



Développement d'applications .NET

Support de cours partie II : Développement WinForms

Niveau : 2A Cycle Ingénieur

Filières : GI/IA/ROC

Enseigné par : Prof. KADDARI Zakaria

Année universitaire
2024/2025



I. Table des matières

■	I. Table des matières.....	2
■	II. Introduction	5
■	III. Environnement de développement intégré.....	5
	A. L'interface de Visual Studio	5
	1. Structure des applications	5
	2. Interface graphique.....	5
	a) Fenêtre des propriétés	7
	b) L'éditeur de code.....	7
■	IV. Outils de débogage	8
	A. Les erreurs de syntaxe	8
	B. Les erreurs d'exécution	9
	1. Erreur de conception	10
	2. Erreurs de l'utilisateur.....	10
	a) Capturer les erreurs avec Try – Catch	10
	C. Les erreurs de logique :.....	11
	1. Débogage d'une application.....	11
	a) Suspendre l'exécution.....	11
	b) Débogage pas à pas.....	12
■	V. Consulter l'aide.....	12
■	VI. Générer un fichier exécutable	13
■	VII. Règles de réalisation d'une interface	14
	A. Les concepts de la programmation événementielle	14
	1. Les conséquences d'une interface ratée	14
	2. Les avantages d'une interface réussie.....	14
	3. Multi-fenêtrage	14
	4. Les icônes.....	15
	5. Les menus	15
	B. Applications SDI et MDI.....	15
	1. SDI.....	15
	2. MDI	15
■	VIII. Les bases des interfaces graphiques.....	16
	A. Un premier projet	16

B. Exemple pratique.....	20
C. Les composants de base	30
1. Les propriétés communes des objets	30
a) Name.....	30
b) Text.....	31
c) Enabled.....	31
d) Visible.....	31
e) Font.....	31
f) BackColor ForeColor.....	31
2. Formulaire Form	32
a) Name.....	32
b) Text.....	32
c) Icon.....	32
d) WindowState	32
e) ControlBox	32
f) MaximizeBox	32
g) MinimizeBox	32
h) FormBorderStyle	32
i) StartPosition	33
j) Opacity	33
k) Les dialogues modale et non modale	33
l) Formulaire d'avant plan.....	33
m) Les événements	33
3. Etiquettes Label et boites de saisie TextBox.....	35
4. Listes deroulantes ComboBox	39
5. Composant ListBox	42
6. Cases a cocher CheckBox, boutons radio ButtonRadio.....	46
7. Variateurs ScrollBar	48
8. Événements souris.....	50
9. Créer une fenêtre avec menu	53
10. Composants non visuels.....	56
a) Boites de dialogue OpenFileDialog et SaveFileDialog	56
b) Boites de dialogue FontColor et ColorDialog	60
11. Timer	62
12. Regroupement de contrôles	65
a) GroupBox et Panel	65
b) PictureBox	65
c) TabControl	65



المدارس الوطنية للذكاء الاصطناعي والرقمنة - بنهار
ECOLE NATIONALE DE L'INTELLIGENCE ARTIFICIELLE ET DU DIGITAL - BENHAR
«ECAD» «IEIA» «DEAD» «DEIAC» «DOEC» - GOS

Développement d'applications .NET

Partie II : WinForms

I. Introduction

L'objectif de cette partie est de présenter les concepts fondamentaux de la programmation événementielle ainsi que les éléments nécessaires à une bonne prise en main de la plateforme de développement Microsoft .Net ainsi que des environnements classiques qui y sont dédiés.

II. Environnement de développement intégré

A. L'interface de Visual Studio

1. Structure des applications

La première chose à faire, c'est d'assimiler l'architecture des édifices que nous allons créer et le vocabulaire qui va avec.

Une application C#, c'est :

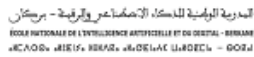
- Un ensemble de fichiers formant ce qu'on appelle le code source, écrit dans le langage C#.
- Un fichier exécutable, produit à partir de ce code source. Rappelons que le fait d'engendrer un fichier exécutable à partir du code source s'appelle la compilation.
- Le point de départ d'une application C#, c'est une solution. Lorsqu'on crée une nouvelle solution, C# demande pour celle-ci un nom et un répertoire.
- Une solution est un fichier ***.sln** contenant les informations de la solution.
- Dans une solution C#, nous allons devoir insérer nos applications, c'est-à-dire nos **projets**. Si on le souhaite, une même solution peut contenir plusieurs projets.
- Dans chaque projet il y'a un certain nombre d'éléments de base. Ces éléments sont, pour l'essentiel, des **Form**, ou formulaires. Une application Windows basique compte un seul formulaire, et une application complexe peut en rassembler plusieurs dizaines. Chaque formulaire sera sauvegardé dans un fichier différent, dont l'extension sera ***.cs**. Il faut noter que c'est dans ces fichiers ***.cs** que se trouve le code proprement dit.

2. Interface graphique

Les produits de la gamme Visual Studio partagent le même environnement de développement intégré (IDE). L'IDE est composé de plusieurs éléments : la barre d'outils Menu, la barre d'outils Standard, différentes fenêtres Outil ancrées ou masquées automatiquement sur les bords gauche, inférieur et droit, ainsi que l'espace d'éditeur. Les fenêtres Outil, menus et barres d'outils disponibles varient en fonction du type de projet ou de fichier dans lequel vous travaillez.

Lorsqu'on va la programmer via Visual Studio, une application va donc toujours pouvoir être abordée sous deux angles complémentaires :

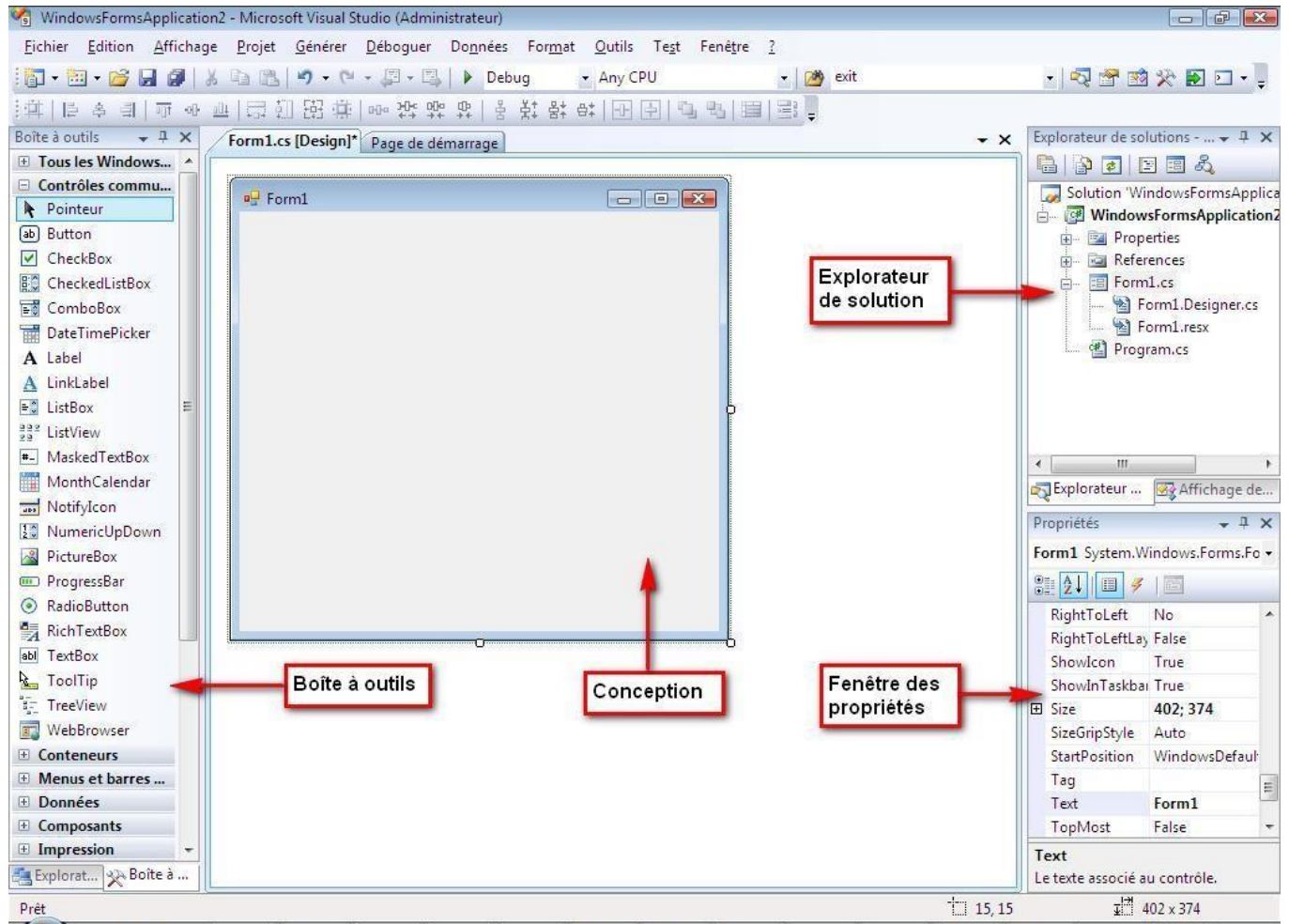
- L'aspect graphique, visuel, bref, son interface. Dans la fenêtre principale de C#, nous pourrions facilement aller piocher les différents objets que nous voulons voir



Développement d'applications .NET

Partie II : WinForms

figurer dans notre application, les poser sur notre formulaire, modifier leurs propriétés par défaut, etc :

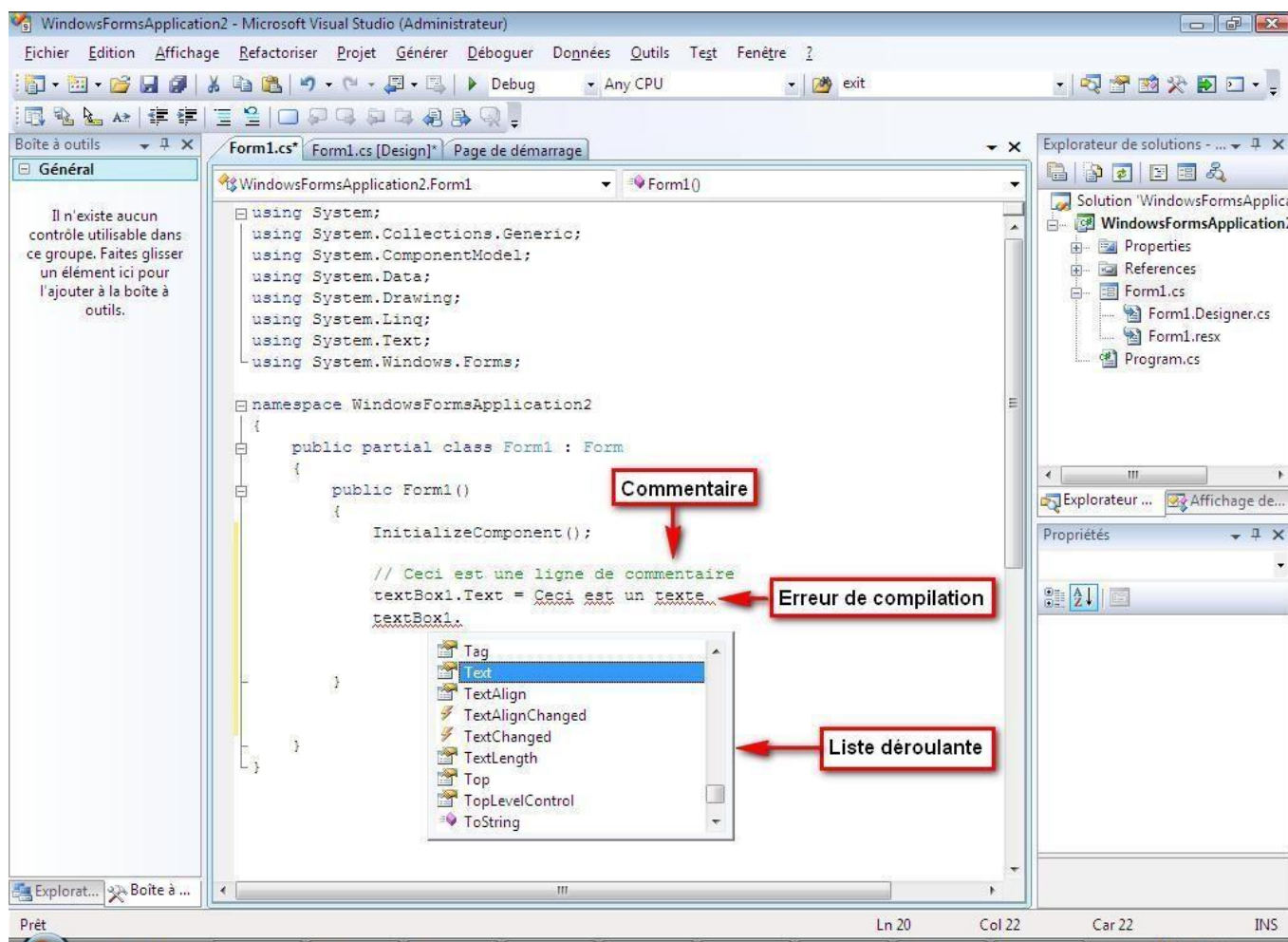


- Le code proprement dit, où nous allons entrer les différentes procédures en rapport avec le formulaire en question :



Développement d'applications .NET

Partie II : WinForms



a) Fenêtre des propriétés

Chaque fois qu'un contrôle (ou plusieurs) est sélectionné, la fenêtre des propriétés (située en standard à droite de l'écran) affiche les valeurs associées à ce contrôle. C'est-à-dire que se mettent à jour la liste des propriétés et la valeur de ces propriétés.

b) L'éditeur de code

Passons au code. Visual Studio, dans sa grande magnanimité, va tâcher de faire au mieux pour nous faciliter la vie. Il va en fait décrypter notre code au fur et à mesure de sa rédaction, et nous donner en temps réel des indications via des codes de couleur, comme on peut le voir sur l'image ci-dessus. Ainsi :

- Les mots-clés du langage seront portés en bleu.
- Les commentaires seront en vert.
- Enfin, toute ligne comportant une faute de syntaxe, ou posant un problème au compilateur, sera immédiatement soulignée.

III. Outils de débogage

Cette section présente les outils de débogage offerts par la plateforme, nous ferons les illustrations à partir de Visual Studio 2008 mais ceci reste valable même pour la version 2022.

En programmation on rencontre trois types d'erreurs qui sont :

- Les erreurs de syntaxe
- Les erreurs d'exécution
- Les erreurs de logique

A. Les erreurs de syntaxe

Elles surviennent en mode conception quand on tape le code :

Exemples :

```

A + 1 = B; // Erreur d'affectation

f.ShowDialog(); // Faute de frappe, il fallait taper ShowDialog

int i; textBox1.Text = i; // Affectation d'un Integer à une propriété
                          // text qui attend une String
    
```

Dans ces cas C# souligne en **ondulé rouge** le code. Il faut mettre le curseur sur le mot souligné, l'explication de l'erreur apparaît.

Exemple : Propriété Text d'un label mal orthographiée:

```

label1.Text() = "13";
    
```

string Label.Text

Erreur :

Un membre 'System.Windows.Forms.Control.Text' ne pouvant pas être appelé ne peut pas être utilisé comme une méthode.

Il faut les corriger immédiatement en tapant le bon code (ici 'Text').

En bas il y a aussi une fenêtre ; "liste des erreurs" :

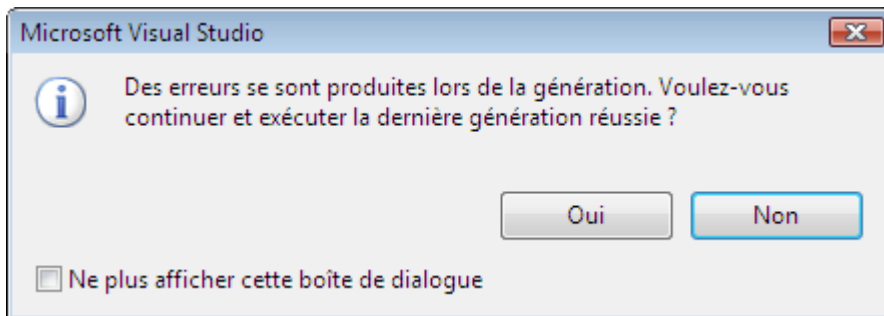
Liste d'erreurs					
5 erreurs		0 avertissements		0 messages	
	Description	Fichier	Ligne	Colonne	Projet
1	La partie gauche d'une assignation doit être une variable, une propriété ou un indexeur	Form1.cs	23	6	WindowsFormsAppli
2	'System.Windows.Forms.Form' ne contient pas une définition pour 'ShowDialog' et aucune méthode d'extension 'ShowDialog' acceptant un premier argument de type 'System.Windows.Forms.Form' n'a été trouvée (une directive using ou une référence d'assembly est-elle manquante ?)	Form1.cs	25	8	WindowsFormsAppli
3	Impossible de convertir implicitement le type 'int' en 'string'	Form1.cs	27	29	WindowsFormsAppli

Développement d'applications .NET

Partie II : WinForms

Elle affiche tous les problèmes ; pour atteindre le code correspondant à une de ces erreurs, double-cliquez sur une des lignes de la liste.

Si vous exécutez le programme dans l'IDE alors qu'il y a un problème, C# demande si on veut exécuter la dernière génération réussie :



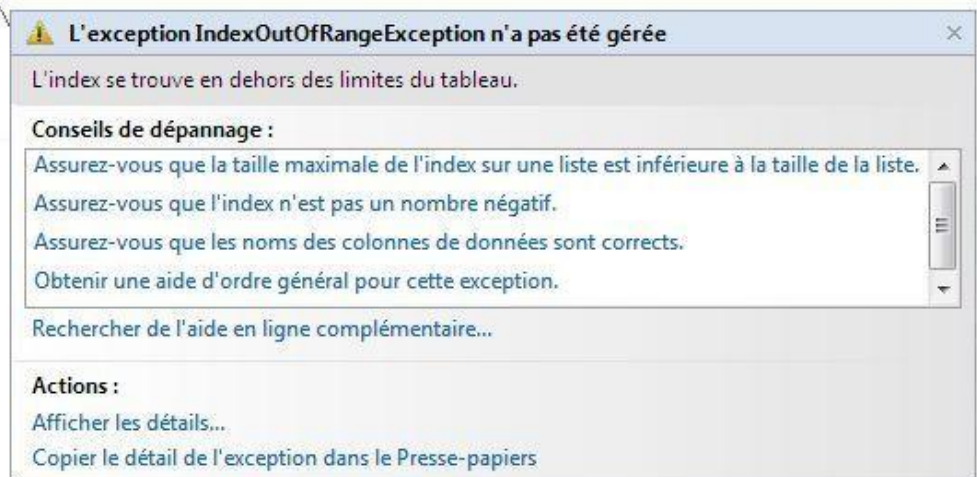
Si vous tapez 'oui' C# exécute la dernière version qui a été générée correctement, mais PAS de code source actuel qui contient des erreurs!!

B. Les erreurs d'exécution

Elles surviennent **en mode Run** ou lors de l'**utilisation de l'exécutable: une instruction ne peut pas être effectuée**. Quand on utilise l'exécutable: **Le logiciel s'arrête brutalement, c'est très gênant!! Pour l'utilisateur c'est un 'BUG'** Il y a levée d'une **exception**, voila ce que cela donne dans l'IDE:

Exemple: tenter d'accéder à un élément d'un tableau qui n'existe pas (l'indice est trop grand cela entraîne une exception 'OutOfRangeException'):

```
private void button1_Click(object sender, EventArgs e)
{
    string[] t = new string[3];
    t[3] = "toto";
}
```



Les erreurs d'exécution sont soit des erreurs de conception ou des erreurs de l'utilisateur.

1. Erreur de conception :

Exemples :

- Ouvrir un fichier qui n'existe pas (On aurait du vérifier qu'il existe avant de l'ouvrir!).
- Utiliser un index d'élément de tableau supérieur au nombre d'élément.
- Envoyer un mauvais paramètre à une fonction.

2. Erreurs de l'utilisateur :

Exemples :

- On lui demande de taper un chiffre, il tape une lettre ou rien puis valide.

Il faut toujours vérifier ce que fait l'utilisateur et prévoir toutes les possibilités.

- Si je demande à l'utilisateur de tapez un nombre entre 1 et 10, il faut:
 - ✓ Vérifier qu'il a tapé quelque chose.
 - ✓ Que c'est bien un chiffre (pas des lettres).
 - ✓ Que le chiffre est bien entre 1 et 10.
 - ✓ Sinon il faudra reposer la question.

a) Capturer les erreurs avec Try – Catch

```
try
{
    // Appel de la fonction susceptible de générer l'exception
}
catch (Exception e)
{
    // traiter l'exception e
}
// instruction suivante
```

Exemples :

```
try
{
    int il = int.Parse("abcd");
    Console.WriteLine(il);
}
catch (Exception e)
{
    Console.WriteLine("Erreur : "+e.Message);
}
```

C. Les erreurs de logique :

Le programme fonctionne, pas d'erreurs apparentes, mais les résultats sont erronés, faux.

Il faut faire des tests dans les conditions réelles avec des données courantes, mais aussi avec des données remarquables pour voir si les résultats sont cohérents et exacts. Une fois l'erreur trouvée, il faut en déterminer la cause et la corriger.

Pour cela il faut analyser le fonctionnement du programme pas à pas, instruction par instruction en surveillant la valeur des variables

1. Débogage d'une application

Les erreurs de logique sont plus difficiles à détecter. Le code est syntaxiquement correct, mais il ne réalise pas les opérations prévues. C'est là qu'intervient le débogage afin de diagnostiquer l'origine de l'erreur.

Pour déboguer, il faut lancer l'exécution du programme puis,

- Suspendre l'exécution à certains endroits du code.
- Voir ce qui se passe puis faire avancer le programme pas à pas.
- Afficher des informations de débogage quand le programme tourne.

a) Suspendre l'exécution

Pour démarrer et arrêter l'exécution, on utilise les boutons suivants :



On lance le programme avec le premier bouton, on le suspend avec le second, on l'arrête définitivement avec le troisième, et le dernier bouton permet de redémarrer le débogage.

On peut suspendre (l'arrêter temporairement) le programme :

- Avec le second bouton.
- Grâce à des points d'arrêt (pour définir un point d'arrêt en mode de conception, cliquez en face d'une ligne dans la marge grise : la ligne est surlignée en marron. Quand le code est exécuté, il s'arrête sur cette ligne marron).



```
int[] t = new int[7];
for (int i = 0; i <= 6; i++)
{
    t[i] = i * i;
}

for (int i = 0; i <= 6; i++)
{
    Console.WriteLine(t[i]);
}
```



Développement d'applications .NET

Partie II : WinForms

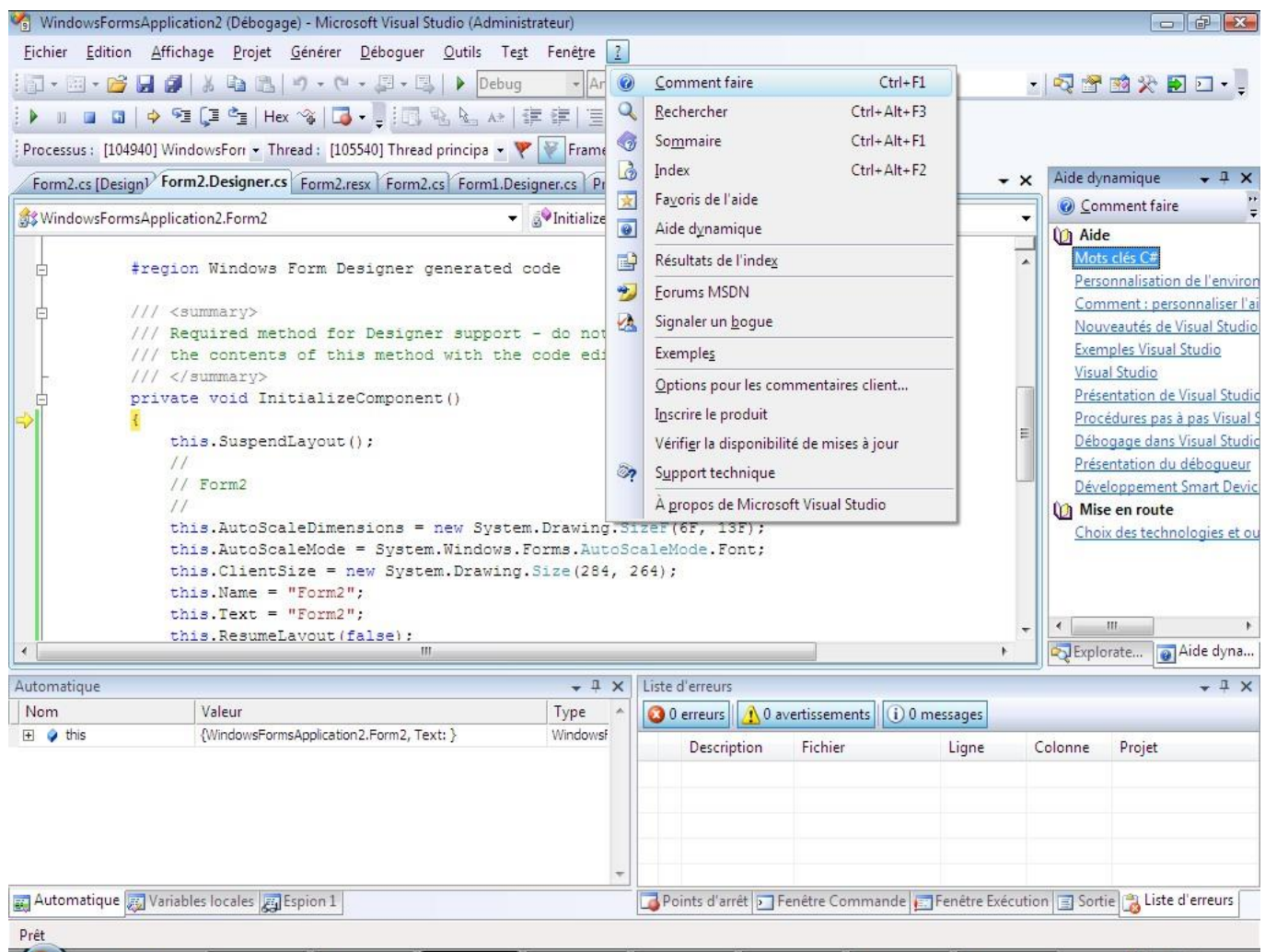
b) Débogage pas à pas

Quand le programme est suspendu, on peut **observer les variables, déplacer le point d'exécution**, on peut aussi faire marcher le programme **pas à pas (instruction par instruction)** et observer **l'évolution de la valeur des variables**, on peut enfin modifier la valeur d'une variable afin de tester le logiciel avec cette valeur.

- **F11** permet l'exécution pas à pas, instruction par instruction (y compris des procédures appelées: si il y a appel à une autre procédure, le pas à pas saute dans l'autre procédure)
- **F10** permet le pas à pas (sans détailler les procédures appelées: exécute la procédure appelée en une fois)

IV. Consulter l'aide

La plupart des interfaces de développement .Net offre un certain nombre d'item d'aide communs, pour cela utiliser le menu ? :



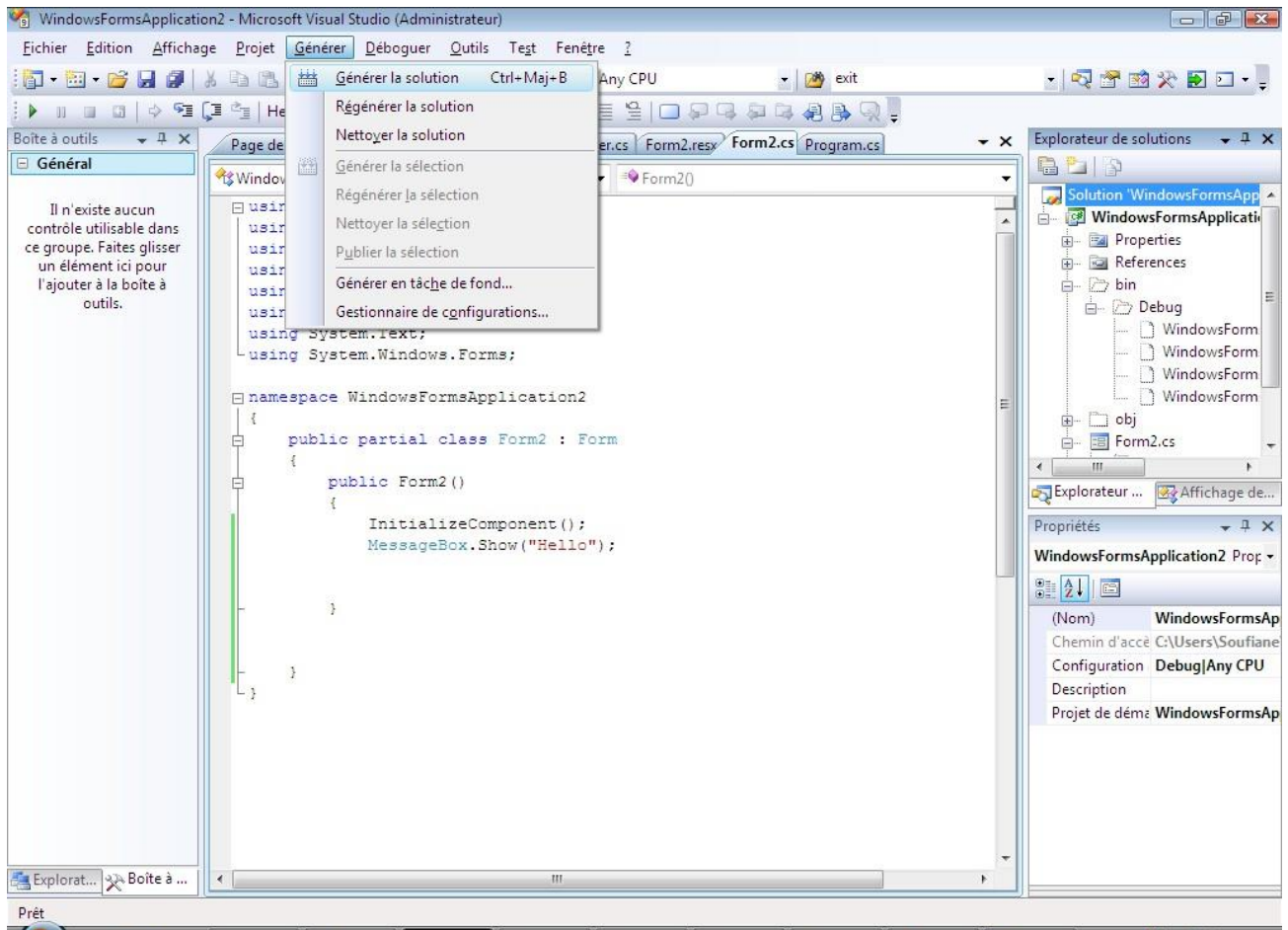


Développement d'applications .NET

Partie II : WinForms

V. Générer un fichier exécutable

La génération de l'exécutable dans un environnement .Net s'effectue au travers du menu « Générer » qui propose comme possibles options la « Génération de la solution » ou sa « Régénération ».



Quand on choisit l'une des deux options un exécutable ainsi qu'une ou plusieurs Dll (Dynamic Link Library) sont générées et placées dans un dossier situé à la racine du projet et qui porte le nom de **Bin**, dépendamment de l'option de compilation choisi l'exécutable sera plus spécifiquement déployé dans un sous dossier du dossier BIN intitulé **Debug** ou **Release**.

VI. Règles de réalisation d'une interface

A. Les concepts de la programmation événementielle

La programmation événementielle est basée sur l'interaction avec l'utilisateur de l'application. Les différents objets de l'interface vont être manipulés indépendamment les uns des autres, en fonction de la tâche qu'il souhaite réaliser.

C'est l'utilisateur qui contrôle l'application. De ce fait l'ergonomie d'une interface utilisateur est un des éléments importants de l'application.

Elle peut être déclinée selon deux axes :

- La présentation des informations à l'utilisateur et leur lisibilité ;
- L'interaction existant entre l'utilisateur et l'application.

1. Les conséquences d'une interface ratée

- Confusion (utilisateur perdu dans l'application ne sachant plus où aller, ni que faire) ;
- Frustration (utilisateur ne sachant pas atteindre son but) ;
- Panique (peur de l'utilisateur d'avoir perdu des données ou détruit des fichiers) ;
- Stress (charge de travail et d'informations reçues trop importante pour l'utilisateur) ;
- Ennui (utilisateur fatigué d'avoir à répéter des étapes fastidieuses et non productives)
- Sous-utilisation (utilisateur refusant de se servir d'une application trop complexe)
- Sur-utilisation (application nécessitant de multiples aller-retour entre les fenêtres).

2. Les avantages d'une interface réussie

- Acceptation (utilisateur s'appropriant l'application plutôt que de la rejeter) ;
- Utilisation (application utilisée à bon escient et de manière efficace) ;
- Formation réduite (apprentissage minimal par l'utilisateur retrouvant les automatismes liés à d'autres applications).

3. Multi-fenêtrage

Windows est une interface graphique multi-fenêtres. C'est à dire que l'écran est fractionné en plusieurs parties où s'exécutent les applications. Ces fenêtres peuvent se chevaucher, se recouvrir mutuellement ou être disposées côte à côte permettant ainsi de partager l'espace de travail.

Windows permet de travailler sur plusieurs applications à la fois. L'environnement multi-contextes est la partie visible du multi-tâches.

L'utilisation du multi-contextes présente par contre des désagréments pour le développeur. Chaque fenêtre et ainsi chaque application doit être clairement identifiée afin que l'utilisateur puisse aisément s'y retrouver.

De plus, plusieurs occurrences d'une même application peuvent fonctionner simultanément, chacune d'entre elles devant être repérée. Cela implique que le programmeur devra réfléchir à son application en tenant compte de l'environnement qui la supporte et devra réfléchir à la

Développement d'applications .NET

Partie II : WinForms

cohérence entre cette application et l'environnement. Dans ces conditions deux remarques s'imposent :

- L'application devra pouvoir communiquer avec l'environnement ;
- L'application ne devra pas monopoliser les ressources communes du système.

L'utilisateur doit toujours pouvoir passer d'une application à une autre, en désignant la fenêtre où elle s'exécute. Cette fenêtre passe alors au premier plan et devient active. Les autres fenêtres se retrouvant placées à l'arrière plan. La notion de fenêtre active implique que les événements en provenance du clavier ou de la souris lui sont destinés.

4. Les icônes

Les objets sont représentés par des icônes (petits dessins) plutôt que par des libellés. L'usage de ces icônes permet de rapprocher l'environnement de travail de l'environnement réel de l'utilisateur. Ces icônes vont permettre de transmettre un message à l'utilisateur par l'intermédiaire d'une métaphore. Par exemple l'icône symbolisant une imprimante indique sans ambiguïté une fonction d'impression.

5. Les menus

L'utilisation de menus permet de proposer un processus basé sur l'association objet/action. On sélectionne un objet, puis on désigne l'action que l'on souhaite associer à cet objet dans un menu. L'avantage essentiel de ce processus réside dans la possibilité de combiner plusieurs actions sans avoir à redéfinir l'objet. Ce processus est le mode d'interaction avec l'interface et se divise en trois phases :

- Désignation de l'objet ;
- Choix de l'action à réaliser ;
- Production du résultat.

B. Applications SDI et MDI

1. SDI

Le SDI (Single Document Interface) est un mode de fonctionnement autorisant le chargement d'un seul document à la fois. Pour travailler sur nouveau document, l'application doit au préalable fermer celui actuellement ouvert. Le principe des applications SDI est exploité dans le Bloc-notes de Windows.

2. MDI

Le MDI (Multiple Document Interface) autorise l'ouverture de plusieurs documents à l'intérieur d'une même application. Il définit le comportement des fenêtres documents à l'intérieur d'une fenêtre mère ou fenêtre de l'application. Les meilleurs exemples de l'utilisation de fenêtres MDI se trouvent dans les outils bureautiques Word et Excel. L'intérêt du principe MDI est suffisamment manifeste pour l'utiliser également dans des applications de gestion. Il se justifie alors dans l'ouverture simultanée de plusieurs vues du système d'informations de l'entreprise. D'un point de vue technique, il permet de limiter les accès au réseau en stockant temporairement sur le poste client les données.

Développement d'applications .NET

Partie II : WinForms

Pour créer un formulaire MDI parent au moment du design, Dans la fenêtre **Propriétés**, affectez **true** à la propriété **IsMDIContainer**. Ce faisant, vous désignez le formulaire comme le conteneur MDI des fenêtres enfants.

Les formulaires MDI enfants représentent un élément essentiel des applications de type MDI car ils constituent le centre de l'interaction utilisateur.

Pour designer un formulaire comme MDI enfant :

```

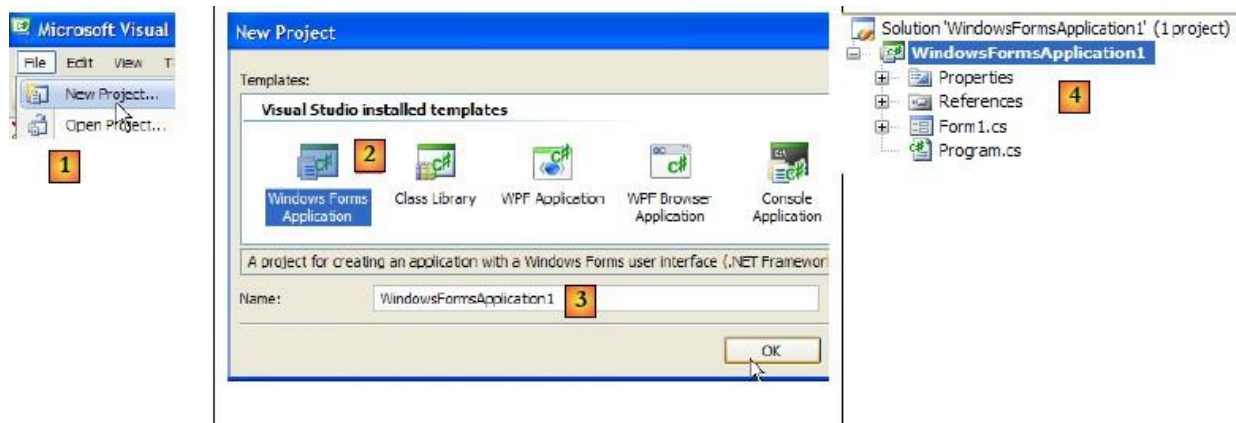
Form1 f1= new Form1();
f1.MdiParent = this;
f1.Show();
  
```

VII. Les bases des interfaces graphiques

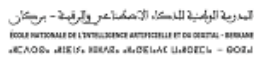
Nous nous proposons ici de donner les premiers éléments pour construire des interfaces graphiques et gérer leurs événements

A. Un premier projet

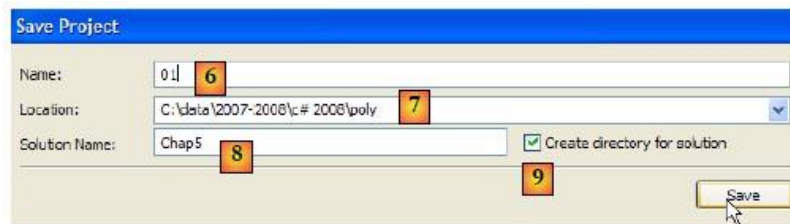
Construisons un premier projet de type "Application windows" :



- [1] : créer un nouveau projet
- [2] : de type Application Windows
- [3] : le nom du projet importe peu pour le moment
- [4] : le projet créé



Partie II : WinForms



[9] : un dossier sera créé pour la solution [Chap5]. Les projets de celle-ci seront dans des sous-dossiers.



[12] : l'application générée peut être exécutée par (Ctrl-F5). La fenêtre [Form1] s'affiche. On peut la déplacer, la redimensionner et la fermer. On a donc les éléments de base d'une fenêtre graphique.

Page 18/67



Développement d'applications .NET

Partie II : WinForms

```
1. using System;
2. using System.Windows.Forms;
3.
4. namespace Chap5 {
5.     static class Program {
6.         /// <summary>
7.         /// The main entry point for the application.
8.         /// </summary>
9.         [STAThread]
10.        static void Main() {
11.            Application.EnableVisualStyles();
12.            Application.SetCompatibleTextRenderingDefault(false);
13.            Application.Run(new Form1());
14.        }
15.    }
16. }
```

- ligne 2 : les applications avec formulaires utilisent l'espace de noms System.Windows.Forms.
- ligne 4 : l'espace de noms initial a été renommé en Chap5.
- ligne 10 : à l'exécution du projet (Ctrl-F5), la méthode [Main] est exécutée.
- lignes 11-13 : la classe Application appartient à l'espace de noms System.Windows.Forms. Elle contient des méthodes statiques pour lancer / arrêter les applications graphiques windows.
- ligne 11 : facultative - permet de donner différents styles visuels aux contrôles déposés sur un formulaire
- ligne 12 : facultative - fixe le moteur de rendu des textes des contrôles : GDI+ (true), GDI (false)
- ligne 13 : la seule ligne indispensable de la méthode [Main] : instancie la classe [Form1] qui est la classe du formulaire et lui demande de s'exécuter.

Le fichier source [Form1.cs] est le suivant :

```
1. using System;
2. using System.Windows.Forms;
3.
4. namespace Chap5 {
5.     public partial class Form1 : Form {
6.         public Form1() {
7.             InitializeComponent();
8.         }
9.     }
10. }
```

- ligne 5 : la classe Form1 dérive de la classe [System.Windows.Forms.Form] qui est la classe mère de toutes les fenêtres. Le mot clé partial indique que la classe est partielle et qu'elle peut être complétée par d'autres fichiers source. C'est le cas ici, où la classe Form1 est répartie dans deux fichiers :
 - [Form1.cs] : dans lequel on trouvera le comportement du formulaire, notamment ses gestionnaires d'événements
 - [Form1.Designer.cs] : dans lequel on trouvera les composants du formulaire et leurs propriétés. Ce fichier a la particularité d'être régénéré à chaque fois que l'utilisateur modifie la fenêtre en mode [conception].
- lignes 6-8 : le constructeur de la classe Form1



Développement d'applications .NET

Partie II : WinForms

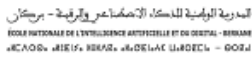
- ligne 7 : fait appel à la méthode InitializeComponent. On voit que cette méthode n'est pas présente dans [Form1.cs]. On la trouve dans [Form1.Designer.cs].

Le fichier source [Form1.Designer.cs] est le suivant :

```
1. namespace Chap5 {
2.     partial class Form1 {
3.         /// <summary>
4.         /// Required designer variable.
5.         /// </summary>
6.         private System.ComponentModel.IContainer components = null;
7.
8.         /// <summary>
9.         /// Clean up any resources being used.
10.        /// </summary>
11.        /// <param name="disposing">true if managed resources should be disposed; otherwise,
12.        false.</param>
13.        protected override void Dispose(bool disposing) {
14.            if (disposing && (components != null)) {
15.                components.Dispose();
16.            }
17.            base.Dispose(disposing);
18.        }
19.        #region Windows Form Designer generated code
20.
21.        /// <summary>
22.        /// Required method for Designer support - do not modify
23.        /// the contents of this method with the code editor.
24.        /// </summary>
25.        private void InitializeComponent() {
26.            this.SuspendLayout();
27.            //
28.            // Form1
29.            //
30.            this.AutoScaleMode = new System.Drawing.SizeF(6F, 13F);
31.            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
32.            this.ClientSize = new System.Drawing.Size(196, 98);
33.            this.Name = "Form1";
34.            this.Text = "Form1";
35.            this.ResumeLayout(false);
36.
37.        }
38.
39.        #endregion
40.
41.    }
42. }
```

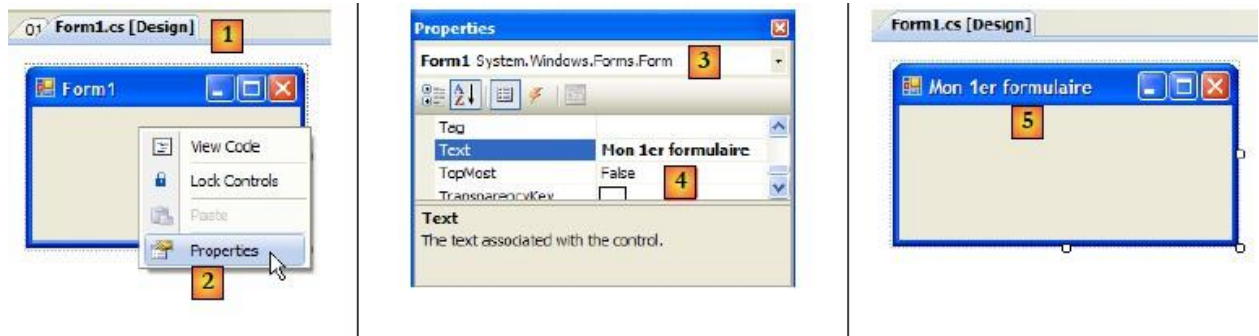
- ligne 2 : il s'agit toujours de la classe Form1. On notera qu'il n'est plus besoin de répéter qu'elle dérive de la classe Form.
- lignes 25-37 : la méthode InitializeComponent appelée par le constructeur de la classe [Form1]. Cette méthode va créer et initialiser tous les composants du formulaire. Elle est régénérée à chaque changement de celui-ci en mode [conception]. Une section, appelée région, est créée pour la délimiter lignes 19-39. Le développeur ne doit pas ajouter de code dans cette région : il sera écrasé à la régénération suivante.

Il est plus simple dans un premier temps de ne pas s'intéresser au code de [Form1.Designer.cs]. Il est généré automatiquement et est la traduction en langage C# des choix que le développeur fait en mode [conception]. Prenons un premier exemple :



Développement d'applications .NET

Partie II : WinForms



- [1] : sélectionner le mode [conception] en double-cliquant sur le fichier [Form1.cs]
- [2] : cliquer droit sur le formulaire et choisir [Propriétés]
- [3] : la fenêtre des propriétés de [Form1]
- [4] : la propriété [Text] représente le titre de la fenêtre
- [5] : le changement de la propriété [Text] est pris en compte en mode [conception] ainsi que dans le code source [Form1.Designer.cs] :

```
1.         private void InitializeComponent() {
2.             this.SuspendLayout();
3.             ...
4.             this.Text = "Mon 1er formulaire";
5.             ...
6.         }
```

B. Exemple pratique

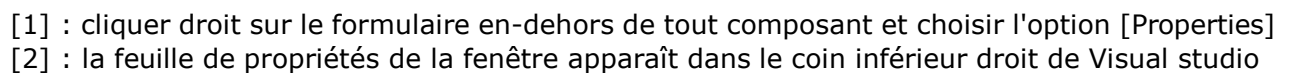
Nous commençons un nouveau projet appelé 02. Pour cela nous suivons la procédure explicitée précédemment pour créer un projet. La fenêtre à créer est la suivante :



Les composants du formulaire sont les suivants :



On pourra procéder comme suit pour construire cette fenêtre :



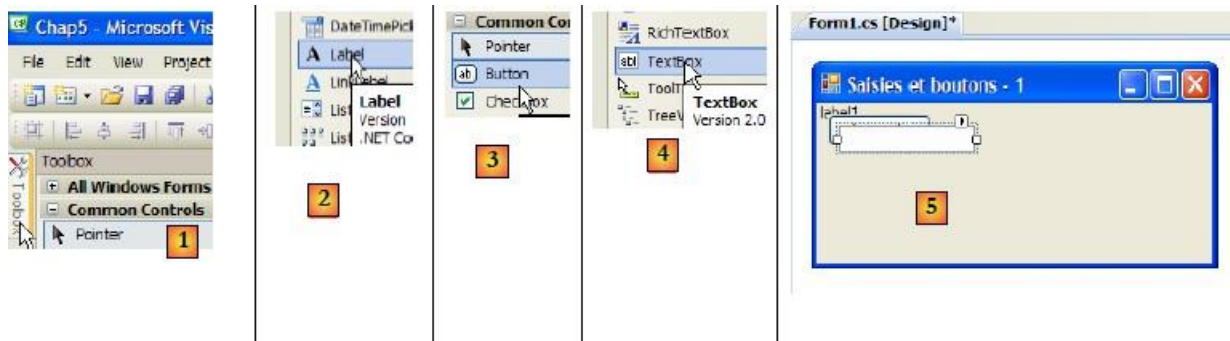
BackColor	pour fixer la couleur de fond de la fenêtre
ForeColor	pour fixer la couleur des dessins ou du texte sur la fenêtre
Text	pour donner un titre à la fenêtre
FormBorderStyle	pour fixer le type de fenêtre
Font	pour fixer la police de caractères des écritures dans la fenêtre
Name	pour fixer le nom de la fenêtre

Text	Saisies et boutons - 1
Name	frmSaisiesBoutons



Développement d'applications .NET

Partie II : WinForms

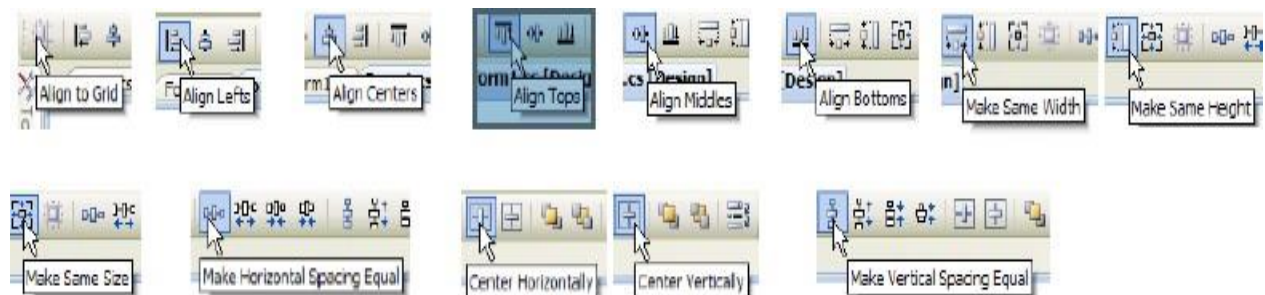
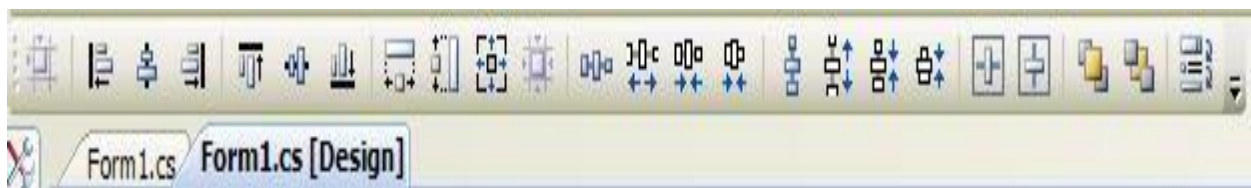


[1] : choisir la boîte à outils [Common Controls] parmi les boîtes à outils proposées par Visual Studio

[2, 3, 4] : double-cliquer successivement sur les composants [Label], [Button] et [TextBox]

[5] : les trois composants sont sur le formulaire

Pour aligner et dimensionner correctement les composants, on peut utiliser les éléments de la barre d'outils :



Le principe du formatage est le suivant :

1. sélectionnez les différents composants à formater ensemble (touche Ctrl appuyée pendant les différents clics sélectionnant les composants).
2. sélectionnez le type de formatage désiré :

- les options Align permettent d'aligner des composants par le haut, le bas, le côté gauche ou droit, le milieu.
- les options Make Same Size permettent que des composants aient la même hauteur ou la même largeur.
- l'option Horizontal Spacing permet d'aligner horizontalement des composants avec des intervalles entre eux de même largeur. Idem pour l'option Vertical Spacing pour aligner verticalement.

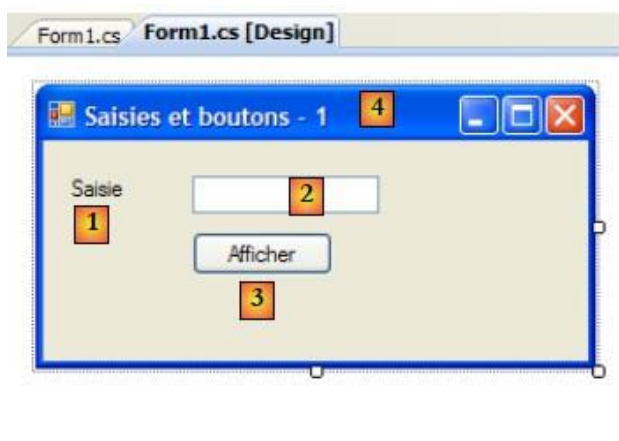


Développement d'applications .NET

Partie II : WinForms

- l'option Center permet de centrer un composant horizontalement (Horizontally) ou verticalement (Vertically) dans la fenêtre

Une fois placés les composants nous fixons leurs propriétés. Pour cela, cliquer droit sur le composant et prendre l'option Propriétés :



- [1] : sélectionner le composant pour avoir sa fenêtre de propriétés. Dans celle-ci, modifier les propriétés suivantes : **name** : *labelSaisie*, **text** : *Saisie*
- [2] : procéder de même : **name** : *textBoxSaisie*, **text** : ne rien mettre
- [3] : **name** : *buttonAfficher*, **text** : *Afficher*
- [4] : la fenêtre elle-même : **name** : *frmSaisiesBoutons*, **text** : *Saisies et boutons - 1*
- [5] : exécuter (Ctrl-F5) le projet pour avoir un premier aperçu de la fenêtre en action.

Ce qui a été fait en mode [conception] a été traduit dans le code de [Form1.Designer.cs] :

```

1. namespace Chaps {
2.     partial class frmSaisiesBoutons {
3.         ...
4.         private System.ComponentModel.IContainer components = null;
5.         ...
6.         private void InitializeComponent() {
7.             this.labelSaisie = new System.Windows.Forms.Label();
8.             this.buttonAfficher = new System.Windows.Forms.Button();
9.             this.textBoxSaisie = new System.Windows.Forms.TextBox();
10.            this.SuspendLayout();
11.            //
12.            // labelSaisie
13.            //
14.            this.labelSaisie.AutoSize = true;
15.            this.labelSaisie.Location = new System.Drawing.Point(12, 19);
16.            this.labelSaisie.Name = "labelSaisie";

```



Développement d'applications .NET

Partie II : WinForms

```
17.         this.labelSaisie.Size = new System.Drawing.Size(35, 13);
18.         this.labelSaisie.TabIndex = 0;
19.         this.labelSaisie.Text = "Saisie";
20.         //
21.         // buttonAfficher
22.         //
23.         this.buttonAfficher.Location = new System.Drawing.Point(80, 49);
24.         this.buttonAfficher.Name = "buttonAfficher";
25.         this.buttonAfficher.Size = new System.Drawing.Size(75, 23);
26.         this.buttonAfficher.TabIndex = 1;
27.         this.buttonAfficher.Text = "Afficher";
28.         this.buttonAfficher.UseVisualStyleBackColor = true;
29.         this.buttonAfficher.Click += new System.EventHandler(this.buttonAfficher_Click);
30.         //
31.         // textBoxSaisie
32.         //
33.         this.textBoxSaisie.Location = new System.Drawing.Point(80, 19);
34.         this.textBoxSaisie.Name = "textBoxSaisie";
35.         this.textBoxSaisie.Size = new System.Drawing.Size(100, 20);
36.         this.textBoxSaisie.TabIndex = 2;
37.         //
38.         // frmSaisiesBoutons
39.         //
40.         this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
41.         this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
42.         this.ClientSize = new System.Drawing.Size(292, 118);
43.         this.Controls.Add(this.textBoxSaisie);
44.         this.Controls.Add(this.buttonAfficher);
45.         this.Controls.Add(this.labelSaisie);
46.         this.Name = "frmSaisiesBoutons";
47.         this.Text = "Saisies et boutons - 1";
48.         this.ResumeLayout(false);
49.         this.PerformLayout();
50.
51.     }
52.
53.     private System.Windows.Forms.Label labelSaisie;
54.     private System.Windows.Forms.Button buttonAfficher;
55.     private System.Windows.Forms.TextBox textBoxSaisie;
56.
57. }
58. }
```

- lignes 53-55 : les trois composants ont donné naissance à trois champs privés de la classe [Form1]. On notera que les noms de ces champs sont les noms donnés aux composants en mode [conception]. C'est le cas également du formulaire.
- ligne 2 qui est la classe elle-même.
- lignes 7-9 : les trois objets de type [Label], [TextBox] et [Button] sont créés. C'est à travers eux que les composants visuels sont gérés.
- lignes 14-19 : configuration du label *labelSaisie*
- lignes 23-29 : configuration du bouton *buttonAfficher*
- lignes 33-36 : configuration du champ de saisie *textBoxSaisie*
- lignes 40-47 : configuration du formulaire *frmSaisiesBoutons*. On notera, lignes 43-45, la façon d'ajouter des composants au formulaire.

Ce code est compréhensible. Il est ainsi possible de construire des formulaires par code sans utiliser le mode [conception]. De nombreux exemples de ceci sont donnés dans la documentation MSDN de Visual Studio. Maîtriser ce code permet de créer des formulaires en cours d'exécution : par exemple, créer à la volée un formulaire permettant la mise à jour d'une table de base de données, la structure de cette table n'étant découverte qu'à l'exécution.



Développement d'applications .NET

Partie II : WinForms

Il nous reste à écrire la procédure de gestion d'un clic sur le bouton *Afficher*. Sélectionner le bouton pour avoir accès à sa fenêtre de propriétés. Celle-ci a plusieurs onglets :

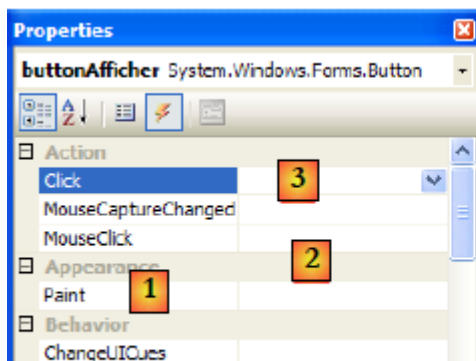


- [1] : liste des propriétés par ordre alphabétique
- [2] : événements liés au contrôle

Les propriétés et événements d'un contrôle sont accessibles par catégories ou par ordre alphabétique :

- [3] : Propriétés ou événements par catégorie
- [4] : Propriétés ou événements par ordre alphabétique

L'onglet *Events* en mode *Catégories* pour le bouton *buttonAfficher* est le suivant :



- [1] : la colonne de gauche de la fenêtre liste les événements possibles sur le bouton. Un clic sur un bouton correspond à l'événement *Click*.
- [2] : la colonne de droite contient le nom de la procédure appelée lorsque l'événement correspondant se produit.
- [3] : si on double-clique sur la cellule de l'événement *Click*, on passe alors automatiquement dans la fenêtre de code pour écrire le gestionnaire de l'événement *Click* sur le bouton *buttonAfficher* :



Développement d'applications .NET

Partie II : WinForms

```
1. using System;
2. using System.Windows.Forms;
3.
4. namespace Chap5 {
5.     public partial class frmSaisiesBoutons : Form {
6.         public frmSaisiesBoutons() {
7.             InitializeComponent();
8.         }
9.
10.        private void buttonAfficher_Click(object sender, EventArgs e) {
11.
12.        }
13.    }
14. }
```

Lignes 10-12 : le squelette du gestionnaire de l'événement *Click* sur le bouton nommé *buttonAfficher*. On notera les points suivants :

- la méthode est nommée selon le schéma **nomDuComposant_NomEvenement**
- la méthode est privée. Elle reçoit deux paramètres :
- sender* : est l'objet qui a provoqué l'événement. Si la procédure est exécutée à la suite d'un clic sur le bouton *buttonAfficher*, *sender* sera égal à *buttonAfficher*. On peut imaginer que la procédure *buttonAfficher_Click* soit exécutée à partir d'une autre procédure. Celle-ci aurait alors tout loisir de mettre comme premier paramètre, l'objet *sender* de son choix.
- EventArgs* : un objet qui contient des informations sur l'événement. Pour un événement *Click*, il ne contient rien.

Pour un événement ayant trait aux déplacements de la souris, on y trouvera les coordonnées (X,Y) de la souris. Nous n'utiliserons aucun de ces paramètres ici.

Ecrire un gestionnaire d'événement consiste à compléter le squelette de code précédent. Ici, nous voulons présenter une boîte de dialogue avec dedans, le contenu du champ *textBoxSaisie* s'il est non vide [1], un message d'erreur sinon [2] :



Développement d'applications .NET

Partie II : WinForms

Le code réalisant cela pourrait-être le suivant :

```

1.     private void buttonAfficher_Click(object sender, EventArgs e) {
2.         // on affiche le texte qui a été saisi dans le TextBox textboxSaisie
3.         string texte = textboxSaisie.Text.Trim();
4.         if (texte.Length != 0) {
5.             MessageBox.Show("Texte saisi= " + texte, "Vérification de la saisie",
6.                 MessageBoxButtons.OK, MessageBoxIcon.Information);
7.         } else {
8.             MessageBox.Show("Saissez un texte...", "Vérification de la saisie",
9.                 MessageBoxButtons.OK, MessageBoxIcon.Error);
10.        }
    
```

La classe MessageBox sert à afficher des messages dans une fenêtre. Nous avons utilisé ici la méthode Show suivante :



```

public static DialogResult Show(string text, string caption, MessageBoxButtons buttons, MessageBoxIcon icon);
    
```

Avec :

text	le message à afficher
caption	le titre de la fenêtre
buttons	les boutons présents dans la fenêtre
icon	l'icone présente dans la fenêtre


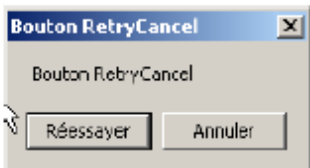
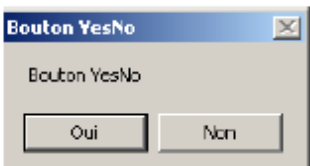
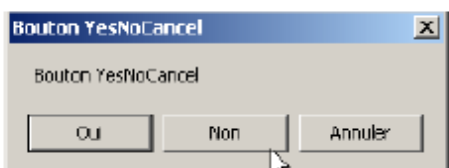
Le paramètre *buttons* peut prendre ses valeurs parmi les constantes suivantes (préfixées par *MessageBoxButtons* comme montré ligne 7) ci-dessus :

constante	boutons
AbortRetryIgnore	
OK	
OKCancel	



Développement d'applications .NET

Partie II : WinForms

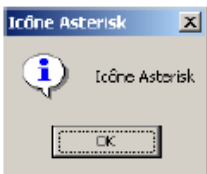


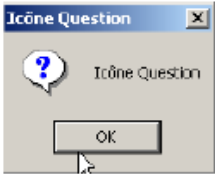

constante	boutons
	
RetryCancel	
YesNo	
YesNoCancel	



Développement d'applications .NET

Partie II : WinForms

Le paramètre *icon* peut prendre ses valeurs parmi les constantes suivantes (préfixées par *MessageBoxIcon* comme montré ligne 7) ci-dessus :








Asterisk		Error	idem Stop
Exclamation	idem Warning	Hand	
Information	idem Asterisk	None	
Question		Stop	idem Hand
Warning			
			

Développement d'applications .NET

Partie II : WinForms

La méthode *Show* est une méthode statique qui rend un résultat de type [System.Windows.Forms.DialogResult] qui est une énumération :

Members

	Member name	Description
	Abort	The dialog box return value is Abort (usually sent from a button labeled Abort).
	Cancel	The dialog box return value is Cancel (usually sent from a button labeled Cancel).
	Ignore	The dialog box return value is Ignore (usually sent from a button labeled Ignore).
	No	The dialog box return value is No (usually sent from a button labeled No).
	None	Nothing is returned from the dialog box. This means that the modal dialog continues running.
	OK	The dialog box return value is OK (usually sent from a button labeled OK).
	Retry	The dialog box return value is Retry (usually sent from a button labeled Retry).
	Yes	The dialog box return value is Yes (usually sent from a button labeled Yes).

Pour savoir sur quel bouton a appuyé l'utilisateur pour fermer la fenêtre de type *MessageBox* on écrira :

```
1. DialogResult res=MessageBox.Show(..);
2. if (res==DialogResult.Yes){ // il a appuyé sur le bouton oui...}
```

C. Les composants de base

Nous présentons maintenant diverses applications mettant en jeu les composants les plus courants afin de découvrir les principales méthodes et propriétés de ceux-ci. Pour chaque application, nous présentons l'interface graphique et le code intéressant, principalement celui des gestionnaires d'événements.

1. Les propriétés communes des objets

Celles héritées de la Classe 'Control' qu'il faut connaître:

a) Name

Il s'agit du nom de l'objet tel qu'il est géré par l'application. Par défaut, C# baptise tous les objets que vous créez de noms génériques, comme Form1, Form2, Form3 pour les fenêtres, List1, List2 pour les listes...

Il est vivement conseillé, avant toute autre chose, de rebaptiser les objets que vous venez de créer afin de donner des noms plus évocateurs. Le bouton sur lequel est écrit « OK » sera nommé **BoutonOK**. La liste qui affiche les utilisateurs sera nommée **ListUtilisateurs**. Il est conseillé de débiter le nom de l'objet par un mot évoquant sa nature: **BoutonOk** ou **BtOk** ou **ButtonOk**, **btnOk** c'est comme vous voulez.

Microsoft conseille:

Btn : pour les Boutons

Lst : pour les ListBox
 chk : pour les CheckBox
 cbo : pour les combos
 dlg : pour les DialogBox
 frm : pour les Form
 lbl : pour les labels
 txt : pour les Textbox
 tb : pour les Toolbar
 rb : pour les radiobutton
 mm : pour les menus
 tmr : pour les timers

b) Text

Il s'agit du texte qui est associé à l'objet. Dans le cas d'une fenêtre c'est le texte qui apparaît dans la barre de titre en haut. Pour un TextBox ou un Label c'est évidemment le texte qui est affiché. On peut modifier cette propriété en mode conception ou dans le code

Exemple : Avec du code comment faire pour que le bouton ButtonOk porte l'inscription 'Ok'

```
ButtonOk.Text = "OK";
```

c) Enabled

Accessible, Indique si un contrôle peut répondre à une interaction utilisateur. La propriété Enabled permet l'activation ou la désactivation des contrôles au moment de l'exécution.

Exemple : désactiver le ButtonOk

```
ButtonOk.Enabled = false;
```

d) Visible

Indique si un contrôle est visible ou non. ButtonOk.Visible=False ; fait disparaître le bouton. Attention pour rendre visible une fenêtre on utilise la méthode .Show.

e) Font

Permet le choix de la police de caractères affichée dans l'objet.

Exemple : ButtonOk.Font = new Font("Arial", 14);

f) BackColor ForeColor

Couleur du fond, Couleur de l'avant plan Pour un bouton Forecolor correspond au cadre et aux caractères.

Exemple :

```
ButtonOk.BackColor = Color.Blue;  
ButtonOk.ForeColor = Color.White;
```

2. Formulaire Form

Nous commençons par présenter le composant indispensable, le formulaire sur lequel on dépose des composants. Nous avons déjà présenté quelques-unes de ses propriétés de base.

Une fenêtre possède des propriétés qui peuvent être modifiées en mode design dans la fenêtre 'Propriétés' à droite ou par du code :

a) Name

Nom du formulaire. Donner un nom explicite. **FrmDemarrage** Dès qu'une fenêtre est créée on modifie immédiatement ses propriétés en mode conception pour lui donner l'aspect que l'on désire.

b) Text

C'est le texte qui apparaîtra dans la barre de titre en haut. Text peut être modifié par le code : `Form1.Text = "Fenêtre" ;`

c) Icon

Propriété qui permet d'associer à la Form un fichier icône. Cette icône s'affiche dans la barre de titre, tout en haut à gauche. Si la Form est la Form par défaut du projet, c'est également cette icône qui Symbolisera votre application dans Windows.

d) WindowState

Donne l'état de la fenêtre :

Plein écran : **FormWindowState.Maximized**

Normale : **FormWindowState.Normal**

Dans la barre de tâche : **FormWindowState.Minimized**

Exemple : mettre une fenêtre en plein écran avec du code

```
this.WindowState = FormWindowState.Maximized ;
```

e) ControlBox

Si cette propriété à comme valeur False, les boutons de contrôle situés à droite de la barre de la fenêtre n'apparaissent pas.

f) MaximizeBox

Si cette propriété à comme valeur False, le boutons de contrôle 'Plein écran' situés à droite de la barre de la fenêtre n'apparaît pas.

g) MinimizeBox

Si cette propriété à comme valeur False, le boutons de contrôle 'Minimize' situés à droite de la barre de la fenêtre n'apparaît pas.

h) FormBorderStyle

Permet de choisir le type des bords de la fenêtre : sans bord (None), bord simple (FixedSingle) ne permettant pas à l'utilisateur de modifier la taille de la fenêtre, bord permettant la modification de la taille de la fenêtre (Sizable).

Exemple :

Développement d'applications .NET

Partie II : WinForms

```
this.FormBorderStyle = FormBorderStyle.FixedSingle;
```

i) StartPosition

Permet de choisir la position de la fenêtre lors de son ouverture. Fenêtre au centre de l'écran ? La position qui existait lors de la conception ...?

Exemple :

```
Form f1 = new Form();  
f1.Text = "titre";  
f1.StartPosition = FormStartPosition.CenterScreen ;  
f1.Show();
```

j) Opacity

Allant de 0% (0) à 100% (1), permet de créer un formulaire plus ou moins transparent. Pour 0 il est transparent, pour 1 il est totalement opaque (normal)

Exemple :

```
this.Opacity = 0.75;
```

k) Les dialogues modale et non modale

Au niveau des interfaces utilisateur on distingue deux types de dialogues :

- Les dialogues de type modal
- Les dialogues de type non modal

Une boîte de dialogue est dite modale lorsqu'elle permet d'accéder à d'autres composants graphiques de notre solution logicielle tandis qu'elle est encore affichée à l'écran. A l'inverse la boîte de dialogue non modale bloque tout accès à d'autres formulaires de la solution jusqu'à ce qu'elle soit fermée.

Pour une boîte de dialogue modale utiliser la méthode Show() et ShowDialog() pour une boîte de dialogue non modale.

Exemple:

```
Form f1 = new Form();  
f1.Text = "Fenêtre 1";  
f1.Show();  
  
Form f2 = new Form();  
f2.Text = "Fenêtre 2";  
f2.ShowDialog();
```

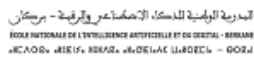
l) Formulaire d'avant plan

Pour définir au moment de la conception un formulaire en tant que formulaire d'avant-plan d'une application. Dans la fenêtre Propriétés, attribuez à la propriété TopMost la valeur true. Pour définir par code un formulaire en tant que formulaire d'avant-plan d'une application. Dans une procédure, attribuez à la propriété TopMost la valeur true : `this.TopMost=true;`

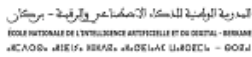
m) Les événements

Nous nous attardons ici sur quelques événements importants d'un formulaire.

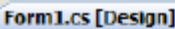
Load	le formulaire est en cours de chargement
Closing	le formulaire est en cours de fermeture
Closed	le formulaire est fermé







Partie II : WinForms



n°	type	nom	rôle
1	TextBox	textBoxSaisie	champ de saisie
2	Label	labelControle	affiche le texte de 1 en temps réel AutoSize=False, Text=(rien)
3	Button	buttonEffacer	pour effacer les champs 1 et 2
4	Button	buttonQuitter	pour quitter l'application

Le code de cette application est le suivant :

```

2.
3. namespace Chap5 {
4.     public partial class Form1 : Form {
5.         public Form1() {
6.             InitializeComponent();
7.         }
8.
9.         private void textBoxSaisie_TextChanged(object sender, System.EventArgs e) {
10.            // le contenu du TextBox a changé - on le copie dans le Label labelControle
11.            labelControle.Text = textBoxSaisie.Text;
12.        }
13.
14.        private void boutonEffacer_Click(object sender, System.EventArgs e) {
15.            // on efface le contenu de la boîte de saisie
16.            textBoxSaisie.Text = "";
17.        }
18.
19.        private void boutonQuitter_Click(object sender, System.EventArgs e) {
20.            // clic sur bouton Quitter - on quitte l'application
21.            Application.Exit();
22.        }
23.
24.        private void Form1_Shown(object sender, System.EventArgs e) {
25.            // on met le focus sur le champ de saisie
26.            textBoxSaisie.Focus();
27.        }
28.    }
29. }

```


Développement d'applications .NET

Partie II : WinForms

L'application permet de taper des lignes directement dans [1] ou d'en ajouter via [2] et [3].
Le code de l'application est le suivant :

```
1. using System.Windows.Forms;
2. using System;
3.
4. namespace Chap5 {
5.     public partial class Form1 : Form {
6.         public Form1() {
7.             InitializeComponent();
8.         }
9.
10.        private void buttonAjouter_Click(object sender, System.EventArgs e) {
11.            // ajout du contenu de textBoxLigne à celui de textBoxLignes
12.            textBoxLignes.Text += textBoxLigne.Text+Environment.NewLine;
13.            textBoxLigne.Text = "";
14.        }
15.
16.        private void Form1_Shown(object sender, EventArgs e) {
17.            // on met le focus sur le champ de saisie
18.            textBoxLigne.Focus();
19.        }
20.    }
21. }
```

- ligne 18 : lorsque le formulaire est affiché (évt *Shown*), on met le focus sur le champ de saisie *textBoxLigne*
- ligne 10 : gère le clic sur le bouton [Ajouter]
- ligne 12 : le texte du champ de saisie *textBoxLigne* est ajouté au texte du champ de saisie *textBoxLignes* suivi d'un saut de ligne.
- ligne 13 : le champ de saisie *textBoxLigne* est effacé

Parmi multiples propriétés du contrôle TextBox on peut signaler :

- **PasswordChar** : crypte le texte entré sous forme d'étoiles.

Exemple : `textBox1.PasswordChar = '*';`

- **MaxLength** : limite le nombre de caractères qu'il est possible de saisir

Exemple : `textBox1.MaxLength = 3; // Limite la saisie à 3 caractères`
`textBox1.MaxLength = 0; // Ne limite pas la saisie`

- **TextLength** : donne la longueur du texte

Exemple : `MessageBox.Show(textBox1.TextLength.ToString());`

- **AppendText** : cette méthode permet ajouter du texte au texte déjà présent dans le TextBox

Exemple : `textBox1.AppendText(textBox2.Text); textBox1.AppendText("mon texte");`

sitant aussi quelques événements liés au contrôle TextBox :

- **KeyDown** : survient quand on appuie sur la touche.
- **KeyPress** : quand la touche est enfoncée.
- **KeyUp** : quand on relâche la touche.

Ils surviennent dans cet ordre.

KeyPress permet de récupérer la touche tapée dans `e.KeyChar` (mais pas F1, F2..)

Exemple : récupérer la touche tapée

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    MessageBox.Show(e.KeyChar.ToString());
}
```

Développement d'applications .NET

Partie II : WinForms

Exemple : Ne permettre de saisir que des chiffres

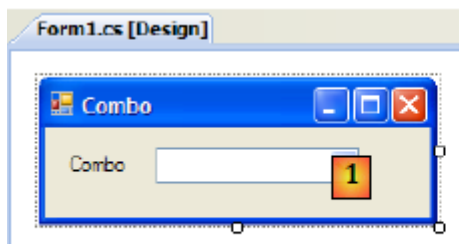
```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    int i=0;
    if (int.TryParse(e.KeyChar.ToString(), out i))
    {
        e.Handled = false; // Handled Obtient ou définit si l'événement
                           // d'exception a été géré
    }
    else
    {
        e.Handled = true ;
    }
}
```

Exemple : Compter le nombre de caractère espace

```
switch (e.KeyChar)
{
    case (char) (Keys.Space) :
        n += 1;
        break;
    ...
}
```

4. Listes deroulantes ComboBox

Nous créons le formulaire suivant :



n°	type	nom	rôle
1	ComboBox	comboNombres	contient des chaînes de caractères <i>DropDownStyle=DropDownList</i>

Un composant *ComboBox* est une liste déroulante doublée d'une zone de saisie : l'utilisateur peut soit choisir un élément dans (2) soit taper du texte dans (1). Il existe trois sortes de *ComboBox* fixées par la propriété *DropDownStyle* :



Développement d'applications .NET

Partie II : WinForms

Simple	liste non déroulante avec zone d'édition
DropDown	liste déroulante avec zone d'édition
DropDownList	liste déroulante sans zone d'édition

Par défaut, le type d'un ComboBox est *DropDown*.
La classe *ComboBox* a un seul constructeur :

<code>new ComboBox()</code>	crée un combo vide
-----------------------------	--------------------

Les éléments du ComboBox sont disponibles dans la propriété *Items* :

```
public ComboBox.ObjectCollection Items {get;}
```

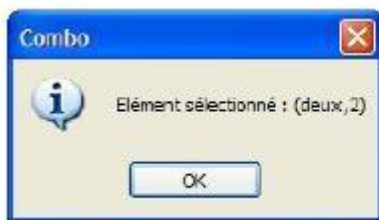
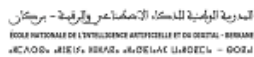
C'est une propriété indexée, *Items[i]* désignant l'élément *i* du Combo. Elle est en lecture seule.
Soit *C* un combo et *C.Items* sa liste d'éléments. On a les propriétés suivantes :

<code>C.Items.Count</code>	nombre d'éléments du combo
<code>C.Items[i]</code>	élément <i>i</i> du combo
<code>C.Add(object o)</code>	ajoute l'objet <i>o</i> en dernier élément du combo
<code>C.AddRange(object[] objets)</code>	ajoute un tableau d'objets en fin de combo
<code>C.Insert(int i, object o)</code>	ajoute l'objet <i>o</i> en position <i>i</i> du combo
<code>C.RemoveAt(int i)</code>	enlève l'élément <i>i</i> du combo
<code>C.Remove(object o)</code>	enlève l'objet <i>o</i> du combo
<code>C.Clear()</code>	supprime tous les éléments du combo
<code>C.IndexOf(object o)</code>	rend la position <i>i</i> de l'objet <i>o</i> dans le combo
<code>C.SelectedIndex</code>	index de l'élément sélectionné
<code>C.SelectedItem</code>	élément sélectionné
<code>C.SelectedItem.Text</code>	texte affiché de l'élément sélectionné
<code>C.Text</code>	texte affiché de l'élément sélectionné

On peut s'étonner qu'un combo puisse contenir des objets alors que visuellement il affiche des chaînes de caractères. Si un *ComboBox* contient un objet *obj*, il affiche la chaîne *obj.ToString()*. On se rappelle que tout objet a une méthode *ToString* héritée de la classe *object* et qui rend une chaîne de caractères "représentative" de l'objet.

L'élément *Item* sélectionné dans le combo *C* est *C.SelectedItem* ou *C.Items[C.SelectedIndex]* où *C.SelectedIndex* est le n° de l'élément sélectionné, ce n° partant de zéro pour le premier élément. Le texte sélectionné peut être obtenu de diverses façons : *C.SelectedItem.Text*, *C.Text*.

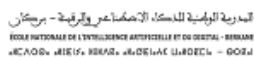
Lors du choix d'un élément dans la liste déroulante se produit l'événement *SelectedIndexChanged* qui peut être alors utilisé pour être averti du changement de sélection dans le combo. Dans l'application suivante, nous utilisons cet événement pour afficher l'élément qui a été sélectionné dans la liste.



```

3. namespace Chap5 {
4.     public partial class Form1 : Form {
5.         private int previousSelectedIndex=0;
6.
7.         public Form1() {
8.             InitializeComponent();
9.             // remplissage combo
10.            comboBoxNombres.Items.AddRange(new string[] { "zéro", "un", "deux", "trois", "quatre" });
11.            // sélection élément n° 0
12.            comboBoxNombres.SelectedIndex = 0;
13.        }
14.
15.        private void comboBoxNombres_SelectedIndexChanged(object sender, System.EventArgs e) {
16.            int newSelectedIndex = comboBoxNombres.SelectedIndex;
17.            if (newSelectedIndex != previousSelectedIndex) {
18.                // l'élément sélectionné à changé - on l'affiche
19.                MessageBox.Show(string.Format("Élément sélectionné : ({0},{1})", comboBoxNombres.Text,
newSelectedIndex), "Combo", MessageBoxButtons.OK, MessageBoxIcon.Information);
20.                // on note le nouvel index
21.                previousSelectedIndex = newSelectedIndex;
22.            }
23.        }
24.    }
25. }

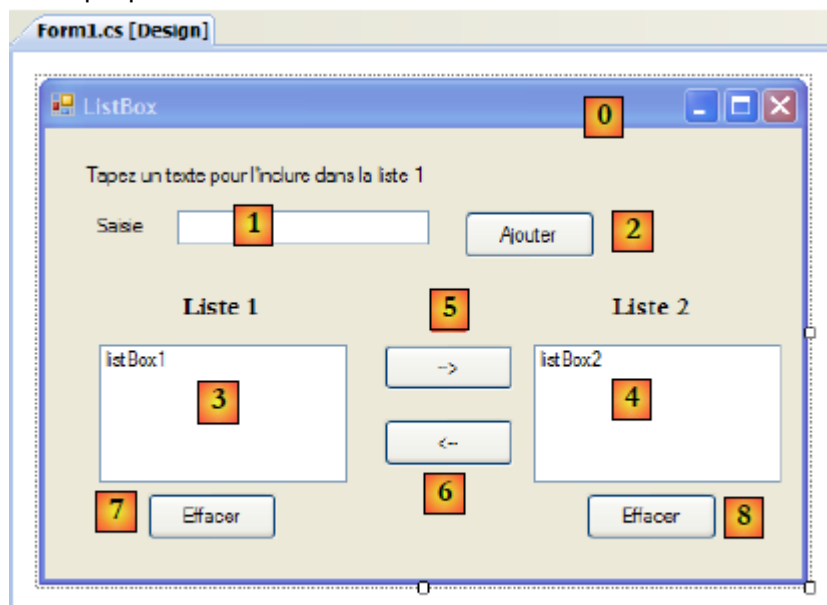
```

Partie II : WinForms

5. Composant ListBox

On se propose de construire l'interface suivante :



Les composants de cette fenêtre sont les suivants :

n°	type	nom	rôle/propriétés
0	Form	Form1	formulaire <i>FormBorderStyle=FixedSingle</i> (cadre non redimensionnable)
1	TextBox	textBoxSaisie	champ de saisie
2	Button	buttonAjouter	bouton permettant d'ajouter le contenu du champ de saisie [1] dans la liste [3]
3	ListBox	listBox1	liste 1 <i>SelectionMode=MultiExtended</i> :
4	ListBox	listBox2	liste 2 <i>SelectionMode=MultiSimple</i> :
5	Button	button1vers2	transfère les éléments sélectionnés de liste 1 vers liste 2
6	Button	button2vers1	fait l'inverse
7	Button	buttonEffacer1	vide la liste 1
8	Button	buttonEffacer2	vide la liste 2

Les composants *ListBox* ont un mode de sélection de leurs éléments qui est défini par leur propriété *SelectionMode* :

One	un seul élément peut être sélectionné
MultiExtended	multi-sélection possible : maintenir appuyée la touche SHIFT et cliquer sur un élément étend la sélection de l'élément précédemment sélectionné à l'élément courant.
MultiSimple	multi-sélection possible : un élément est sélectionné / désélectionné par un clic de souris ou par appui sur la barre d'espace.

Développement d'applications .NET

Partie II : WinForms

Fonctionnement de l'application

- L'utilisateur tape du texte dans le champ 1. Il l'ajoute à la liste 1 avec le bouton *Ajouter* (2). Le champ de saisie (1) est alors vidé et l'utilisateur peut ajouter un nouvel élément.
- Il peut transférer des éléments d'une liste à l'autre en sélectionnant l'élément à transférer dans l'une des listes et en choisissant le bouton de transfert adéquat 5 ou 6. L'élément transféré est ajouté à la fin de la liste de destination et enlevé de la liste source.
- Il peut double-cliquer sur un élément de la liste 1. Cet élément est alors transféré dans la boîte de saisie pour modification et enlevé de la liste 1.

Les boutons sont allumés ou éteints selon les règles suivantes :

- le bouton *Ajouter* n'est allumé que s'il y a un texte non vide dans le champ de saisie
- le bouton [5] de transfert de la liste 1 vers la liste 2 n'est allumé que s'il y a un élément sélectionné dans la liste 1
- le bouton [6] de transfert de la liste 2 vers la liste 1 n'est allumé que s'il y a un élément sélectionné dans la liste 2
- les boutons [7] et [8] d'effacement des listes 1 et 2 ne sont allumés que si la liste à effacer contient des éléments.

Dans les conditions précédentes, tous les boutons doivent être éteints lors du démarrage de l'application. C'est la propriété *Enabled* des boutons qu'il faut alors positionner à *false*. On peut le faire au moment de la conception ce qui aura pour effet de générer le code correspondant dans la méthode *InitializeComponent* ou le faire nous-mêmes dans le constructeur comme ci-dessous :

```
1.     public Form1() {
2.         InitializeComponent();
3.         // --- initialisations complémentaires ---
4.         // on inhibe un certain nombre de boutons
5.         buttonAjouter.Enabled = false;
6.         button1vers2.Enabled = false;
7.         button2vers1.Enabled = false;
8.         buttonEffacer1.Enabled = false;
9.         buttonEffacer2.Enabled = false;
10.    }
```

L'état du bouton *Ajouter* est contrôlé par le contenu du champ de saisie. C'est l'événement *TextChanged* qui nous permet de suivre les changements de ce contenu :

```
1.     private void textBoxSaisie_TextChanged(object sender, System.EventArgs e) {
2.         // le contenu de textBoxSaisie a changé
3.         // le bouton Ajouter n'est allumé que si la saisie est non vide
4.         buttonAjouter.Enabled = textBoxSaisie.Text.Trim() != "";
5.     }
6.
```

L'état des boutons de transfert dépend du fait qu'un élément a été sélectionné ou non dans la liste qu'ils contrôlent :



Développement d'applications .NET

Partie II : WinForms

```
1. private void listBox1_SelectedIndexChanged(object sender, System.EventArgs e) {  
2.     // un élément a été sélectionné  
3.     // on allume le bouton de transfert 1 vers 2  
4.     button1vers2.Enabled = true;  
5. }  
6.  
7. private void listBox2_SelectedIndexChanged(object sender, System.EventArgs e) {  
8.     // un élément a été sélectionné  
9.     // on allume le bouton de transfert 2 vers 1  
10.    button2vers1.Enabled = true;  
11. }
```

Le code associé au clic sur le bouton *Ajouter* est le suivant :

```
1. private void buttonAjouter_Click(object sender, System.EventArgs e) {  
2.     // ajout d'un nouvel élément à la liste 1  
3.     listBox1.Items.Add(textBoxSaisie.Text.Trim());  
4.     // raz de la saisie  
5.     textBoxSaisie.Text = "";  
6.     // Liste 1 n'est pas vide  
7.     buttonEffacer1.Enabled = true;  
8.     // retour du focus sur la boîte de saisie  
9.     textBoxSaisie.Focus();  
10. }
```

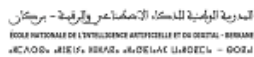
On notera la méthode *Focus* qui permet de mettre le "focus" sur un contrôle du formulaire. Le code associé au clic sur les boutons *Effacer* :

```
1. private void buttonEffacer1_Click(object sender, System.EventArgs e) {  
2.     // on efface la liste 1  
3.     listBox1.Items.Clear();  
4.     // bouton Effacer  
5.     buttonEffacer1.Enabled = false;  
6. }  
7.  
8. private void buttonEffacer2_Click(object sender, System.EventArgs e) {  
9.     // on efface la liste 2  
10.    listBox2.Items.Clear();  
11.    // bouton Effacer  
12.    buttonEffacer2.Enabled = false;  
13. }
```

Le code de transfert des éléments sélectionnés d'une liste vers l'autre :

```
1. private void button1vers2_Click(object sender, System.EventArgs e) {  
2.     // transfert de l'élément sélectionné dans Liste 1 dans Liste 2  
3.     transfert(listBox1, button1vers2, buttonEffacer1, listBox2, button2vers1, buttonEffacer2);  
4. }  
5.  
6. private void button2vers1_Click(object sender, System.EventArgs e) {  
7.     // transfert de l'élément sélectionné dans Liste 2 dans Liste 1  
8.     transfert(listBox2, button2vers1, buttonEffacer2, listBox1, button1vers2, buttonEffacer1);  
9. }  
10.
```

Les deux méthodes ci-dessus délèguent le transfert des éléments sélectionnés d'une liste à l'autre à une même méthode privée appelée **transfert** :



Partie II : WinForms

```
(a) // transfert
(b) private void transfert(ListBox l1, Button button1vers2, Button buttonEffacer1, ListBox l2,
    Button button2vers1, Button buttonEffacer2) {
(c)     // transfert dans la liste l2 des éléments sélectionnés de la liste l1
(d)     for (int i = l1.SelectedIndices.Count - 1; i >= 0; i--) {
(e)         // index de l'élément sélectionné
(f)         int index = l1.SelectedIndices[i];
(g)         // ajout dans l2
(h)         l2.Items.Add(l1.Items[index]);
(i)         // suppression dans l1
(j)         l1.Items.RemoveAt(index);
(k)     }
(l)     // boutons Effacer
(m)     buttonEffacer2.Enabled = l2.Items.Count != 0;
(n)     buttonEffacer1.Enabled = l1.Items.Count != 0;
(o)     // boutons de transfert
(p)     button1vers2.Enabled = false;
(q) }
```

Ligne b : la méthode **transfert** reçoit six paramètres :

- une référence sur la liste contenant les éléments sélectionnés appelée ici **l1**. Lors de l'exécution de l'application, l1 est soit *listBox1* soit *listBox2*. On voit des exemples d'appel, lignes 3 et 8 des procédures de transfert *buttonXversY_Click*.
- une référence sur le bouton de transfert lié à la liste **l1**. Par exemple si **l1** est *listBox2*, ce sera *button2vers1*(cf appel ligne 8)
- une référence sur le bouton d'effacement de la liste **l1**. Par exemple si **l1** est *listBox1*, ce sera *buttonEffacer1*(cf appel ligne 3)
- les trois autres références sont analogues mais font référence à la liste **l2**.

Ligne d : la collection [ListBox].**SelectedIndices** représente les indices des éléments sélectionnés dans le composant [ListBox]. C'est une collection :

- `[ListBox].SelectedIndices.Count` est le nombre d'élément de cette collection
- `[ListBox].SelectedIndices[i]` est l'élément n° i de cette collection

On parcourt la collection en sens inverse : on commence par la fin de la collection pour terminer par le début. Nous expliquerons pourquoi.

Ligne f : indice d'un élément sélectionné de la liste **l1**

Ligne h : cet élément est ajouté dans la liste **12**

Ligne j : et supprimé de la liste **l1**. Parce qu'il est supprimé, il n'est plus sélectionné. La collection **l1.SelectedIndices** de la ligne d va être recalculée. Elle va perdre l'élément qui vient d'être supprimé. Tous les éléments qui sont après celui-ci vont voir leur n° passer de n à n-1.

- si la boucle de la ligne (d) est croissante et qu'elle vient de traiter l'élément n° 0, elle va ensuite traiter l'élément n° 1. Or l'élément qui portait le n° 1 avant la suppression de l'élément n° 0, va ensuite porter le n° 0. Il sera alors oublié par la boucle.
- si la boucle de la ligne (d) est décroissante et qu'elle vient de traiter l'élément n° n, elle va ensuite traiter l'élément n° n-1. Après suppression de l'élément n° n, l'élément n° n-1 ne change pas de n°. Il est donc traité au tour de boucle suivant.

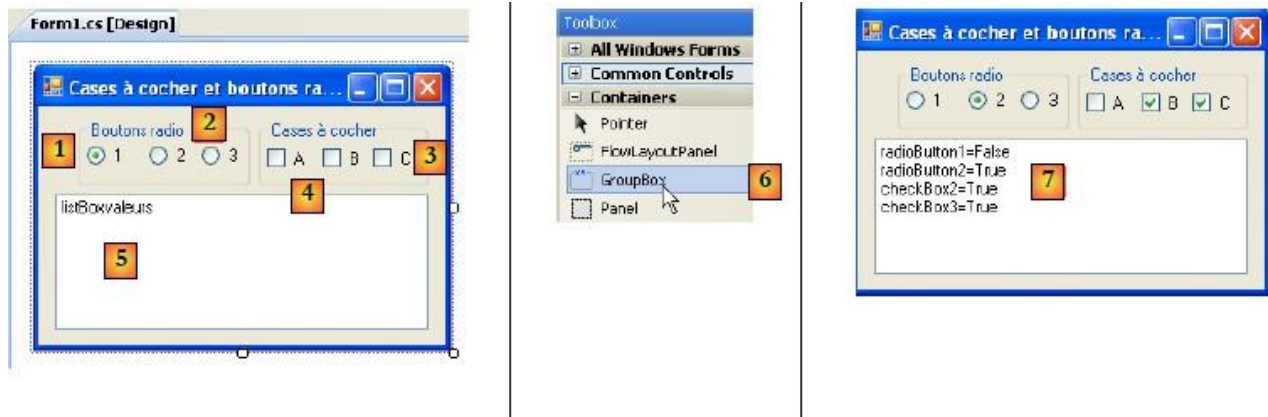
Développement d'applications .NET

Partie II : WinForms

Lignes m-n : l'état des boutons [Effacer] dépend de la présence ou non d'éléments dans les listes associées

Ligne p : la liste I2 n'a plus d'éléments sélectionnés : on éteint son bouton de transfert.

6. Cases à cocher CheckBox, boutons radio ButtonRadio



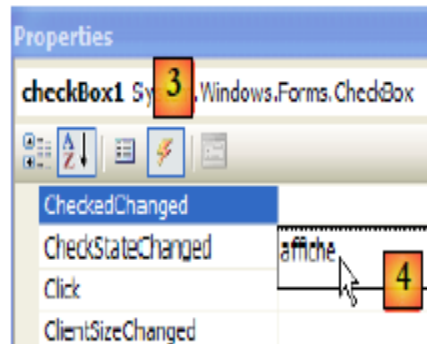
Les composants de la fenêtre sont les suivants :

n°	type	nom	rôle
1	GroupBox	groupBox1 cf [6]	un conteneur de composants. On peut y déposer d'autres composants.
2	RadioButton	radioButton1 radioButton2 radioButton3	3 boutons radio - <i>radioButton1</i> a la propriété <i>Checked</i> =True et la propriété <i>Text</i> =1 - <i>radioButton2</i> a la propriété <i>Text</i> =2 - <i>radioButton3</i> a la propriété <i>Text</i> =3 Des boutons radio présents dans un même conteneur, ici le <i>GroupBox</i> , sont exclusifs l'un de l'autre : seul l'un d'entre-eux est allumé.
3	GroupBox	groupBox2	
4	CheckBox	checkBox1 checkBox2 checkBox3	3 cases à cocher. <i>checkBox1</i> a la propriété <i>Checked</i> =True et la propriété <i>Text</i> =A - <i>checkBox2</i> a la propriété <i>Text</i> =B - <i>checkBox3</i> a la propriété <i>Text</i> =C
5	ListBox	listBoxValeurs	une liste qui affiche les valeurs des boutons radio et des cases à cocher dès qu'un changement intervient.
6			montre où trouver le conteneur <i>GroupBox</i>

L'événement qui nous intéresse pour ces six contrôles est l'événement *CheckChanged* indiquant que l'état de la case à cocher ou du bouton radio a changé. Cet état est représenté dans les deux cas par la propriété booléenne *Checked* qui à **vrai** signifie que le contrôle est coché. Nous n'utiliserons ici qu'une seule méthode pour traiter les six événements *CheckChanged*, la méthode *affiche*.

Pour faire en sorte que les six événements *CheckChanged* soient gérés par la même méthode *affiche*, on pourra procéder comme suit :

Sélectionnons le composant *radioButton1* et cliquons droit dessus pour avoir accès à ses propriétés :



```
1. private void affiche(object sender, EventArgs e) {
2.     }
```

Dans la méthode *InitializeComponent* du code a été généré. La méthode *affiche* a été déclarée comme gestionnaire des six événements *CheckedChanged* de la façon suivante :

```
1. this.radioButton1.CheckedChanged += new System.EventHandler(this.affiche);
2. this.radioButton2.CheckedChanged += new System.EventHandler(this.affiche);
3. this.radioButton3.CheckedChanged += new System.EventHandler(this.affiche);
4. this.checkBox1.CheckedChanged += new System.EventHandler(this.affiche);
5. this.checkBox2.CheckedChanged += new System.EventHandler(this.affiche);
6. this.checkBox3.CheckedChanged += new System.EventHandler(this.affiche);
```

Page 48/67



Développement d'applications .NET

Partie II : WinForms

```
1. private void affiche(object sender, System.EventArgs e) {  
2.     // affiche l'état du bouton radio ou de la case à cocher  
3.     // est-ce un checkbox ?  
4.     if (sender is CheckBox) {  
5.         CheckBox chk = (CheckBox)sender;  
6.         listBoxvaleurs.Items.Add(chk.Name + "=" + chk.Checked);  
7.     }  
8.     // est-ce un radiobutton ?  
9.     if (sender is RadioButton) {  
10.        RadioButton rdb = (RadioButton)sender;  
11.        listBoxvaleurs.Items.Add(rdb.Name + "=" + rdb.Checked);  
12.    }  
13. }
```

La syntaxe

```
if (sender is CheckBox) {
```

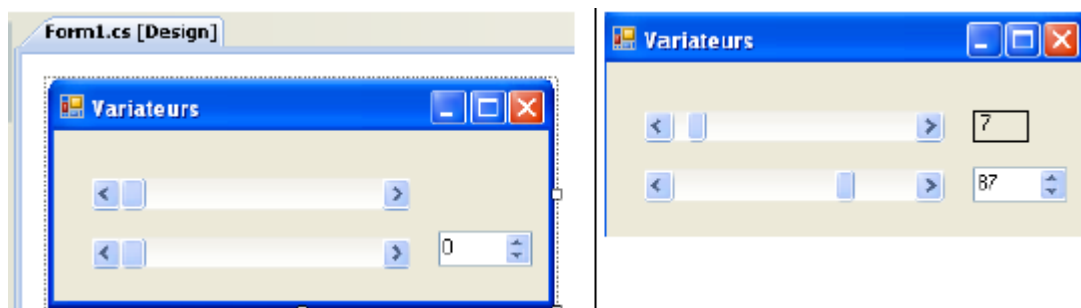
Permet de vérifier que l'objet *sender* est de type *CheckBox*. Cela nous permet ensuite de faire un transtypage vers le type exact de *sender*. La méthode *affiche* écrit dans la liste *listBoxValeurs* le nom du composant à l'origine de l'événement et la valeur de sa propriété *Checked*. A l'exécution [7], on voit qu'un clic sur un bouton radio provoque deux événements *CheckChanged* : l'un sur l'ancien bouton coché qui passe à "non coché" et l'autre sur le nouveau bouton qui passe à "coché".

7. Variateurs ScrollBar

Il existe plusieurs types de variateur : le variateur horizontal (*HScrollBar*), le variateur vertical (*VScrollBar*), l'incrémenteur (*NumericUpDown*).



Réalisons l'application suivante :





Développement d'applications .NET

Partie II : WinForms

n°	type	nom	rôle
1	hScrollBar	hScrollBar1	un variateur horizontal
2	hScrollBar	hScrollBar2	un variateur horizontal qui suit les variations du variateur 1
3	Label	labelValeurHS1	affiche la valeur du variateur horizontal
4	NumericUpDown	numericUpDown2	permet de fixer la valeur du variateur 2

- Un variateur *ScrollBar* permet à l'utilisateur de choisir une valeur dans une plage de valeurs entières symbolisée par la "bande" du variateur sur laquelle se déplace un curseur. La valeur du variateur est disponible dans sa propriété **Value**.
- Pour un variateur horizontal, l'extrémité gauche représente la valeur minimale de la plage, l'extrémité droite la valeur maximale, le curseur la valeur actuelle choisie. Pour un variateur vertical, le minimum est représenté par l'extrémité haute, le maximum par l'extrémité basse. Ces valeurs sont représentées par les propriétés **Minimum** et **Maximum** et valent par défaut 0 et 100.
- Un clic sur les extrémités du variateur fait varier la valeur d'un incrément (positif ou négatif) selon l'extrémité cliquée appelée **SmallChange** qui vaut par défaut 1.
- Un clic de part et d'autre du curseur fait varier la valeur d'un incrément (positif ou négatif) selon l'extrémité cliquée appelée **LargeChange** qui vaut par défaut 10.
- Lorsqu'on clique sur l'extrémité supérieure d'un variateur vertical, sa valeur diminue. Cela peut surprendre l'utilisateur moyen qui s'attend normalement à voir la valeur "monter". On règle ce problème en donnant une valeur négative aux propriétés *SmallChange* et *LargeChange*.
- Ces cinq propriétés (**Value**, **Minimum**, **Maximum**, **SmallChange**, **LargeChange**) sont accessibles en lecture et écriture.
- L'événement principal du variateur est celui qui signale un changement de valeur : l'événement **Scroll**.

Un composant **NumericUpDown** est proche du variateur : il a lui aussi les propriétés *Minimum*, *Maximum* et *Value*, par défaut 0, 100, 0. Mais ici, la propriété *Value* est affichée dans une boîte de saisie faisant partie intégrante du contrôle. L'utilisateur peut lui même modifier cette valeur sauf si on a mis la propriété *ReadOnly* du contrôle à vrai. La valeur de l'incrément est fixée par la propriété *Increment*, par défaut 1. L'événement principal du composant *NumericUpDown* est celui qui signale un changement de valeur : l'événement **ValueChanged**.

Le code de l'application est le suivant :



Développement d'applications .NET

Partie II : WinForms

```
1. using System.Windows.Forms;
2.
3. namespace Chap5 {
4.     public partial class Form1 : Form {
5.         public Form1() {
6.             InitializeComponent();
7.             // on fixe les caractéristiques du variateur 1
8.             hScrollBar1.Value = 7;
9.             hScrollBar1.Minimum = 1;
10.            hScrollBar1.Maximum = 130;
11.            hScrollBar1.LargeChange = 11;
12.            hScrollBar1.SmallChange = 1;
13.            // on donne au variateur 2 les mêmes caractéristiques qu'au variateur 1
14.            hScrollBar2.Value = hScrollBar1.Value;
15.            hScrollBar2.Minimum = hScrollBar1.Minimum;
16.            hScrollBar2.Maximum = hScrollBar1.Maximum;
17.            hScrollBar2.LargeChange = hScrollBar1.LargeChange;
18.            hScrollBar2.SmallChange = hScrollBar1.SmallChange;
19.            // idem pour l'incrémenteur
20.            numericUpDown2.Value = hScrollBar1.Value;
21.            numericUpDown2.Minimum = hScrollBar1.Minimum;
22.            numericUpDown2.Maximum = hScrollBar1.Maximum;
23.            numericUpDown2.Increment = hScrollBar1.SmallChange;
24.
25.            // on donne au Label la valeur du variateur 1
26.            labelValeurHS1.Text = hScrollBar1.Value.ToString();
27.        }
28.
29.        private void hScrollBar1_Scroll(object sender, ScrollEventArgs e) {
30.            // changement de valeur du variateur 1
31.            // on répercute sa valeur sur le variateur 2 et sur le label
32.            hScrollBar2.Value = hScrollBar1.Value;
33.            labelValeurHS1.Text = hScrollBar1.Value.ToString();
34.        }
35.
36.        private void numericUpDown2_ValueChanged(object sender, System.EventArgs e) {
37.            // l'incrémenteur a changé de valeur
38.            // on fixe la valeur du variateur 2
39.            hScrollBar2.Value = (int)numericUpDown2.Value;
40.        }
41.    }
42. }
```

8. Événements souris

Lorsqu'on dessine dans un conteneur, il est important de connaître la position de la souris pour, par exemple, afficher un point lors d'un clic. Les déplacements de la souris provoquent des événements dans le conteneur dans lequel elle se déplace.

1 Mouse	2 Drag Drop
MouseDown	DragDrop
MouseEnter	DragEnter
MouseHover	DragLeave
MouseLeave	DragOver
MouseMove	GiveFeedback
MouseUp	QueryContinueDrag

[1] : les événements survenant lors d'un déplacement de la souris sur le formulaire ou sur un contrôle

[2] : les événements survenant lors d'un glisser / lâcher (Drag'nDrop)

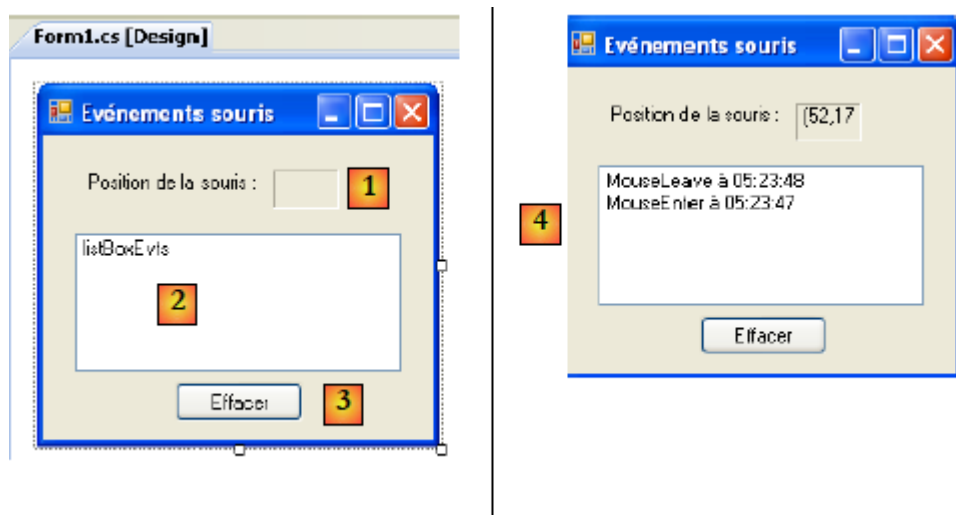


Développement d'applications .NET

Partie II : WinForms

MouseEnter	la souris vient d'entrer dans le domaine du contrôle
MouseLeave	la souris vient de quitter le domaine du contrôle
MouseMove	la souris bouge dans le domaine du contrôle
MouseDown	Pression sur le bouton gauche de la souris
MouseUp	Relâchement du bouton gauche de la souris
DragDrop	l'utilisateur lâche un objet sur le contrôle
DragEnter	l'utilisateur entre dans le domaine du contrôle en tirant un objet
DragLeave	l'utilisateur sort du domaine du contrôle en tirant un objet
DragOver	l'utilisateur passe au-dessus domaine du contrôle en tirant un objet

Voici une application permettant de mieux appréhender à quels moments se produisent les différents événements souris :



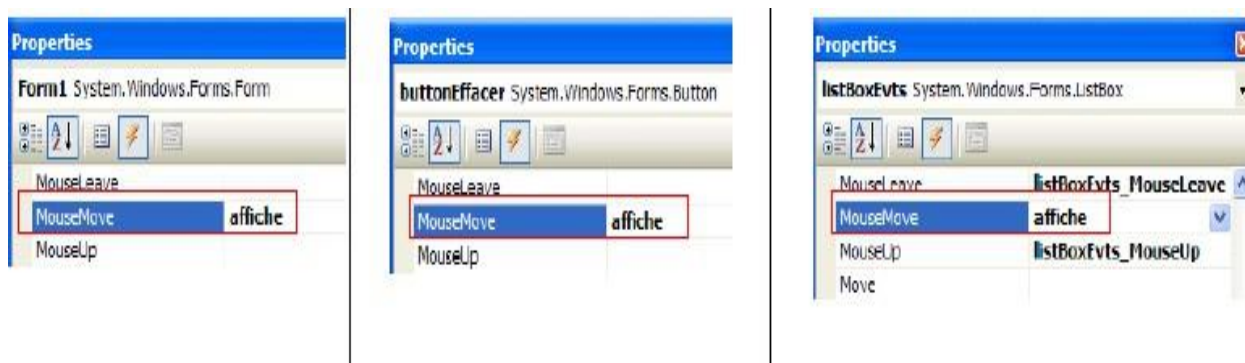
n°	type	nom	rôle
1	Label	lblPositionSouris	pour afficher la position de la souris dans le formulaire 1, la liste 2 ou le bouton 3
2	ListBox	listBoxEvts	pour afficher les évts souris autres que <i>MouseMove</i>
3	Button	buttonEffacer	pour effacer le contenu de 2

Pour suivre les déplacements de la souris sur les trois contrôles, on n'écrit qu'un seul gestionnaire, le gestionnaire *affiche* :



Développement d'applications .NET

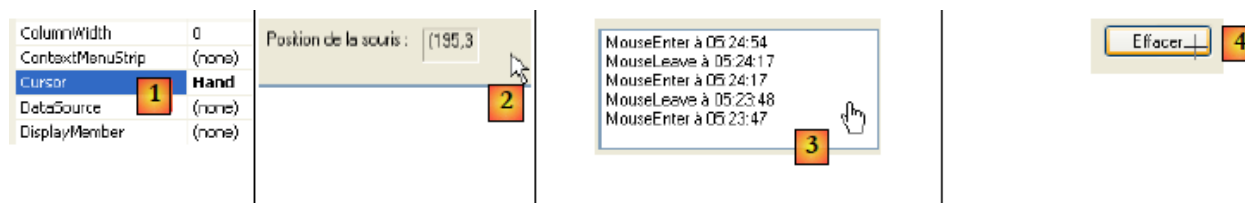
Partie II : WinForms



Le code de la procédure *affiche* est le suivant :

```
1. private void affiche(object sender, MouseEventArgs e) {  
2.     // mvt souris - on affiche les coordonnées (X,Y) de celle-ci  
3.     labelPositionSouris.Text = "(" + e.X + "," + e.Y + ")";  
4. }
```

A chaque fois que la souris entre dans le domaine d'un contrôle son système de coordonnées change. Son origine (0,0) est le coin supérieur gauche du contrôle sur lequel elle se trouve. Ainsi à l'exécution, lorsqu'on passe la souris du formulaire au bouton, on voit clairement le changement de coordonnées. Afin de mieux voir ces changements de domaine de la souris, on peut utiliser la propriété *Cursor* [1] des contrôles :



Cette propriété permet de fixer la forme du curseur de souris lorsque celle-ci entre dans le domaine du contrôle. Ainsi dans notre exemple, nous avons fixé le curseur à **Default** pour le formulaire lui-même [2], **Hand** pour la liste 2 [3] et à **Cross** pour le bouton 3 [4]. Par ailleurs, pour détecter les entrées et sorties de la souris sur la liste 2, nous traitons les événements *MouseEnter* et *MouseLeave* de cette même liste :

```
1. private void listBoxEvts_MouseEnter(object sender, System.EventArgs e) {  
2.     // on signale l'évt  
3.     listBoxEvts.Items.Insert(0, string.Format("MouseEnter à {0:hh:mm:ss}", DateTime.Now));  
4. }  
5.  
6. private void listBoxEvts_MouseLeave(object sender, EventArgs e) {  
7.     // on signale l'évt  
8.     listBoxEvts.Items.Insert(0, string.Format("MouseLeave à {0:hh:mm:ss}", DateTime.Now));  
9. }
```

Pour traiter les clics sur le formulaire, nous traitons les événements *MouseDown* et *MouseUp* :

Développement d'applications .NET

Partie II : WinForms

```

1. private void listBoxEvts_MouseDown(object sender, MouseEventArgs e) {
2.     // on signale l'évt
3.     listBoxEvts.Items.Insert(0, string.Format("MouseDown à {0:hh:mm:ss}", DateTime.Now));
4. }
5.
6. private void listBoxEvts_MouseUp(object sender, MouseEventArgs e) {
7.     // on signale l'évt
8.     listBoxEvts.Items.Insert(0, string.Format("MouseUp à {0:hh:mm:ss}", DateTime.Now));
9. }
  
```

Lignes 3 et 8 : les messages sont placés en 1ère position dans le *ListBox* afin que les événements les plus récents soient les premiers dans la liste.



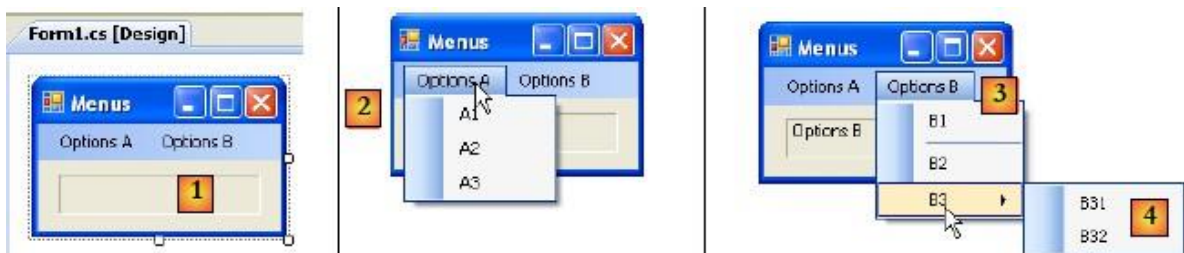
Enfin, le code du gestionnaire de clic sur le bouton *Effacer* :

```

1. private void buttonEffacer_Click(object sender, EventArgs e) {
2.     listBoxEvts.Items.Clear();
3. }
  
```

9. Créer une fenêtre avec menu

Voyons maintenant comment créer une fenêtre avec menu. Nous allons créer la fenêtre suivante :

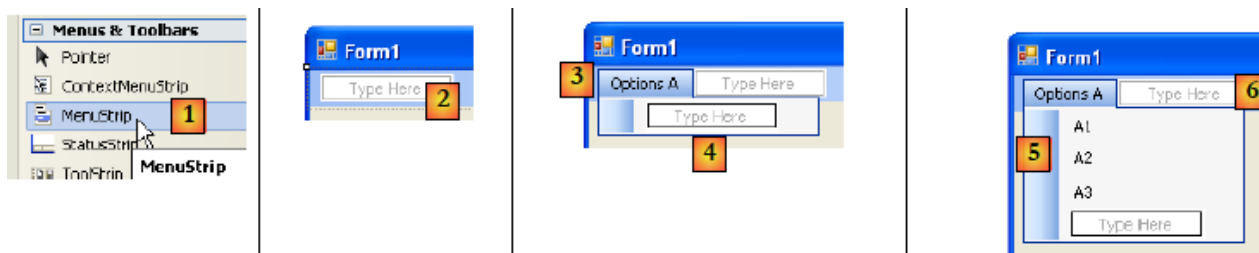


Pour créer un menu, on choisit le composant "*MenuStrip*" dans la barre "*Menus & Tollbars*" :



Développement d'applications .NET

Partie II : WinForms

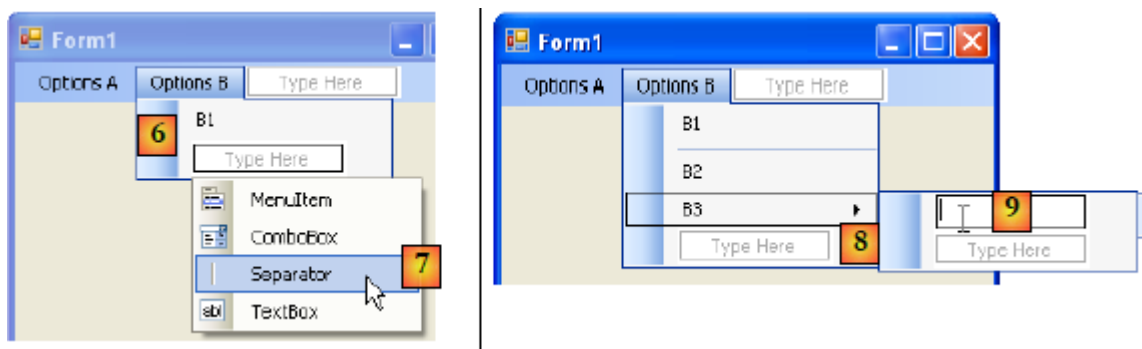


[1] : choix du composant [MenuStrip]

[2] : on a alors un menu qui s'installe sur le formulaire avec des cases vides intitulées "Type Here". Il suffit d'y indiquer les différentes options du menu.

[3] : le libellé "Options A" a été tapé. On passe au libellé [4].

[5] : les libellés des options A ont été saisis. On passe au libellé [6]

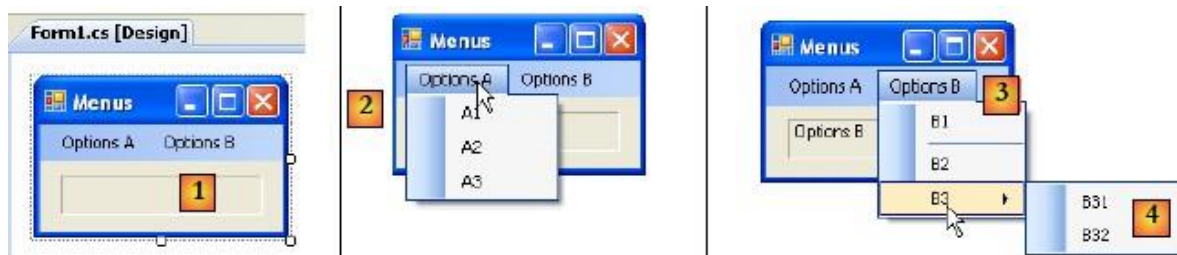


[6] : les premières options B

[7] : sous B1, on met un séparateur. Celui-ci est disponible dans un combo associé au texte "Type Here"

[8] : pour faire un sous-menu, utiliser la flèche [8] et taper le sous-menu dans [9]

Il reste à nommer les différents composants du formulaire :



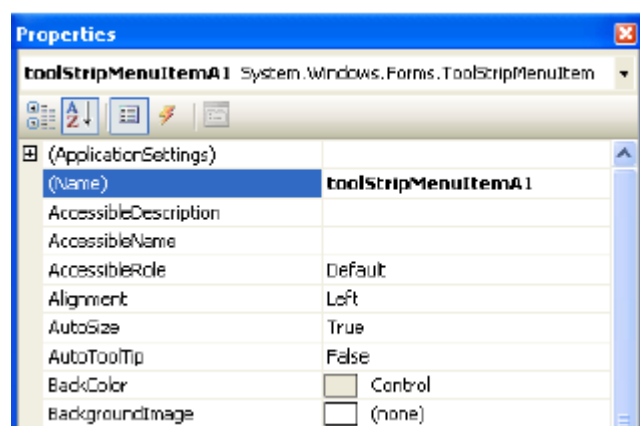


Développement d'applications .NET

Partie II : WinForms

n°	type	nom(s)	rôle
1	Label	labelStatut	pour afficher le texte de l'option de menu cliquée
2	toolStripMenuItem	toolStripMenuItemOptionsA toolStripMenuItemA1 toolStripMenuItemA2 toolStripMenuItemA3	options de menu sous l'option principale "Options A"
3	toolStripMenuItem	toolStripMenuItemOptionsB toolStripMenuItemB1 toolStripMenuItemB2 toolStripMenuItemB3	options de menu sous l'option principale "Options B"
4	toolStripMenuItem	toolStripMenuItemB31 toolStripMenuItemB32	options de menu sous l'option principale "B3"

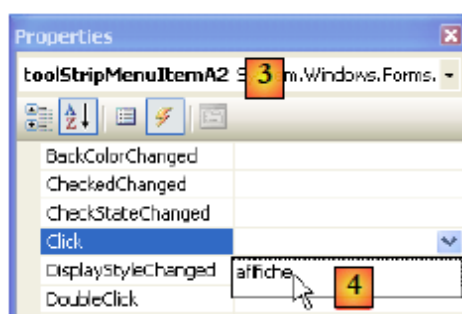
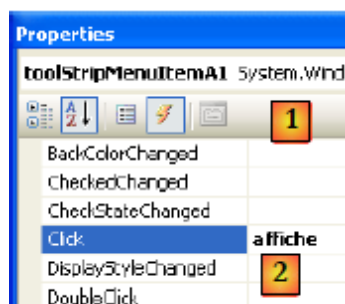
Les options de menu sont des contrôles comme les autres composants visuels et ont des propriétés et événements. Par exemple les propriétés de l'option de menu A1 sont les suivantes :



Deux propriétés sont utilisées dans notre exemple :

Name	le nom du contrôle menu
Text	le libellé de l'option de menu

Dans la structure du menu, sélectionnons l'option A1 et cliquons droit pour avoir accès aux propriétés du contrôle :



Développement d'applications .NET

Partie II : WinForms

Dans l'onglet *événements* [1], on associe la méthode *affiche* [2] à l'événement *Click*. Cela signifie que l'on souhaite que le clic sur l'option *A1* soit traitée par une méthode appelée *affiche*. Visual studio génère automatiquement la méthode *affiche* dans la fenêtre de code :

```

3. private void affiche(object sender, EventArgs e) {
4.     }
  
```

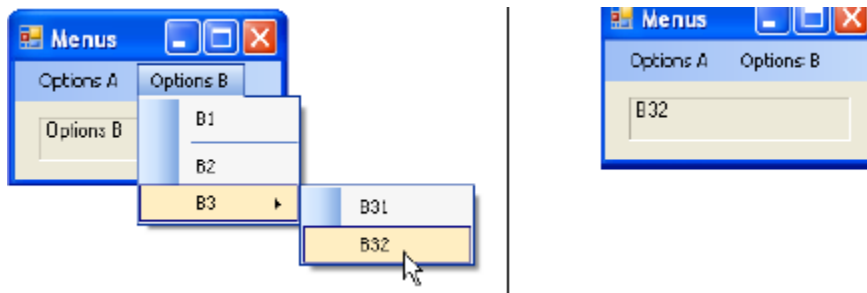
Dans cette méthode, nous nous contenterons d'afficher dans le label *labelStatut* la propriété *Text* de l'option de menu qui a été cliquée :

```

1. private void affiche(object sender, EventArgs e) {
2.     // affiche dans le TextBox le nom du sous-menu choisi
3.     labelStatut.Text = ((ToolStripMenuItem)sender).Text;
4. }
  
```

La source de l'événement *sender* est de type *object*. Les options de menu sont elle de type *ToolStripMenuItem*, aussi est-on obligé de faire un transtypage de *object* vers *ToolStripMenuItem*.

Pour toutes les options de menu, on fixe le gestionnaire du clic à la méthode *affiche* [3,4]. Exécutons l'application et sélectionnons un élément de menu :

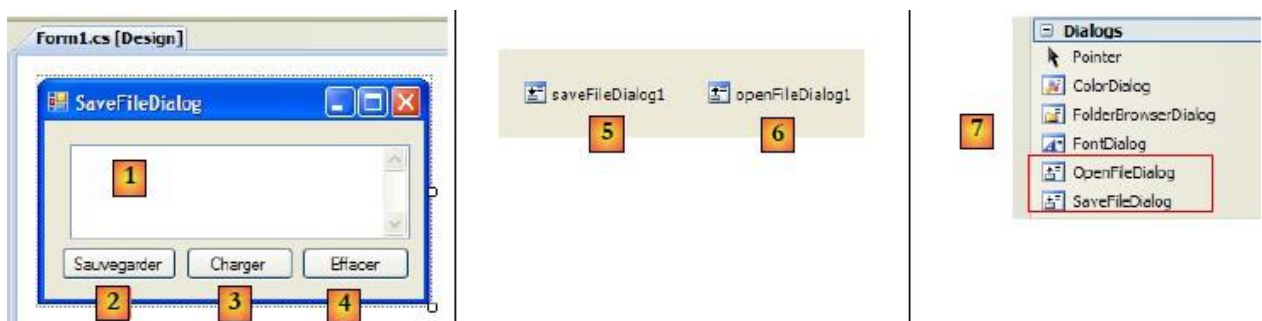


10. Composants non visuels

Nous nous intéressons maintenant à un certain nombre de composants non visuels : on les utilise lors de la conception mais on ne les voit pas lors de l'exécution.

a) Boîtes de dialogue OpenFileDialog et SaveFileDialog

Nous allons construire l'application suivante :





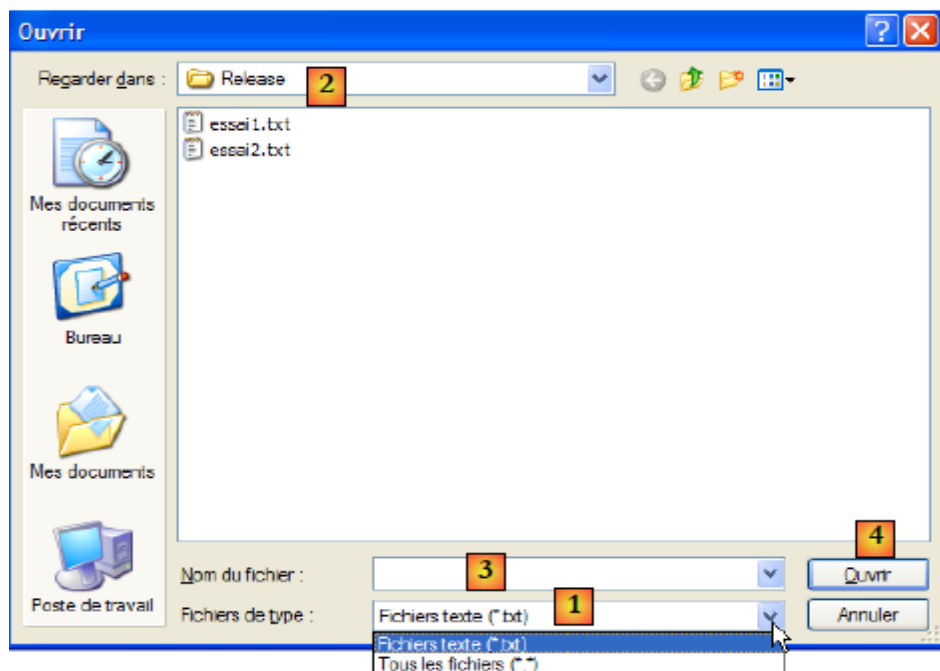


Développement d'applications .NET

Partie II : WinForms

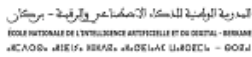
- ligne 4 : on fixe le dossier initial (*InitialDirectory*) au dossier (*Application.ExecutablePath*) qui contient l'exécutable de l'application.
- ligne 5 : on fixe les types de fichiers à présenter. On notera la syntaxe des filtres : *filtre1|filtre2|..|filtren* avec *filtrei= Texte| modèle de fichier*. Ici l'utilisateur aura le choix entre les fichiers *.txt et *.*.
- ligne 6 : on fixe le type de fichier à présenter en premier à l'utilisateur. Ici l'index 0 désigne les fichiers *.txt.
- ligne 8 : la boîte de dialogue est affichée et son résultat récupéré. Pendant que la boîte de dialogue est affichée, l'utilisateur n'a plus accès au formulaire principal (boîte de dialogue dite modale). L'utilisateur fixe le nom du fichier à sauvegarder et quitte la boîte soit par le bouton *Enregistrer*, soit par le bouton *Annuler*, soit en fermant la boîte. Le résultat de la méthode *ShowDialog* est *DialogResult.OK* uniquement si l'utilisateur a utilisé le bouton *Enregistrer* pour quitter la boîte de dialogue.
- Ceci fait, le nom du fichier à créer est maintenant dans la propriété *FileName* de l'objet *saveFileDialog1*. On est alors ramené à la création classique d'un fichier texte. On y écrit le contenu du *TextBox* : *textBoxLignes.Text* tout en gérant les exceptions qui peuvent se produire.

La classe **OpenFileDialog** est très proche de la classe *SaveFileDialog*. On utilisera les mêmes méthodes et propriétés que précédemment. La méthode *ShowDialog* affiche une boîte de dialogue analogue à la suivante :



- [1] liste déroulante construite à partir de la propriété **Filter**. Le type de fichier proposé par défaut est fixé par **FilterIndex**
- [2] dossier courant, fixé par **InitialDirectory** si cette propriété a été renseignée
- [3] nom du fichier choisi ou tapé directement par l'utilisateur. Sera disponible dans la propriété **FileName**
- [4] boutons **Ouvrir/Annuler**. Si le bouton *Ouvrir* est utilisé, la fonction *ShowDialog* rend le résultat **DialogResult.OK**

La procédure de chargement du fichier texte peut s'écrire ainsi :

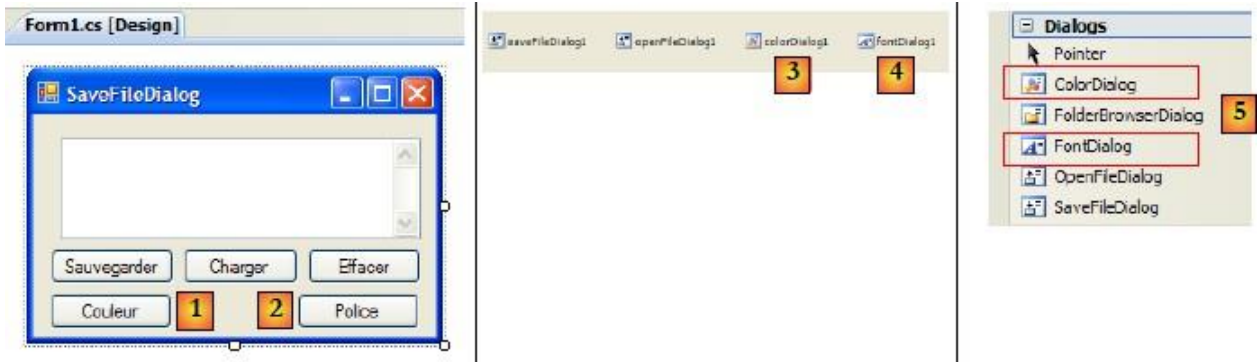


Partie II : WinForms



Développement d'applications .NET

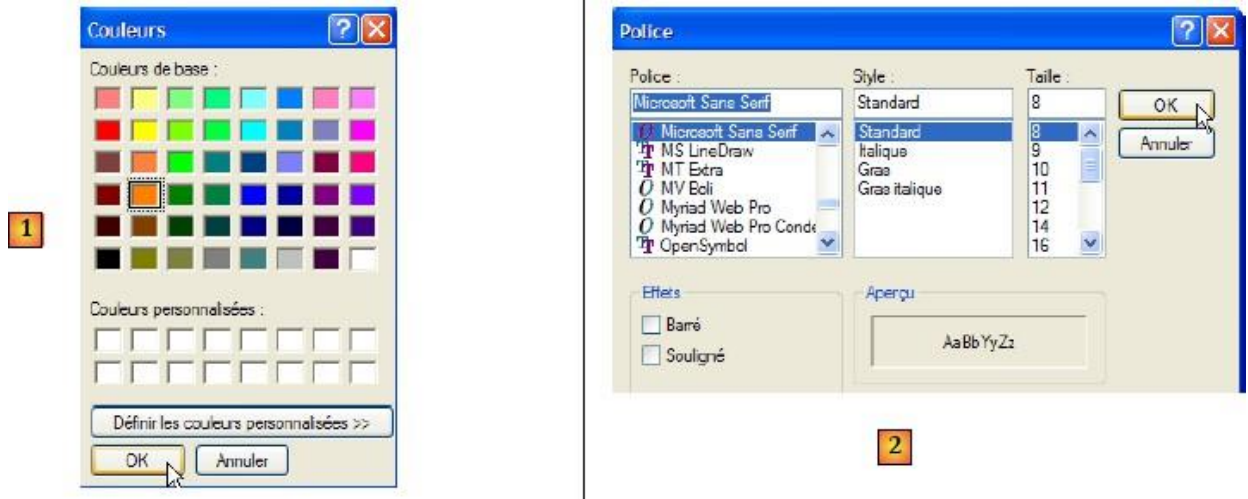
Partie II : WinForms



N°	type	nom	rôle
1	Button	buttonCouleur	pour fixer la couleur des caractères du TextBox
2	Button	buttonPolice	pour fixer la police de caractères du TextBox
3	ColorDialog	colorDialog1	le composant qui permet la sélection d'une couleur - pris dans la boîte à outils [5].
4	FontDialog	colorDialog1	le composant qui permet la sélection d'une police de caractères - pris dans la boîte à outils [5].

Les classes *FontDialog* et *ColorDialog* ont une méthode *ShowDialog* analogue à la méthode *ShowDialog* des classes *OpenFileDialog* et *SaveFileDialog*.

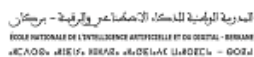
La méthode *ShowDialog* de la classe *ColorDialog* permet de choisir une couleur [1]. Celle de la classe *FontDialog* permet de choisir une police de caractères [2] :



[1] : si l'utilisateur quitte la boîte de dialogue avec le bouton *OK*, le résultat de la méthode *ShowDialog* est *DialogResult.OK* et la couleur choisie est dans la propriété *Color* de l'objet *ColorDialog* utilisé.

[2] : si l'utilisateur quitte la boîte de dialogue avec le bouton *OK*, le résultat de la méthode *ShowDialog* est *DialogResult.OK* et la police choisie est dans la propriété *Font* de l'objet *FontDialog* utilisé.

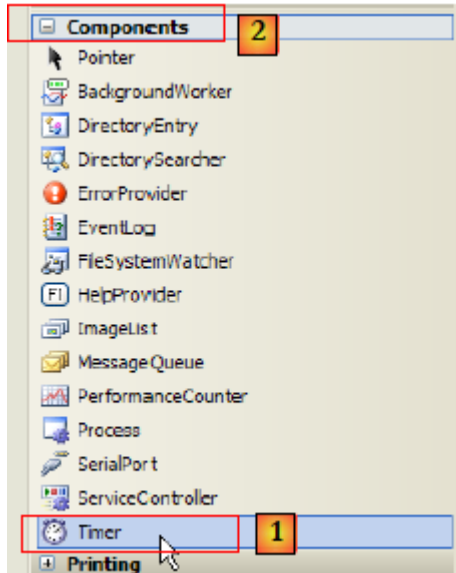
Nous avons désormais les éléments pour traiter les clics sur les boutons *Couleur* et *Police* :





Développement d'applications .NET

Partie II : WinForms



Les propriétés du composant Timer utilisées ici seront les suivantes :

Interval	nombre de millisecondes au bout duquel un événement <i>Tick</i> est émis.
Tick	l'événement produit à la fin de <i>Interval</i> millisecondes
Enabled	rend le timer actif (true) ou inactif (false)

Dans notre exemple le timer s'appelle *timer1* et *timer1.Interval* est mis à 1000 ms (1s). L'événement *Tick* se produira donc toutes les secondes. Le clic sur le bouton Arrêt/Marche est traité par la procédure *buttonArretMarche_Click* suivante :



Développement d'applications .NET

Partie II : WinForms

```
1. using System;
2. using System.Windows.Forms;
3.
4. namespace Chap5 {
5.     public partial class Form1 : Form {
6.         public Form1() {
7.             InitializeComponent();
8.         }
9.
10.        // variable d'instance
11.        private DateTime début = DateTime.Now;
12.    ...
13.    private void buttonArretMarche_Click(object sender, EventArgs e) {
14.        // arrêt ou marche ?
15.        if (buttonArretMarche.Text == "Marche") {
16.            // on note l'heure de début
17.            début = DateTime.Now;
18.            // on l'affiche
19.            labelChrono.Text = "00:00:00";
20.            // on lance le timer
21.            timer1.Enabled = true;
22.            // on change le libellé du bouton
23.            buttonArretMarche.Text = "Arrêt";
24.            // fin
25.            return;
26.        }
27.        if (buttonArretMarche.Text == "Arrêt") {
28.            // arrêt du timer
29.            timer1.Enabled = false;
30.            // on change le libellé du bouton
31.            buttonArretMarche.Text = "Marche";
32.            // fin
33.            return;
34.        }
35.    }
36.
37. }
38. }
```

- ligne 13 : la procédure qui traite le clic sur le bouton Arrêt/Marche.
- ligne 15 : le libellé du bouton Arrêt/Marche est soit "Arrêt" soit "Marche". On est donc obligé de faire un test sur ce libellé pour savoir quoi faire.
- ligne 17 : dans le cas de "Marche", on note l'heure de début dans une variable *début* qui est une variable globale (ligne 11) de l'objet formulaire
- ligne 19 : initialise le contenu du label *LabelChrono*
- ligne 21 : le timer est lancé (Enabled=true)
- ligne 23 : libellé du bouton passe à "Arrêt".
- ligne 27 : dans le cas de "Arrêt"
- ligne 29 : on arrête le timer (Enabled=false)
- ligne 31 : on passe le libellé du bouton à "Marche".

Il nous reste à traiter l'événement *Tick* sur l'objet *timer1*, événement qui se produit toutes les secondes :

```
1. private void timer1_Tick(object sender, EventArgs e) {
2.     // une seconde s'est écoulée
3.     DateTime maintenant = DateTime.Now;
4.     TimeSpan durée = maintenant - début;
5.     // on met à jour le chronomètre
6.     labelChrono.Text = durée.Hours.ToString("d2") + ":" + durée.Minutes.ToString("d2") + ":" +
       durée.Seconds.ToString("d2");
7. }
```


- ligne 3 : on note l'heure du moment
- ligne 4 : on calcule le temps écoulé depuis l'heure de lancement du chronomètre. On obtient un objet de type *TimeSpan* qui représente une durée dans le temps.
- ligne 6 : celle-ci doit être affichée dans le chronomètre sous la forme *hh:mm:ss*. Pour cela nous utilisons les propriétés *Hours*, *Minutes*, *Seconds* de l'objet *TimeSpan* qui représentent respectivement les heures, minutes, secondes de la durée que nous affichons au format *ToString("d2")* pour avoir un affichage sur 2 chiffres.

12. Regroupement de contrôles

On peut regrouper des contrôles dans :

- Les *GroupBox*.
- Les *Panels*.
- Les *PictureBox*.
- Les *TabControl*.

a) GroupBox et Panel

Il est possible de regrouper des contrôles dans un container, on peut par exemple regrouper plusieurs *RadioButton*. Le container peut être un *GroupBox* ou un *Panel*.



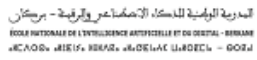
Pour l'utilisateur, le fait que toutes les options soient regroupées dans un panneau est un indice visuel logique (Tous les *RadioButton* permettent un choix dans une même catégorie de données). Au moment de la conception, tous les contrôles peuvent être déplacés facilement ; si vous déplacez le contrôle *GroupBox* ou *Panel*, tous les contrôles qu'il contient sont également déplacés.

b) PictureBox

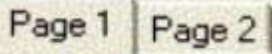
Le contrôle *PictureBox* peut afficher une image mais peut aussi servir de conteneur à d'autres contrôles. Retenons la notion de conteneur qui est le contrôle parent.

c) TabControl

Ce contrôle permet de créer des onglets comme dans un classeur, onglets entièrement gérés par C#. Chaque page peut contenir d'autres contrôles. En mode conception, en passant par la propriété *TabPage*, on ajoute des onglets dont la propriété *Text* contient le texte à afficher en haut (Ici: Page 1..). il suffit ensuite de cliquer sur chaque onglet et d'y ajouter les contrôles.



Partie II : WinForms



[LinkLabel2](#)