
# Explorer

## 206 Explorer Application Winform et CRUD

Rédigé par

**David ROUMANET**
Professeur BTS SIO

Changement

| Date | Révision |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |

# Sommaire

# A Objectifs

Cette activité doit permettre de découvrir les notions importantes pour l'année prochaine :

1. Gestion des classes en C#
2. Gestion d'un CRUD
3. Création d'un client lourd graphique en C#
4. Utilisation d'UML pour représenter le fonctionnement d'une application

Ces quatre points seront réutilisés dans des activités plus complexes en B2-DEV3 et B2-DEV4.

# B Scénario

Pour l'activité chasse au trésor, les scores ont été inscrits dans une base de données (accessible publiquement).

L'objectif du programme sera de gérer les scores : en effet, il est possible que certains accès parasitent la base de données, ou bien de devoir remettre à zéro les données, ou encore de changer le nom d'un utilisateur.

Ce programme devra donc permettre la création (Create) l'affichage (Read), la modification (Update), l'effacement (Delete) de données. On appelle CRUD, l'ensemble des opérations qui s'appliquent sur un même ensemble de données.

# C Conception

Contrairement à l'idée reçue, la conception d'une application commence sur papier. Il faut représenter l'application dans ses différents états et comportements.

Nous allons utiliser deux types de diagrammes UML :

- Diagramme d'utilisation
- Diagramme de classe

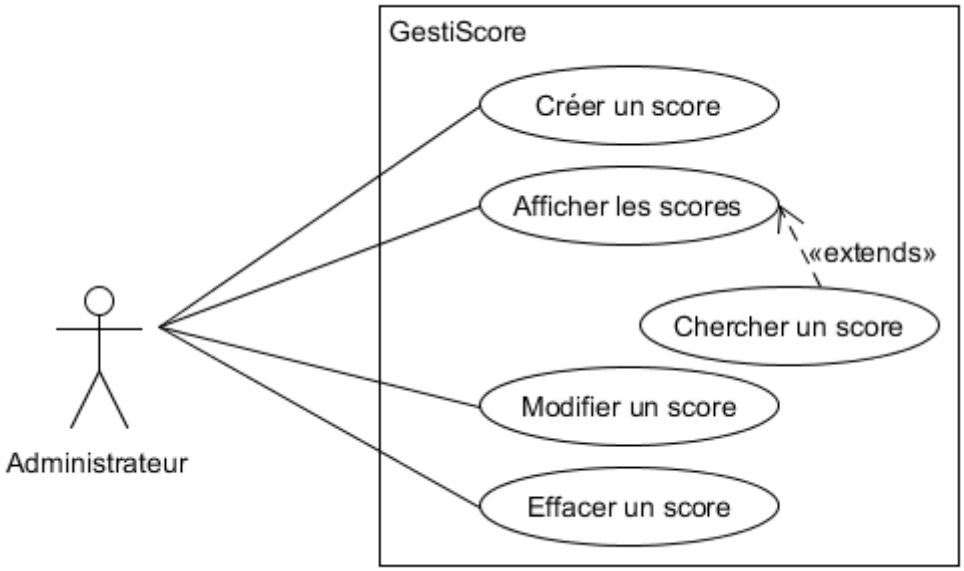
Enfin, il faut représenter l'affichage graphique que l'utilisateur aura en face de lui (UX, User eXperience). La plupart du temps, il s'agit d'une maquette, c'est-à-dire un ensemble de dessins représentant l'interface du programme.

## C.1 Diagramme d'utilisation

Le diagramme d'utilisation représente un système (ici, l'application de gestion des scores), les utilisateurs – ou plutôt les différents rôles des utilisateurs – et les interactions possibles, qui sont appelées les cas d'utilisation.

Pour notre application, il n'y aura qu'un utilisateur qui aura le rôle administrateur. Il pourra effectuer toutes les actions de notre CRUD, ainsi il peut créer des nouveaux scores, afficher un score particulier ou tous les scores, mettre à jour un score et supprimer un score.

Voici la représentation sous UML (logiciel : UMLet ► https://www.umlet.com/) :




Ce simple graphique permet à un développeur de comprendre qu'il faudra développer 5 éléments : des vues, des fenêtres, des méthodes et des accès pour ces 5 fonctionnalités. Ce sont les "stories" que l'on trouve en développement Agile.

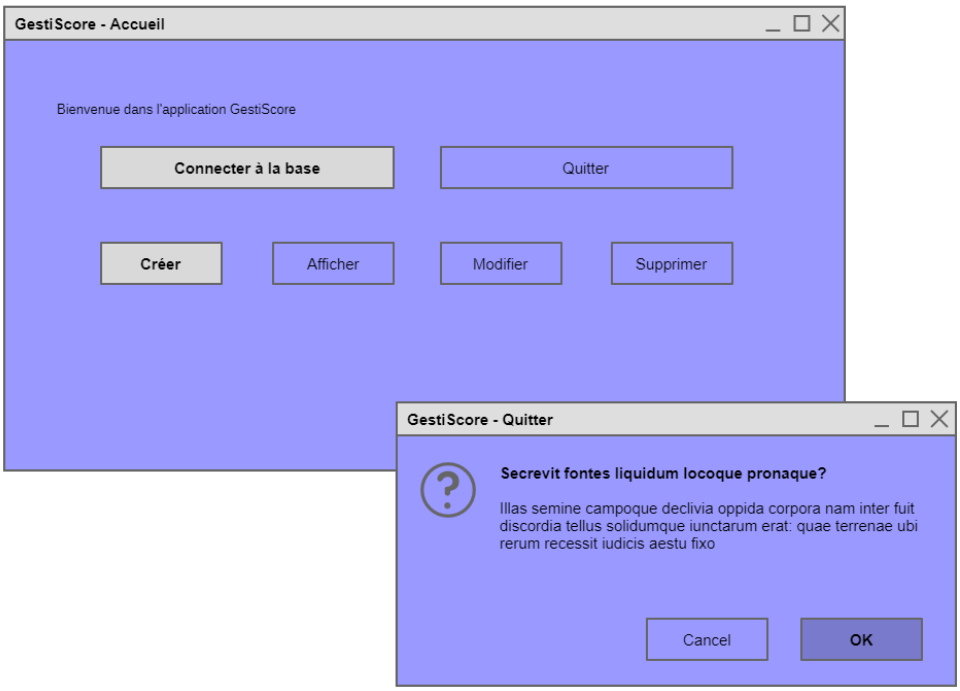
## C.2 Maquettes

Les maquettes sont de simples représentations graphiques du code : dans cette étape, vous ne devez surtout pas taper une seule ligne de code.

Il s'agit de montrer au client à quoi ressemblerait votre application : c'est lui qui décide si cela lui convient. Donc, pour ne pas perdre de temps en création de code, on utilise des outils de mookup pour dessiner les différentes fenêtres. Ici, j'utilise Pencil (https://pencil.evolus.vn/).

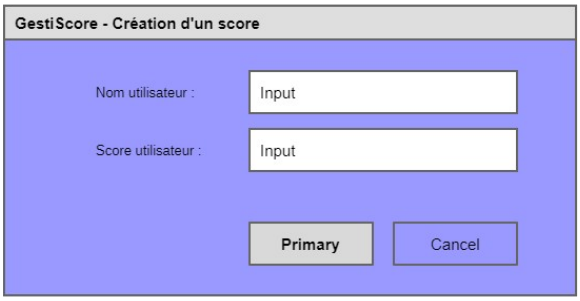
### C.2.1 Écran d'accueil

Lorsque l'application est lancée, c'est le premier écran qui sera visible.



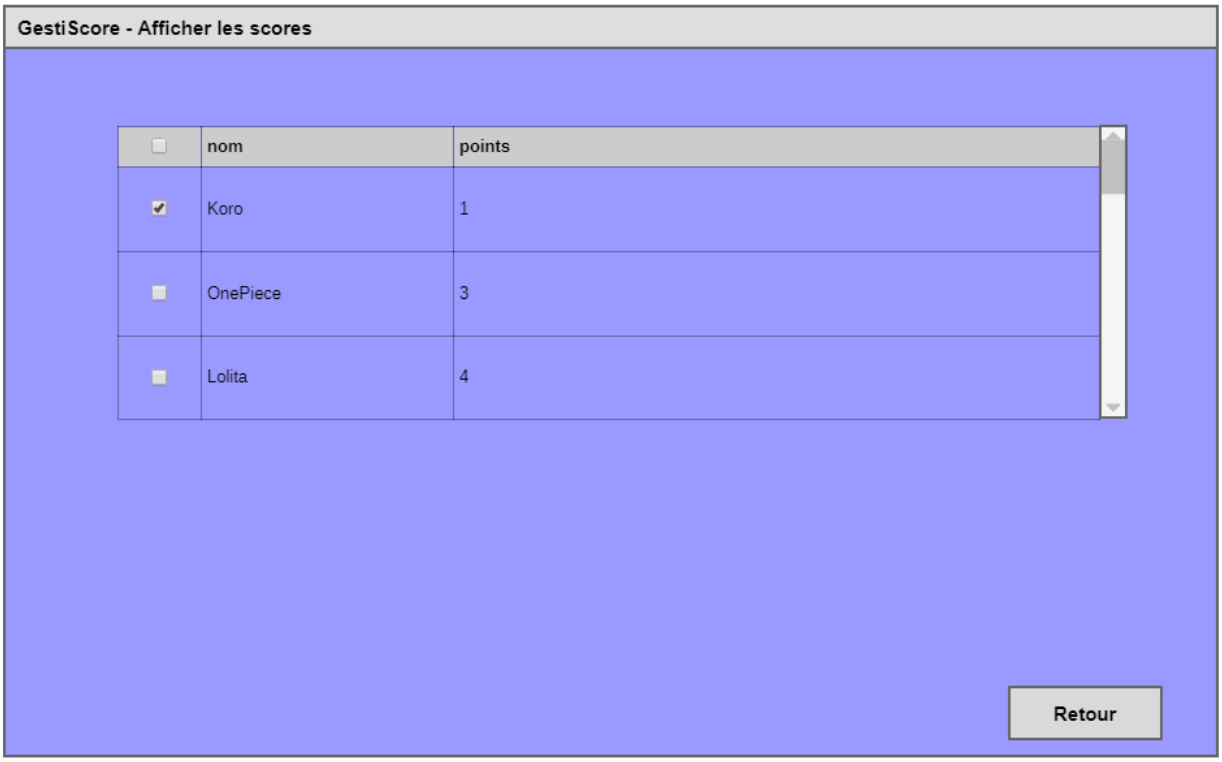

### C.2.2 Écran de création

Lorsque l'administrateur veut créer un nouveau score.



### C.2.3 Écran d'affichage

Dans cet écran, l'administrateur peut afficher l'ensemble des scores de la base.




Comme Rechercher un score est une extension de l'écran d'affichage, il est envisageable de proposer un champ de saisie et un bouton qui lancera la recherche ou bien filtrera la recherche dans l'affichage existant.

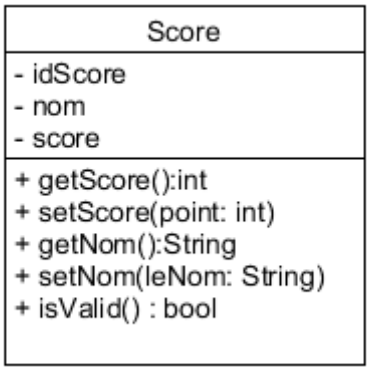
Je ne vais pas représenter toutes les fenêtres, car nous verrons d'autres moyens pour obtenir certains résultats.

## C.3 Diagramme de classe

Le diagramme de classe UML est relativement simple pour les attributs, mais il faut prévoir les différentes méthodes :

- Il y a les méthodes liées aux fenêtres graphiques
- Il y a les méthodes liées aux requêtes dans la base de données (envoi de requêtes et récupération des données).

Les données des requêtes ne sont pas incluses dans la classe Score.




En effet, le programme travaillera sur des informations de type Score, mais la classe Score n'a pas à connaître le moyen de stocker les données dans la base MySQL. La classe est indépendante, et les requêtes peuvent changer pour une base NoSQL ou PostgreSQL, etc.

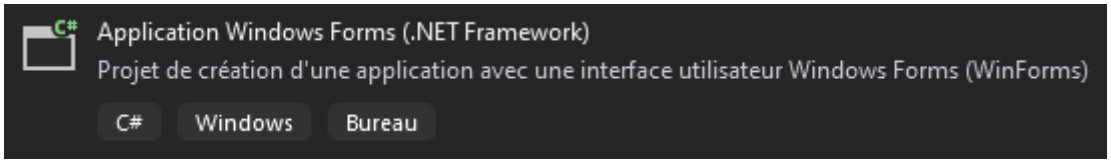
On parle souvent de modèle de données.

# D Développement

Avec les informations données, nous pouvons commencer un projet qui sera un peu long dans sa création.

## D.1 Démarrage

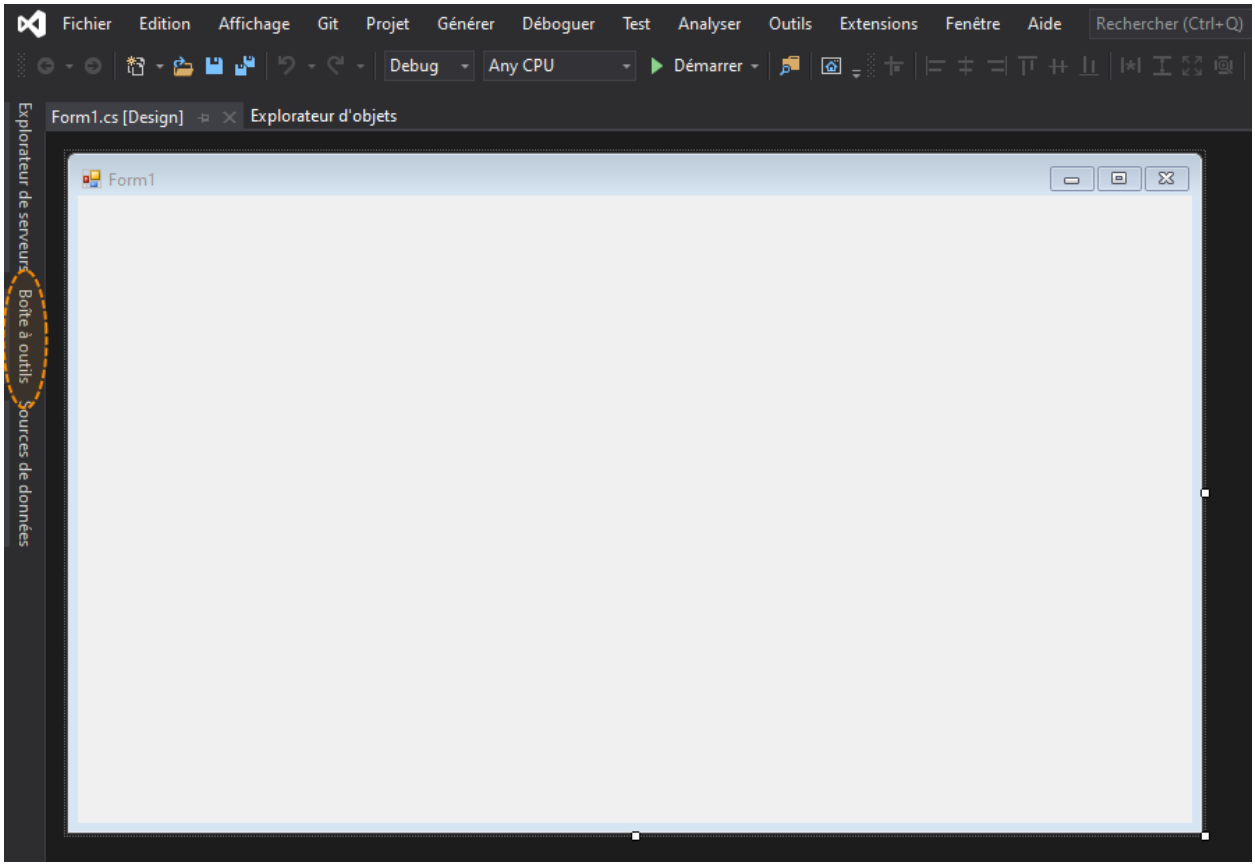
Ouvrir Visual Studio 2019, et créer un nouveau projet de type Windows forms (.NET Framework)




Proposez le nom **GestiScore**.

## D.2 Création de l'interface d'accueil

L'interface ouverte ne semble pas proposer d'outils pour placer des ressources dans la fenêtre, mais il suffit d'afficher la boite à outil sur la gauche (éventuellement de la "punaiser" pour qu'elle reste présente en permanence).



Une fois la boite à outils prête, il suffit de faire du glisser/déposer d'éléments, comme les boutons pour l'écran d'accueil.

Surtout, ne double-cliquez pas sur un bouton pour modifier ses propriétés : C# interprète cette action comme la création d'une méthode associée au bouton (vous ne gérez pas le nom de la méthode).

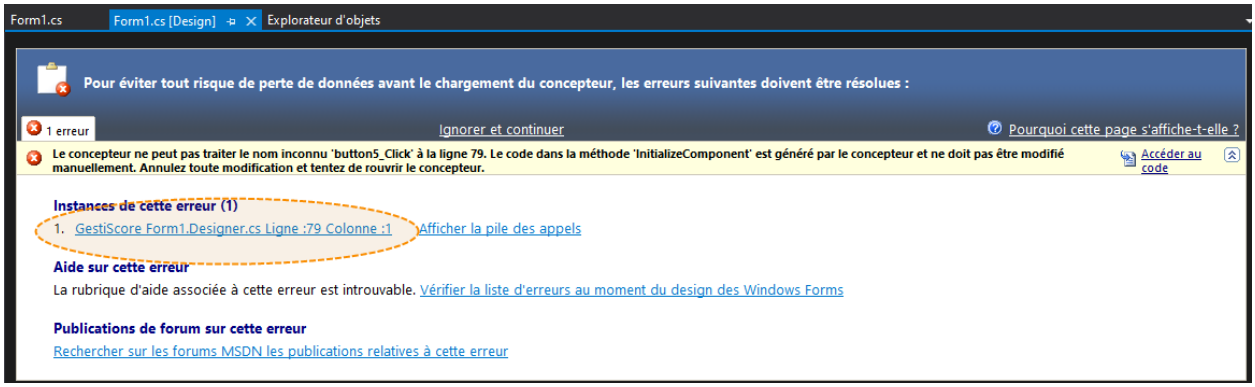
Il faut préférer éditer les propriétés dans l'espace à droite, sinon vous devrez effacer les traces de votre malencontreuse erreur comme suit. Pour reproduire le problème, double-cliquez sur un bouton dans la forme.

### D.2.1 Suppression d'une méthode sur double-clic (par erreur)

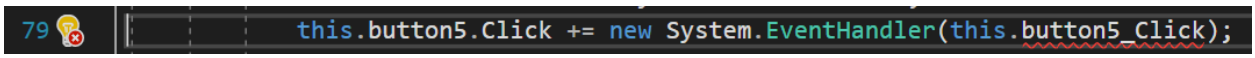
dans form1.cs, supprimer la méthode buttonX_click

```
        private void button5_Click(object sender, EventArgs e) {

        }
```

Sauvegardez, puis dans Form1.cs [Design] choisissez l'accès à la ligne en erreur :



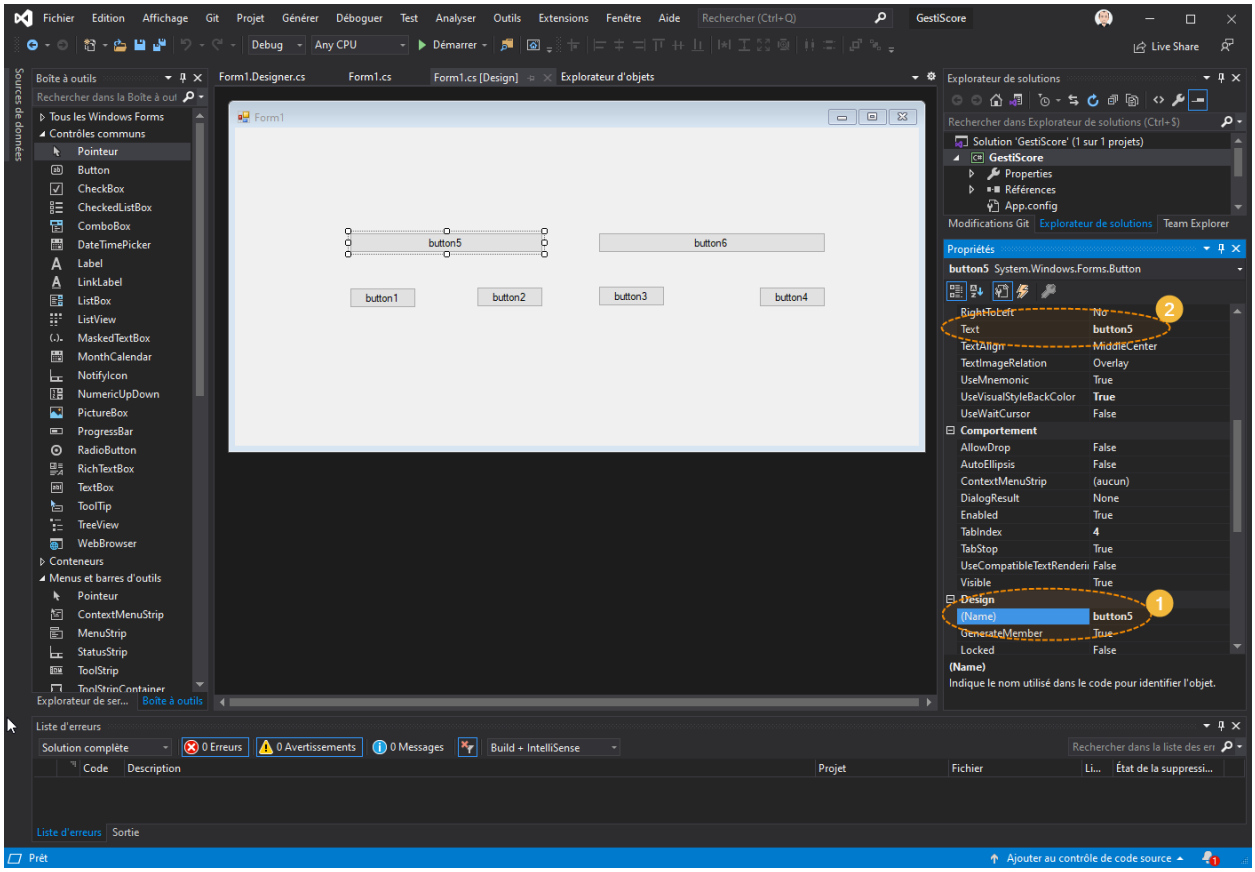

et supprimez la ligne en erreur (celle où C# ajoute le bouton au système de surveillance des événements.




Sauvegardez le fichier : normalement votre interface graphique fonctionne à nouveau.

### D.2.2 Modification des propriétés d'un élément graphique

Dans la barre à droite, se trouve les propriétés de l'élément sélectionné. Il ne faut pas confondre le nom de l'objet et la valeur de texte qu'il contient.



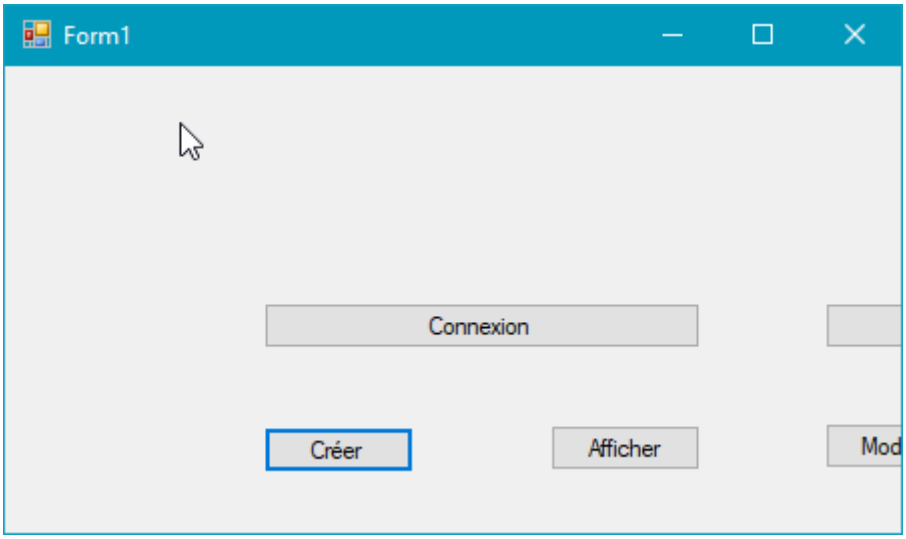

Sur le bouton en haut à droite qui sera le bouton "Quitter", voici les propriétés :

| Nom de la propriété | Valeur de la propriété |
|---|---|
| (name) | bt_Quitter |
| Text | Quitter |

Corrigez toutes les valeurs des boutons conformément à la maquette de la page d'accueil.

Une fois terminé, vous pouvez sauvegarder le projet, puis le démarrer : il doit ressembler à la maquette, et le comportement des boutons doit réagir au survol de la souris ainsi qu'au clic mais sans action associée.

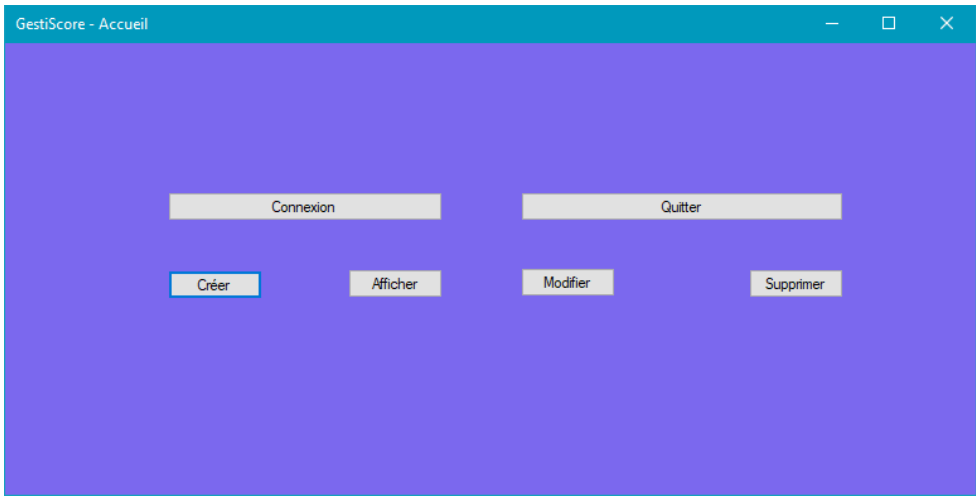
Le défaut de cette forme (outre son nom) est qu'on peut changer la taille de la fenêtre mais les boutons ne changent pas de place, ce qui peut entraîner un affichage incohérent.




Éditer les propriétés suivantes de la fenêtre :

| Nom de la propriété | Valeur de la propriété |
|---|---|
| (name) | w_Accueil |
| Text | GestiScore - Accueil |
| BackColor | MediumSlateBlue |
| FormBorderStyle | FixedDialog |

Retestez votre forme qui doit désormais ne plus être redimensionnable et elle doit avoir une couleur proche de ce que nous avons mis dans la maquette (à ce niveau, vous pouvez choisir une autre couleur, ce n'est pas ce qui compte).



### D.2.3 Comprendre le code

#### *D.2.3.a Fichier Form1.Designer.cs*

Ce fichier contient le code de la forme graphique. Ce qui est remarquable :

La classe est une classe partielle qui contient une méthode InitializeComponent() :

La méthode InitializeComponent() est privée et ne renvoie rien (void), mais elle contient la définition de tout les éléments dans la fenêtre.

```
this.bt_Creer = new System.Windows.Forms.Button();
this.bt_Afficher = new System.Windows.Forms.Button();
this.bt_Modifier = new System.Windows.Forms.Button();
this.bt_Supprimer = new System.Windows.Forms.Button();
this.bt_Connecter = new System.Windows.Forms.Button();
this.bt_Quitter = new System.Windows.Forms.Button();
```

On constate que la création d'un objet bt_Afficher se fait en instanciant une classe System.Windows.Forms.Button() ce qui implique beaucoup d'héritage.

D'autre part, on retrouve les instructions pour dessiner le bouton et y mettre ses propriétés :

```
this.bt_Afficher.Location = new System.Drawing.Point(272, 179);
this.bt_Afficher.Name = "bt_Afficher";
this.bt_Afficher.Size = new System.Drawing.Size(75, 23);
this.bt_Afficher.TabIndex = 1;
this.bt_Afficher.Text = "Afficher";
this.bt_Afficher.UseVisualStyleBackColor = true;
```

On peut comparer avec un code HTML d'un formulaire et le style CSS associé.

#### *D.2.3.b Fichier Form1.cs*

Il s'agit d'un simple fichier C# qui crée la classe de notre fenêtre en héritant de la classe Form (appartenant à .NET Framework) et qui a comme constructeur un simple appel de la méthode InitializeComponent().

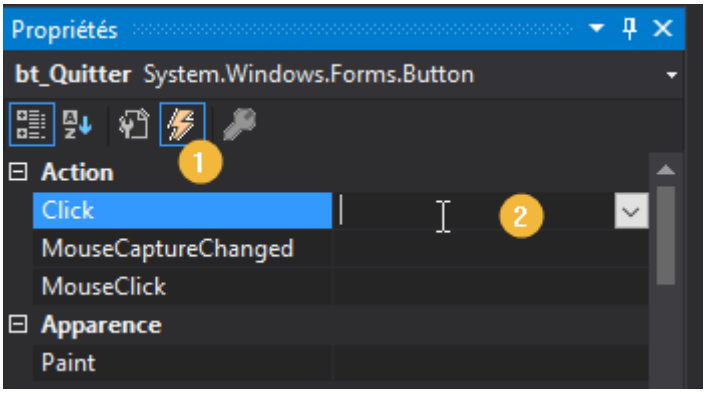
```
namespace GestiScore {
    public partial class w_Accueil : Form {
        public w_Accueil() {
            InitializeComponent();
        }
    }
}
```

#### *D.2.3.c Fichier Program.cs*

C'est le code qui démarre notre application, en instanciant notre classe w_Accueil.

## D.3 Création des événements

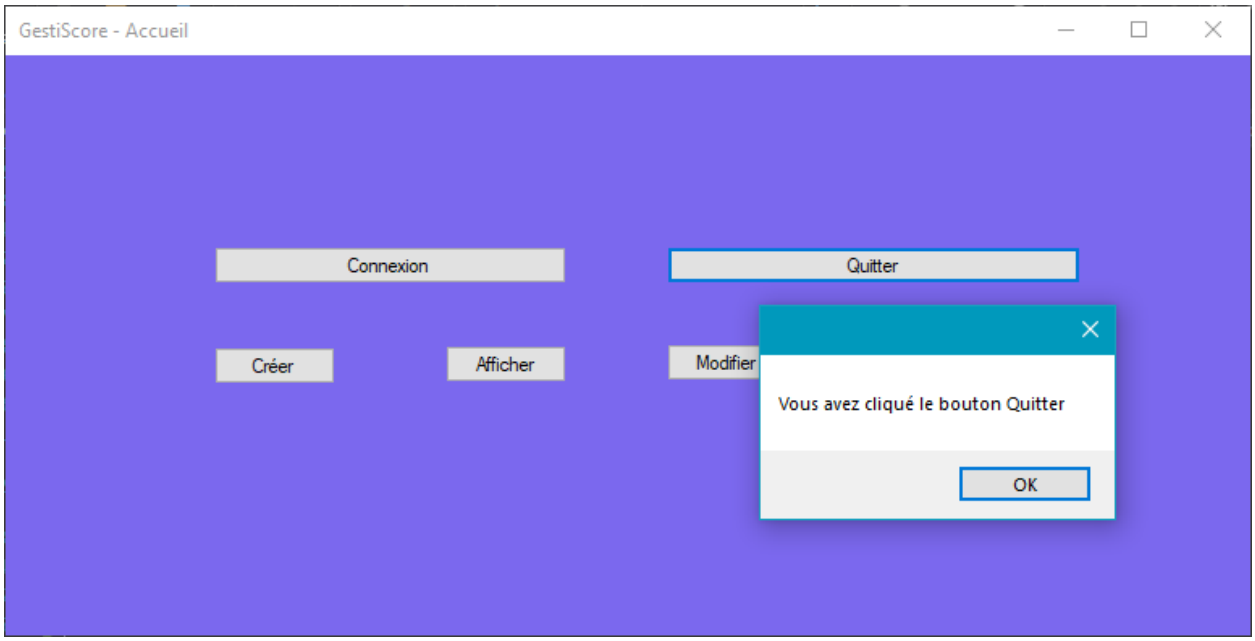
Comme JavaScript, et contrairement au mode console, le codage en Windows Form est événementiel. Il faut donc préciser au système que nous voulons surveiller le bouton et y associer une méthode. Nous pouvons désormais double-cliquer le bouton "Quitter" ou bien 1) aller dans sa propriété Click (sous l'onglet éclair) et 2) saisir un nom de méthode et appuyer sur la touche [Entrée] :




Dans mon cas, j'ai double-cliqué et j'ai modifié la méthode comme suit :

```
private void bt_Quitter_Click(object sender, EventArgs e) {
    MessageBox.Show("Vous avez cliqué le bouton Quitter");
}
```

Sauvegardez et testez votre code (cliquez sur le bouton "Quitter")




Vous savez maintenant agir sur un événement.

### D.3.1 Utilisation des MessageBox

Les messageBox sont des boites de dialogue simplifiées qui vous évitent de créer une forme : c'est une sorte de modèle que vous pouvez utiliser avec 4 paramètres :

- Le contenu de la boite principale (texte)
- Le contenu du titre de la boite (texte)
- Les boutons :
  - OK (de base)
  - Oui/Non
  - Ok/Annuler
  - ...
- Les icônes systèmes :
  - Information (MessageBoxIcon.Exclamation)
  - Attention (MessageBoxIcon.Warning)
  - Erreur (MessageBoxIcon.Error)

Puisqu'elle propose des boutons, on peut récupérer le choix de l'utilisateur comme ceci :

```
var choix = MessageBox("Hello", "Titre", MessageBoxButtons.OKCancel, MessageBoxIcon.Warning);

if (choix == DialogResult.Yes) { ... }
if (choix == DialogResult.No) { ... }
(choix == DialogResult.Yes) { ... }
```

Les différentes possibilités sont accessibles en tapant DialogResult. Et en laissant Visual Studio afficher les propositions.

### D.3.2 Quitter l'application

Le choix étant accessible, il reste à quitter réellement l'application.

Dans le fichier Program.cs on constate que la ligne suivante démarre l'application :

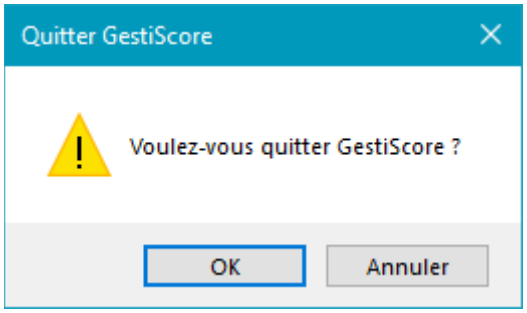
```
Application.Run(new w_Accueil());
```

Il suffit donc de fermer l'application depuis n'importe où avec l'instruction suivante :

```
if (choix == DialogResult.Yes) {
    Application.Exit();
}
```

Remplacez donc la méthode bt_Quitter_Click() par celle-ci :

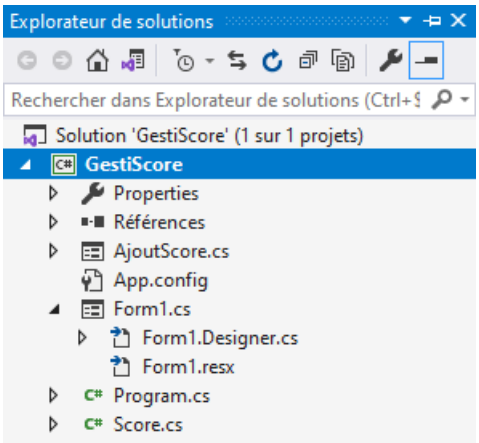
```
private void bt_Quitter_Click(object sender, EventArgs e) {
    var choix = MessageBox.Show("Voulez-vous quitter GestiScore ?", "Quitter GestiScore",
MessageBoxButtons.OKCancel, MessageBoxIcon.Warning);
    if (choix == DialogResult.OK) {
        Application.Exit();
    }
}
```




## D.4 Création d'une fenêtre spécifique

### D.4.1 Création de la fenêtre w_AjoutScore

La fenêtre de dialogue par défaut ne permet pas la saisie des informations que nous voulons ajouter à la liste (qui elle sera décrite plus loin). Voici comment créer une nouvelle fenêtre sur mesure :




Dans l'explorateur de solutions (à droite dans Visual Studio), faire un clic droit sur la solution (en gras) : GestiScore.

Sélectionner Ajouter > Formulaire (Windows Forms)

Choisir un nom logique, par exemple w_AjoutScore.cs (w_ pour windows, suivi par l'utilité du formulaire).

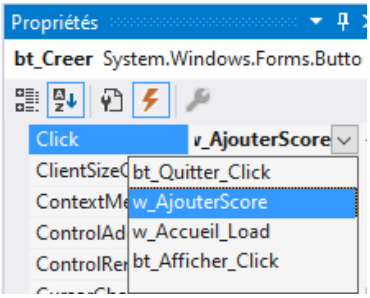
Dans l'éditeur, créer la forme qui permet de créer un score, tel que décrit dans la maquette.

| Nom de la propriété | Valeur de la propriété |
|---|---|
| (name) | w_AjoutScore |
| Text | GestiScore – Ajouter un score |
| FormBorderStyle | FixedDialog |

Dans les champs à ajouter à la fenêtre, choisissez correctement les identifiants, comme ceux ci-dessous :

| Champs | Propriétés : Valeur |
|---|---|
| TextBox | (Name) : txt_Nom |
| TextBox | (Name) : txt_Point |
| Button | (Name) : bt_ValiderAjout<br>Text : Valider |
| Button | (Name) : bt_AnnulerAjout<br>Text : Annuler |

Désormais, la nouvelle forme est utilisable, il faut l'afficher, lorsque l'utilisateur clique sur le bouton [Créer] de la fenêtre principale.




Sélectionnez le bouton [Créer] de la fenêtre principale,
Dans les propriétés, choisir les actions (icône 'éclair'),
Dans la propriété 'Click' saisir w_AjouterScore pour définir le nom de la méthode associée,
Validez votre saisie (touche [Entrée]).

Voici le code à placer dans cette méthode :

```
private void w_AjouterScore(object sender, EventArgs e) {
    Form ajouterScore = new AjoutScore();         // instanciation de notre fenêtre
    ajouterScore.Show();
}
```

La fenêtre est créée dans la mémoire de l'ordinateur puis affichée au moment du clic sur le bouton.

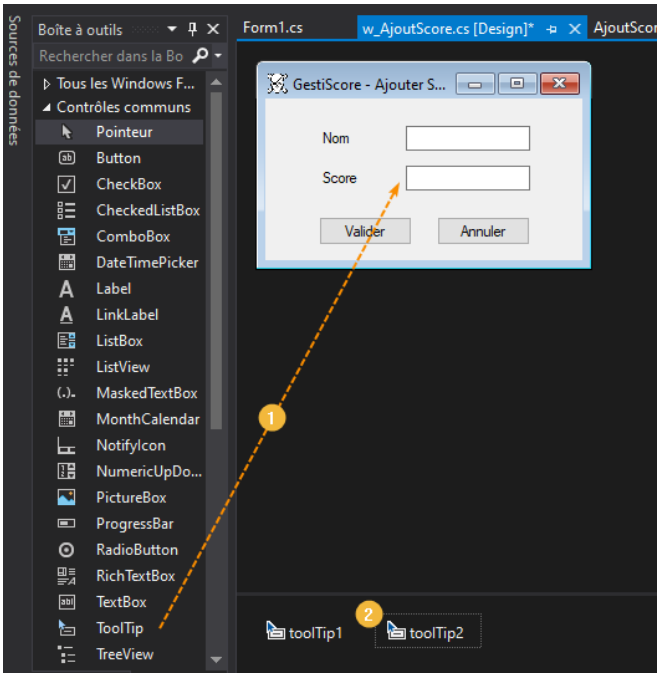
Durant la présence de la nouvelle fenêtre, les événements sur les autres fenêtres continuent à exister : clics, survols, etc.

Nous verrons comment rendre notre fenêtre incontournable et éviter que la fenêtre w_AjoutScore n'existe plusieurs fois ou ne soit mise au second plan.

### D.4.2 Ajout des bulles d'information (ToolTips)

Nous profitons de cette fenêtre pour ajouter une aide visuelle sur les champs, qui apparaîtront au survol de ceux-ci.

Il faut insérer des objets "ToolTip", mais le comportement n'est pas naturel




En effet, le composant ToolTip n'apparaît pas dans la fenêtre mais dans une partie de l'IDE, sous la forme.

Les ToolTips doivent être associés à un champ (ou composant) par une propriété du composant lui-même.

Insérez autant de composant ToolTip que de forme de messages différents (avec icône stop, avec icône attention, etc.).

🚫 Il n'est pas nécessaire de créer un Tooltip par composant !

Voici les propriétés de toolTip1 :

| Nom de la propriété | Valeur de la propriété |
|---|---|
| (name) | toolTip1 |
| Title | Aide |
| icon | Info |
| isBalloon | False |

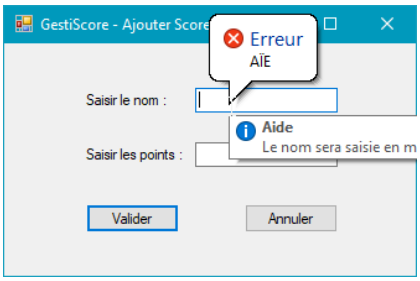
Voici les propriétés de toolTip2 :

| Nom de la propriété | Valeur de la propriété |
|---|---|
| (name) | toolTip2 |
| Title | Erreur |
| icon | Error |
| isBalloon | True |

Dans le composant txt_Nom, voici les éléments des Tooltips :

| Nom de la propriété | Valeur de la propriété |
|---|---|
| ToolTip sur toolTip1 | Le nom sera saisi en majuscule |
| ToolTip sur toolTip2 | AÏE |

En laissant la souris immobile sur le champ, pendant l'exécution du programme, vous devriez obtenir l'effet suivant :




Il est possible de modifier le contenu des propriétés par programmation :

```
// Modification du texte d'information
toolTip2.SetToolTip(txt_Nom, "Erreur de saisie, le champ est vide !");

// Suppression du texte d'information
toolTip2.SetToolTip(txt_Nom, null);
```

### D.4.3 Fenêtre modale ou non modale

Dans notre code actuel, il est possible de créer plusieurs fenêtres d'ajout de score, en cliquant plusieurs fois sur le bouton [Créer]. On dit que cette fenêtre est non modale : elle ne bloque pas l'application en attendant une action. Dans notre cas, cela pose problème et nous allons la rendre modale.

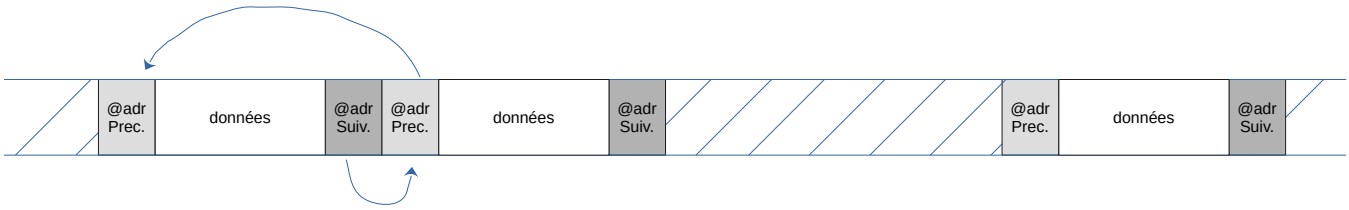
```
// Instruction pour fenêtre non-modale
ajouterScore.Show();

// Instruction pour fenêtre modale
ajouterScore.ShowDialog();
```

Comme vous pouvez le constater, hormis la méthode d'appel de la fenêtre, il n'y a rien à modifier dans notre code. Désormais, il est impossible de cliquer sur d'autres boutons de l'application tant que la fenêtre "Ajouter Score" n'est pas fermée.

## D.5 Gestion d'une collection

Une collection est une liste (chaînée) qui peut changer dans le temps : ajout, suppression et modification d'éléments. La liste contient un type d'élément comme des entiers, des chaînes de caractères ou bien… des classes.

Contrairement à un tableau (dans lequel les éléments sont stockés de manière continue dans la mémoire), une liste (ou collection) peut s'agrandir, car elle utilise un système de pointeur pour trouver l'élément précédent et l'élément suivant.




La création d'une liste utilise une syntaxe particulière :

```
List<type> nomListe = new List<type>();
```

Les instructions sur la liste sont :

| Ajout | Suppression | Suppression sur index | Insérer sur index |
|---|---|---|---|
| nomListe.Add(x) | nomListe.Remove(x) | nomListe.RemoveAt(t) | NomListe.Insert(t, x) |

On a également des fonctions pour la recherche.

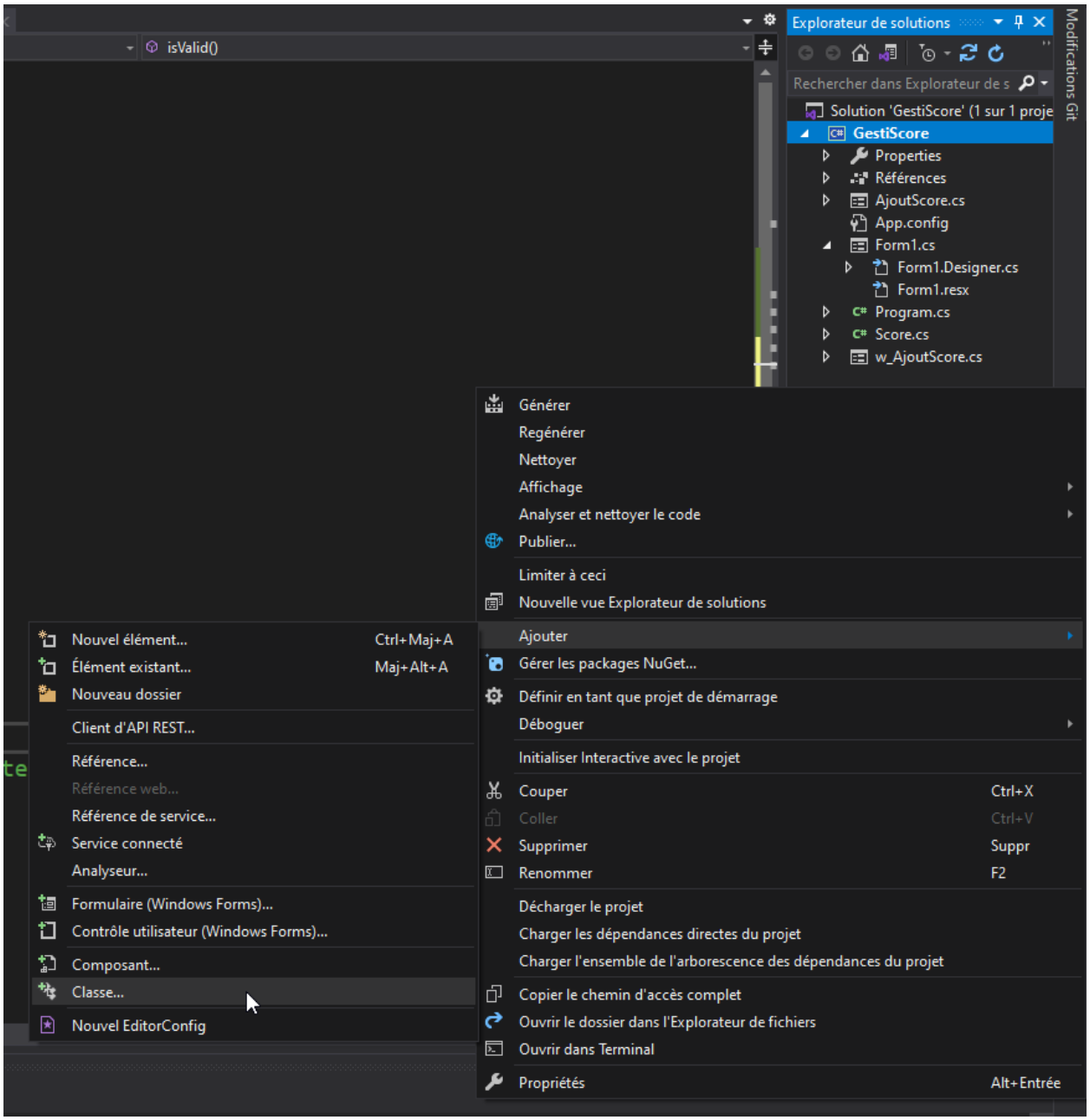
| Recherche d'index | Recherche binaire |
|---|---|
| Int res = nomListe.IndexOf(x) | Int res = nomListe.BinarySearch(x) |

### D.5.1 Création de la liste dans l'application

La création d'une liste doit se faire dans la classe Program, comme un attribut public et statique. Le type de la liste est notre classe Score, que nous allons donc créer comme nouveau fichier dans le même workspace.

### *D.5.1.a Création de la classe Score*

Dans la partie "Explorateur de solutions", faire un clic droit et choisir Ajouter > Classe…




Dans la fenêtre qui s'affiche, changez le nom du fichier par Score.cs et validez.

Nous pouvons maintenant créer les membres de notre classe.

```
namespace GestiScore {
    class Score {
        private int IdScore;
        private String nom;
        private int score;

        public void setNom(String nom) { this.nom = nom; }
        public String getNom() { return this.nom; }
        public void setScore(int score) { this.score = score; }
        public int getScore() { return this.score; }
        public Boolean isValid() {
            // incapable de me rappeler qu'est-ce que je pensais tester
            return false;
        }
    }
}
```

Après enregistrement du fichier, la classe Score est instanciable, nous pouvons désormais créer une liste utilisant ce type.

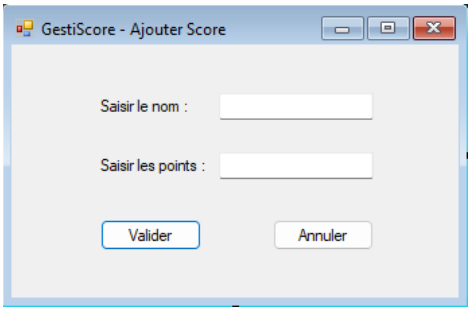
### *D.5.1.b Création de la liste d'objet de type Score*

Dans le fichier Program.cs nous créons une liste que nous rendons publique pour plus de facilité :

```
    static class Program {
        public static List<Score> listeScore = new List<Score>();

        [STAThread]
        static void Main() {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new w_Accueil());
        }
    }
```

### *D.5.1.c Utilisation de la liste dans la fenêtre AjoutScore*

Il suffit de double-cliquer sur le bouton de validation de la fenêtre "Ajouter Score" pour ouvrir le code associé à ce bouton :




```
private void bt_ValiderAjout_Click(object sender, EventArgs e) {
    Score x = new Score();
    Program.listeScore.Add(x);
}
```

La liste listeScore n'est pas visible directement dans la classe AjoutScore, il faut donc préciser dans quelle classe elle se trouve.

```
private void bt_ValiderAjout_Click(object sender, EventArgs e) {
    Score x = new Score();
    x.setNom(txt_Nom.Text.ToUpper());
    x.setScore(Convert.ToInt32(txt_Points.Text));
    Program.listeScore.Add(x);
}
```

La fonction permet de mettre en lettres capitales le champ nom et de convertir le champ txt_Point en un nombre entier.