronoToolStripMenuItem_Click(object sender, EventArgs e)
        {
 if (this.frmC == null) // le form secondaire n'est pas affiché ==> l'instancier
            {
                this.frmC = new frmChrono(); // instancie le form
```

Dès l'instanciation du form secondaire, implémenter *un événement sur le form secondaire* permettant de déclencher *une méthode du form MDI* ; par exemple pour la fermeture du from frmChrono :

```
        private void chronoToolStripMenuItem_Click(object sender, EventArgs e)
```

```
        {
    if (this.frmC == null) // le form secondaire n'est pas affiché ==> l'instancier
            {
                this.frmC = new frmChrono(); // instancie le form
                // implémente un événement sur le nouveau form
                // pour libérer la ref à ce form
              this.frmC.FormClosing += new FormClosingEventHandler(this.fermeChrono);
```

La syntaxe peut sembler bizarre ; retenez qu'un événement intercepté au moment de l'exécution peut fort bien déclencher *plusieurs* procédures événementielles (d'où l'écriture **+=**) et que *ces procédures événementielles ne sont pas forcément implémentées dans la classe qui déclenche l'événement* (c'est là la puissance de ces *EventHandler* !) ; ici, nous avons littéralement écrit : « ***ton** événement FormClosing déclenchera fermChrono() **chez moi*** ».
Pour que tout cela fonctionne correctement, il reste à *transformer* la méthode **`fermeChrono()`** *en procédure événementielle* : ajouter simplement les 2 paramètres nécessaires à toute procédure événementielle, le premier de type `Object`, et pourquoi pas de nom `sender` comme le fait Visual Studio, et le second d'un type `EventArgs` ou d'un de ses dérivés, et pourquoi pas de type `FormClosingEventArgs` et de nom e comme le fait si bien Visual Studio.
Procédez de même pour le form `frmRandom`.
Voilà l'essentiel pour la classe `frmExo5.`

Dans les classes `frmChrono` et `frmRandom`, adapter en conséquence les constructeurs (plus besoin de passation desparamètres) ; par exemple, pour `frmChrono` :

```
        /// <summary>
        /// constructeur par défaut
        /// </summary>
        public frmChrono()
        {
            InitializeComponent();
            this.affiche(); // actualise la valeur affichée
        }
```

Remettre tous les contrôles graphiques des forms secondaires en `private` ; ils n'ont plus besoin d'être exposés car les form secondaires peuvent aller chercher eux-mêmes leurs données dans le classe commune `Donnees`.

Affiner le comportement des forms secondaires en utilisant au mieux les méthodes événementielles des Winfoms ; par exemple pour `frmChrono` :

```
        /// <summary>
        /// méthode déclenchée par le bouton actualiser
        // /==> recopie la valeur courante du chrono
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void btnActualiser_Click(object sender, EventArgs e)
        {
            this.affiche(); // actualise la valeur affichée
        }
```

Et aussi quand ce form reprend le focus :

```
        /// <summary>
        /// activation du form ==> actualiser les données
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
```

```
        private void frmChrono_Activated(object sender, EventArgs e)
        {
            this.affiche(); // actualise la valeur affichée
        }
```

Etablir la méthode d'affichage des forms secondaires de manière à piocher les données dans la classe prévue à cet effet ; par exemple pour `frmChrono` :

```
        /// <summary>
        /// recopie le chrono depuis la classe de dépôt de données statiques
        /// </summary>
        private void affiche()
        {
            this.txtChrono.Text = Donnees.LeTemps.ToString(); // beaucoup lus
simple !
        }
```

Tester et mettre au point…

Au final :

- les boutons `Fermer` des forms secondaires déclenchent simplement la fermeture des form ;
- cette fermeture déclenche une méthode dans le form MDI qui lui permet de « libérer » la référence au form secondaire :
- les boutons `Actualiser` des forms secondaires se contentent *d'aller piocher des données partagées* par une autre classe prévue à cet effet ;
- le form principal *gère les données partagées* et les recopie dans la classe prévue à cet effet ;
- le form MDI *gère lui-même complètement* les affichages des forms secondaires (mémorisation des références aux fenêtres, instanciations ou activations) ; ses références aux fenêtres secondaires `frmC` et `frmR` restent *privées* ;
- les form secondaires sont indépendants de leur « géniteur » ; ils n'en connaissent ni le type ni la référence.

> *En bref, les classes s'exposent moins, sont moins dépendantes les unes des autres, plus réutilisables et évolutives indépendamment les unes des autres, et ce, au prix d'une classe supplémentaire, c'est-à-dire au prix d'une augmentation de la complexité. Cette démarche de conception est à la base d'un « catalogue de bonnes pratiques » connu sous le nom de « design patterns ».*

6. Cerise sur le gâteau…

Allez pour finir, *Ajoutez* au projet un *Nouvel élément* de type *Boite de dialogue A propos de*. Codez ce qu'il faut pour instancier et afficher cette fenêtre comme une boite de dialogue, en *mode Modal*, grâce à `.ShowDialog()` au lieu de `.Show()`, compilez, admirez le résultat (et remerciez Microsoft de vous avoir mâché le travail !).

Synthèse sur les forms et la portée des variables en C# .Net

Variable locale

- Une variable déclarée avec son type à l'intérieur d'une procédure ou d'une fonction n'est accessible qu'à l'intérieur de cette procédure/fonction, on dit que c'est une *variable locale*.
- Une variable locale est créée à l'entrée dans la procédure/fonction, détruite à la sortie.
- En principe dans C# les variables locales sont initialisées par défaut (à zéro, à null, à ''…), mais il est de bonne pratique de les initialiser explicitement dans le code, au moment opportun.

Variable de feuille/form/classe

- Une variable définie au début du code d'une feuille est visible de toutes les procédures et fonctions de la feuille : c'est *une variable de niveau classe* (sa *portée* est la classe entière).
- Le constructeur de classe (ou l'événement *load* éventuellement) est l'opportunité d'initialiser les variables de niveau feuille/classe.
- Ne définir au niveau feuille/classe que les informations qui doivent persister tant que la feuille est chargée en mémoire.
- Les variables de feuilles/classe peuvent être définies ***private***, dans ce cas elles ne sont pas visibles depuis d'autres feuilles de l'application ; elles peuvent être déclarées ***public***, dans ce cas elles sont visibles depuis toute autre classe qui y *fait référence* ; elles peuvent encore être exposées ***internal*** pour limiter leur accès uniquement aux autres composantes du projet (de *l'assembly .Net*).
- Une variable de portée feuille/classe peut être déclarée partagée (***static***) : elle sera alors *unique et partagée par toutes les instances* de cette classe. De plus, cette variable partagée « existe » et ne nécessite pas que la classe soit instanciée pour pouvoir y accéder.
- Les contrôles graphiques sont déclarés par défaut en *private* (par le Concepteur de Vues de Visual Studio) et ne sont donc pas accessibles aux autres classes du projet ; il est parfois nécessaire de modifier leur portée en *public* ou *internal* en reprenant le code généré dans le fichier source partiel nommé xxx.Designer.cs ou en adaptant la propriété *Modifiers* dans le Concepteur de Vues.

Feuilles/form

- Les objets feuilles sont potentiellement visibles dans toute l'assembly (le projet), lorsqu'ils sont chargés/instanciées. Les variables de niveau feuille/classe et les contrôles graphiques (et leurs propriétés) sont donc visibles selon leur exposition ; pour y avoir accès, il faut avoir une *référence* sur l'objet concerné.
- Un form incomplet peut être complété après son instanciation, en particulier il est fréquent de lui ajouter la détection d'événements qui doivent déclencher des procédures événementielles stockées dans d'autres forms

Conseils

- Sauf nécessité contraire, il faut définir les variables *le plus locale possible*. Limiter au maximum les variables de niveau classe et portée public.
- Bien commenter les variables, soigner leur portée et leur exposition.
- Bien documenter les variables de niveau classe, quelque soit leur portée

Synthèse sur les constructeurs et la passation de paramètres entre classes

Une *classe* reste quelque chose de très *abstrait*, une sorte de « moule » à fabriquer des objets similaires, du moins de même nature.

Chaque objet « prend vie » en mémoire lors de son *instanciation* par l'instruction *new*.

Dans C# .Net, tout est *objet*, form, contrôles graphiques, palette de couleur… Certains objets sont *instanciés automatiquement* lors de l'exécution : le form de démarrage du projet, les contrôles graphiques déposés sur le form… D'autres objets sont *instanciés explicitement par programmation* à l'aide de *l'instruction new* comme dans :

```
frmChrono frmC = new frmChrono() ;
```

qui contracte les 2 instructions :

```
frmChrono frmC ;
frmC = new frmChrono() ;
```

L'instanciation d'un objet permet de récupérer sa *référence*, utile pour y accéder ultérieurement.

Dans tous les cas, l'instanciation d'un objet déclenche l'exécution d'une procédure/méthode particulière appelée « *constructeur* »

```
public frmChrono()
        {
            InitializeComponent();
        }
```

que doit contenir toute classe C# .Net.

Une manière « orientée objet » de faire communiquer les objets entre eux est d'effectuer des *passages de paramètres entre objets lors de leur instanciation.*

Tous les paramètres ainsi transmis restent considérés comme des *informations locales pour le constructeur de l'objet instancié* (tout comme les paramètres passés à un sous-programme, comme une procédure événementielle ou votre sous-fonction de recherche de caractères, restent des variables locales visibles de ce seul sous-programme).
Il est donc souvent nécessaire de recopier ces paramètres en *variables de niveau classe* pour pouvoir les mémoriser durablement et y accéder depuis d'autres procédures ou fonctions de cette même classe.

On peut aussi bien passer en paramètre des *valeurs* de variables (cas de la sous-fonction de recherche) que des *références* d'objets (ce que vous avez fait dans les améliorations de cet exercice).

Le passage de la référence d'un objet permet au programme de l'objet instancié d'accéder facilement à toutes les propriétés et toutes les méthodes *exposées* de l'objet appelant mais cette technique nécessite que l'objet appelé connaisse précisément le type de l'objet appelant. Pour rendre les classes plus *indépendantes* les unes des autres tout en conservant un partage d'informations communes, on peut ajouter au projet une classe exposant les données à partager.

La technique événementielle des Winfoms, basé sur un *handler d'événement associé à une procédure/méthode événementielle*, permet au besoin qu'un événement détecté par un objet déclenche une méthode *écrite dans un autre objet*.