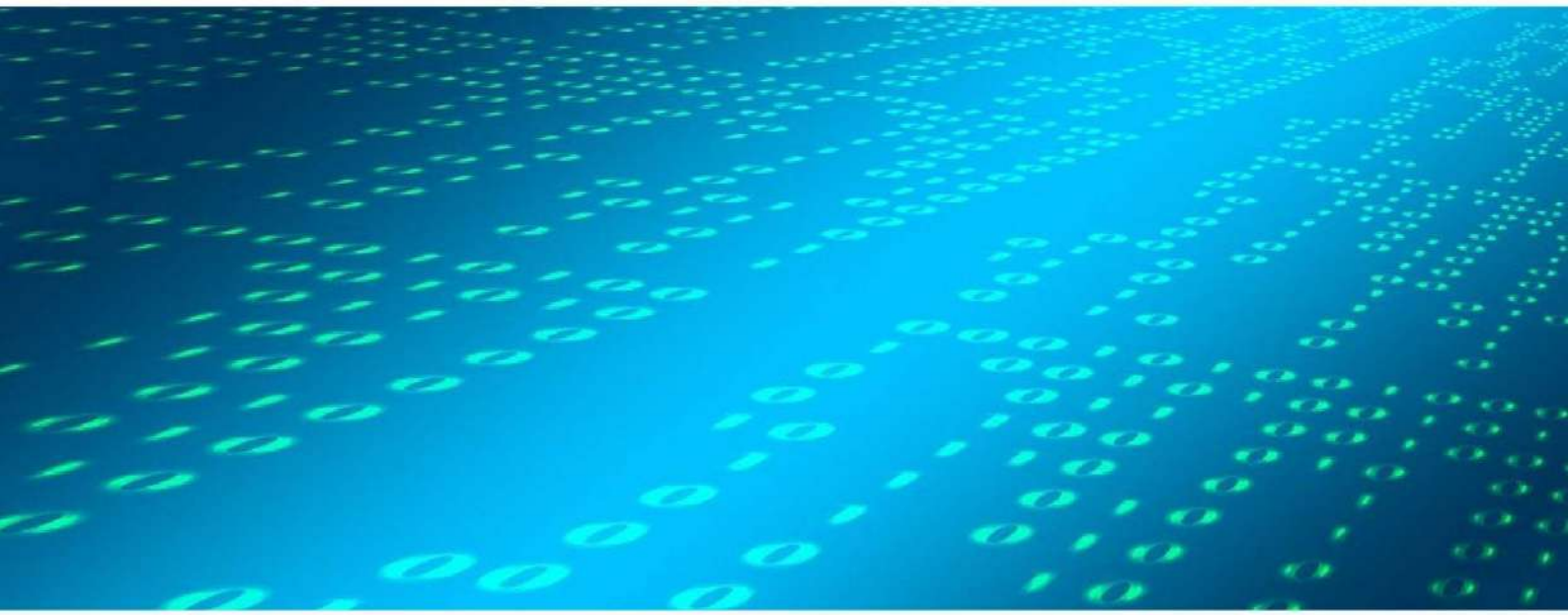
Bordeaux INP
ENSC

Programmation Evènementielle Avec les WinForms



Baptiste Pesquet

Table des matières

Programmation évènementielle avec les WinForms

Ce livre est un support de cours à l'Ecole Nationale Supérieure de Cognitique.

Les exemples de code associés sont disponibles en ligne.

Résumé

Ce livre aborde la programmation évènementielle et son application dans le cadre de la technologie Microsoft WinForms.

- Introduction à la programmation évènementielle.
- La technologie WinForms.
- Aperçu des principaux contrôles WinForms.
- Opérations courantes avec les WinForms.
- Interactions avec des fichiers.
- WinForms et multithreading.

Pré-requis

L'étude des exemples de code de ce livre nécessite une connaissance minimale de la programmation orientée objet et du langage C#.

> Au besoin, consultez le livre Programmation orientée objet en C#.

Remerciements

Certains éléments et illustrations sont empruntés à d'autres supports, notemment ceux de mes collègues David Duron et Jean-Marc Salotti.

Contributions

Merci d'avance et bonne lecture !

La programmation évènementielle

L'objectif de ce chapitre est de découvrir les grands principes de la programmation évènementielle.

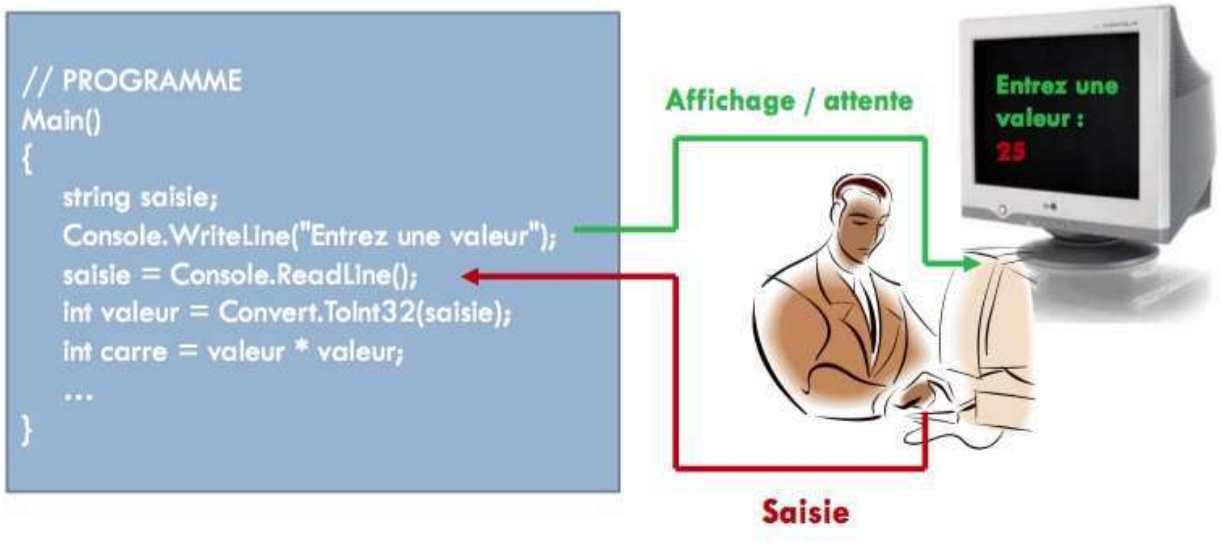
Un nouveau paradigme

> Au sens large, un paradigme est un ensemble partagé de croyances et de valeurs, une manière commune de voir les choses. En informatique, un paradigme est un style fondamental de programmation. La programmation procédurale et la programmation orientée objet sont deux exemples de paradigmes.

Prenons l'exemple d'un programme très simple écrit en langage C#.

```
static void Main(string[] args)
{
    string saisie;
    Console.WriteLine("Entrez une valeur");
    saisie = Console.ReadLine();
    int valeur = Convert.ToInt32(saisie);
    int carre = valeur * valeur;
    // ...
}
```

Ce programme est écrit selon le paradigme de programmation séquentielle. A partir du point d'entrée (ici la méthode statique `Main`), ses instructions se déroulent toujours dans le même ordre prévu à l'avance. L'utilisateur fait ce que lui demande le programme : c'est ce dernier qui a le contrôle.



Un programme écrit selon le paradigme évènementiel fonctionne différemment : il *réagit* à des évènements provenant du système ou de l'utilisateur. L'ordre d'exécution des instructions n'est donc plus prévu à l'avance. C'est l'utilisateur qui a le contrôle du programme.

```
// PROGRAMME
Main()
{
  ...
  while(true)  // tantque Mamie s'active
  {
     // récupérer son action (faire une maille ...)
     e = getNextEvent();
     // traiter son action (agrandir le tricot ...)
     processEvent();
  }
  ...
}
```

La programmation évènementielle s'oppose donc à la programmation séquentielle. Elle est notamment utilisée pour gérer des interactions riches avec l'utilisateur, comme celles des interfaces graphiques homme-machine (GUI, *Graphical User Interface*).

Les évènements

La programmation évènementielle est fondée sur les **évènements**. Un évènement représente un message envoyé à l'application. Les évènements peuvent être d'origines diverses : action de l'utilisateur (déplacement de la souris, clic sur un bouton, appui sur une touche du clavier, etc) ou évènement système (chargement d'un fichier, déclenchement d'une minuterie, etc).

Le plus souvent, un évènement contient des informations qui dépendent du type d'évènement. Ces données peuvent être utilisées par l'application pour réagir au mieux. Elle shosit les évènements auxquels elle va répondre par un traitement particulier, et ignore tous les autres.

La technologie WinForms

L'objectif de ce chapitre est d'appréhender le fonctionnement de la technologie Microsoft WinForms.

Introduction

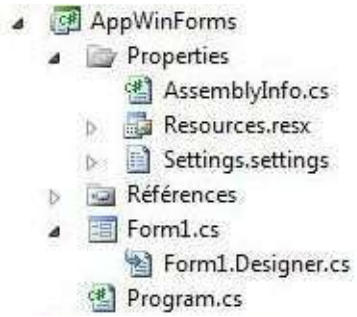
WinForms (abréviation de Windows Forms) est une plate-forme de création d'interfaces graphiques créée par Microsoft. Elle est adossée au framework .NET et peut être déployée sur des environnements de bureau ou mobiles.

Cette technologie suit le paradigme de programmation évènementielle : une application WinForms est pilotée par des évènements auxquels elle réagit.

Structure d'une application WinForms

Une application WinForms est structurée autour d'un ou plusieurs formulaires, appelés **forms**.

Lorsqu'on crée une nouvelle application WinForms, l'IDE Visual Studio génère automatiquement plusieurs éléments qu'il est important d'identifier.



Les fichiers du répertoire `Properties` sont gérés par Visual Studio. Il ne doivent pas être édités manuellement. Etudions en détail le reste de l'arborescence.

Programme principal

Le fichier `Program.cs` correspond au point d'entrée dans l'application. Voici son contenu par défaut.

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
}
```

Comme pour une application console, la méthode statique `Main` est le point d'entrée dans le programme. Cette méthode crée (`new Form1()`) puis affiche le premier formulaire de l'application.

Anatomie d'un formulaire

Chaque formulaire WinForms est décrit par deux fichiers :

- Un fichier `.Designer.cs` qui contient le code généré automatiquement par l'IDE lors de la conception graphique du formulaire.
- Un fichier `.cs` qui contient le code C# écrit par le développeur pour faire réagir le formulaire aux évènements qui se produisent. Ce fichier est appelé "*code behind*".

> Après chaque création de formulaire, une bonne pratique consiste à lui donner immédiatement un nom plus parlant, par exemple `MainForm` pour le formulaire principal. Pour cela, faites un clic droit sur le formulaire dans l'arborescence, puis choisissez **Renommer**.

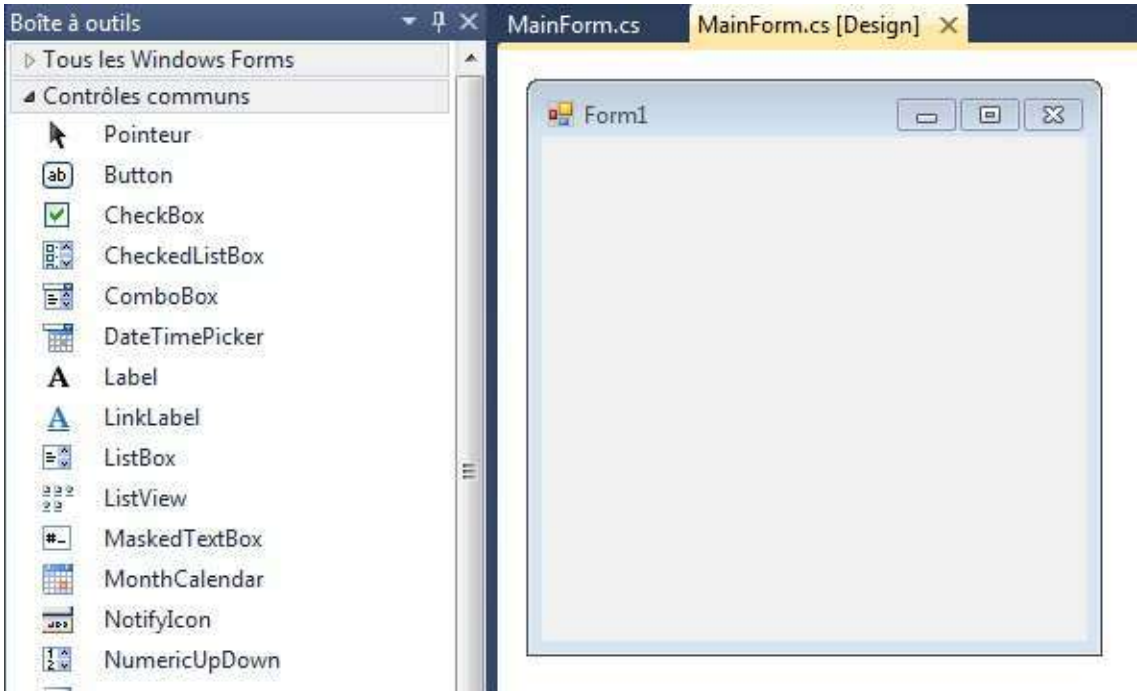
Le fichier "code behind" `.cs` associé à un formulaire est accessible en faisant un clic droit sur le formulaire puis en choisissant **Afficher le code**, ou à l'aide du raccourci clavier **F7**. Voici son contenu initial.

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }
}
```

Il s'agit de la définition d'une classe avec son constructeur. Cette classe hérite de la classe `Form`, définie par le framework .NET et qui rassemble les fonctionnalités communes à tous les formulaires. On remarque la présence du mot-clé `partial`. Il indique que seul une partie du code de la classe est présent dans ce fichier. Le reste se trouve, comme vous l'avez deviné, dans le fichier `.Designer.cs`.

Edition graphique d'un formulaire

Un double-clic sur le formulaire dans l'arborescence déclenche l'apparition du **concepteur de formulaire**. Cette interface va permettre d'éditer l'apparence du formulaire. Nous en reparlerons plus loin.



Ajout d'un contrôle

L'édition du formulaire se fait en y glissant/déposant des **contrôles**, rassemblées dans une boîte à outils (liste de gauche). De nombreux contrôles sont disponibles pour répondre à des besoins variés et construire des IHM riches et fonctionnelles. Parmi les plus utilisés, on peut citer :

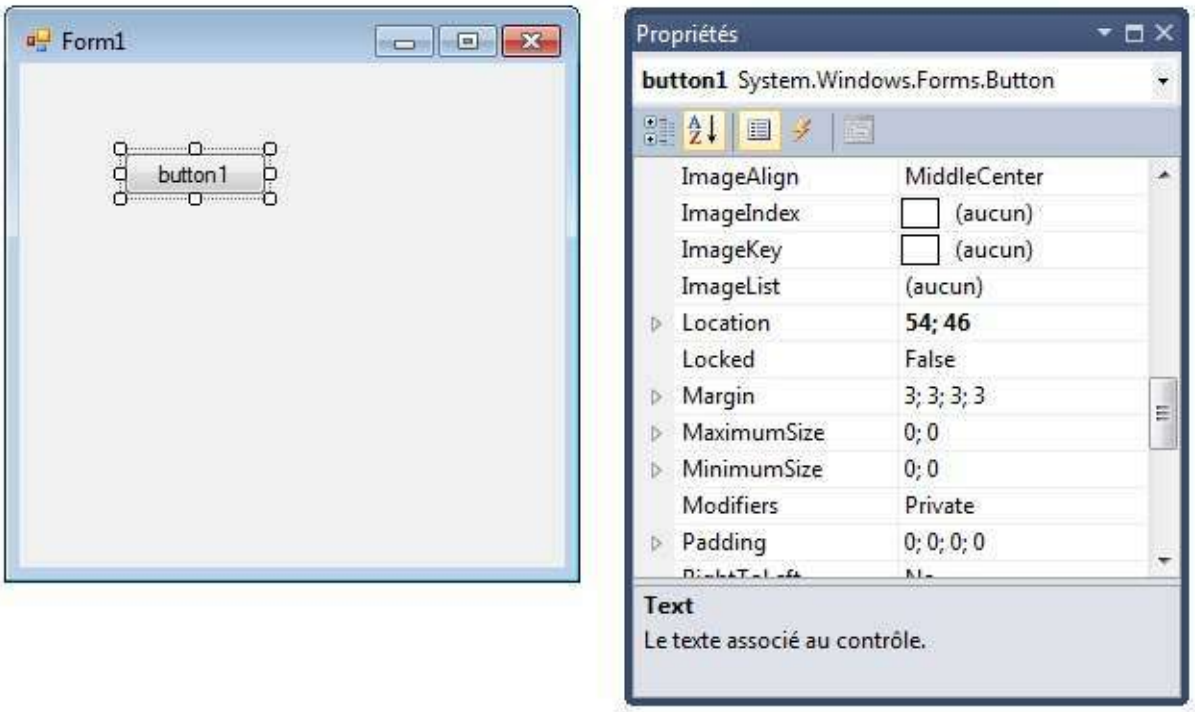
- **Label** qui affiche un simple texte;
- **TextBox** qui crée une zone de saisie de texte;
- **Button** qui affiche un bouton;
- **ListBox** qui regroupe une liste de valeurs.

> Pour découvrir le comportement d'un contrôle, testez-le !

Par exemple, l'ajout d'un bouton au formulaire se fait en cliquant sur le contrôle "Button" dans la boîte à outils, puis en faisant glisser le contrôle vers le formulaire.

Propriétés d'un contrôle

La sélection d'un contrôle (ou du formulaire lui-même) dans le concepteur permet d'afficher ses **propriétés** dans une zone dédiée, située par défaut en bas à droite de l'IDE.

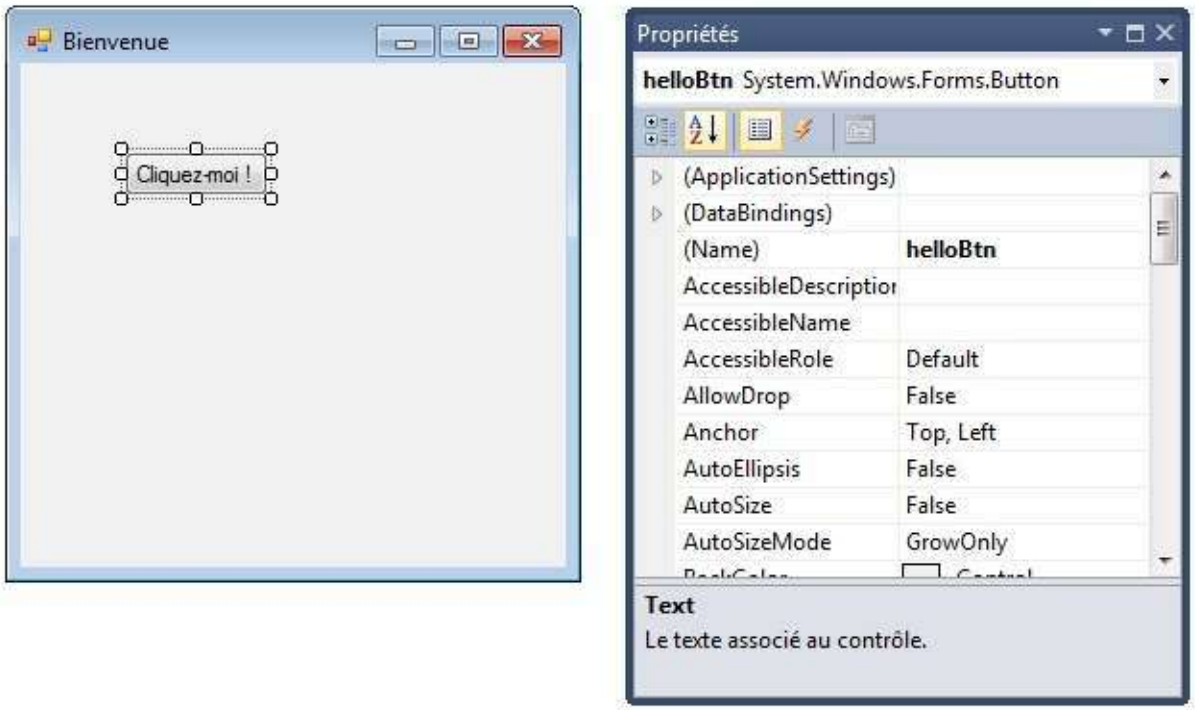


Chaque contrôle dispose d'un grand nombre de propriétés qui gouvernent son apparence et son comportement. Parmi les propriétés essentielles, citons :

- **(Name)** : le nom de l'attribut représentant le contrôle dans la classe.
- **Dock** : l'ancrage du contrôle dans celui qui le contient.
- **Enabled** : indique si le contrôle est actif ou non.
- **Text** : le texte affiché par le contrôle.
- **Visible** : indique si le contrôle est visible ou non.

> Tout comme le nom d'un formulaire, celui d'un contrôle doit être immédiatement modifié avec une valeur plus parlante.

Par exemple, donnons à notre nouveau bouton le nom `helloBtn` et le texte `Cliquez-moi !`. Donnons également à notre formulaire le titre `Bienvenue` (propriété **Text** du formulaire).



Gestion des évènements

Ajout d'un gestionnaire d'évènement

En double-cliquant sur un contrôle dans le concepteur de formulaire, on ajoute un **gestionnaire d'évènement** pour l'évènement par défaut associé au contrôle. Dans le cas d'un bouton, l'évènement par défaut est le clic.

Un gestionnaire d'évènement représente le code exécuté lorsque l'évènement associé se produit. Tous les gestionnaires sont regroupées dans le fichier "code behind" `.cs`. Voici le gestionnaire par défaut associé à un clic sur un bouton.

```
public partial class MainForm : Form
{
    // ...

    // Gère le clic sur le boutton helloBtn
    private void helloBtn_Click(object sender, EventArgs e)
    {

    }
}
```

Un gestionnaire d'évènement WinForms correspond à une méthode dans la classe associée au formulaire. Le nom de cette méthode est composé du nom du contrôle suivi de celui de l'évènement. Cette méthode reçoit deux paramètres offrant des détails sur l'évènement :

- `sender` représente le contrôle qui a déclenché l'évènement.
- `e` rassemble les paramètres liés à l'évènement. Son contenu dépend du type de l'évènement.

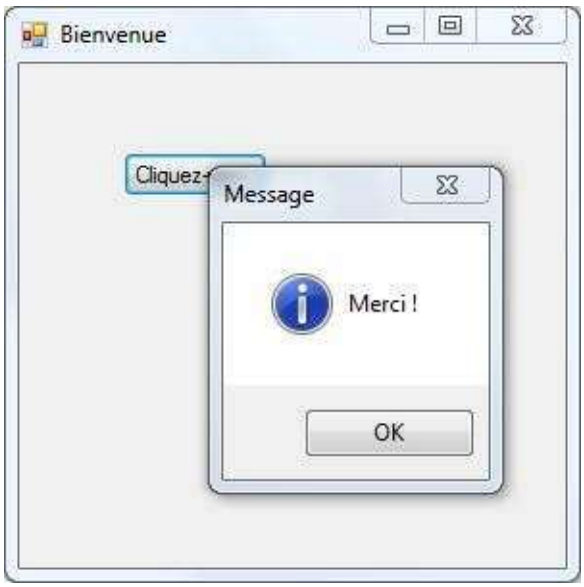
Affichage d'un message

Modifions le gestionnaire pour afficher un message à l'utilisateur.

```
private void helloBtn_Click(object sender, EventArgs e)
{
    MessageBox.Show("Merci !", "Message",
        MessageBoxButtons.OK, MessageBoxIcon.Information);
}
```

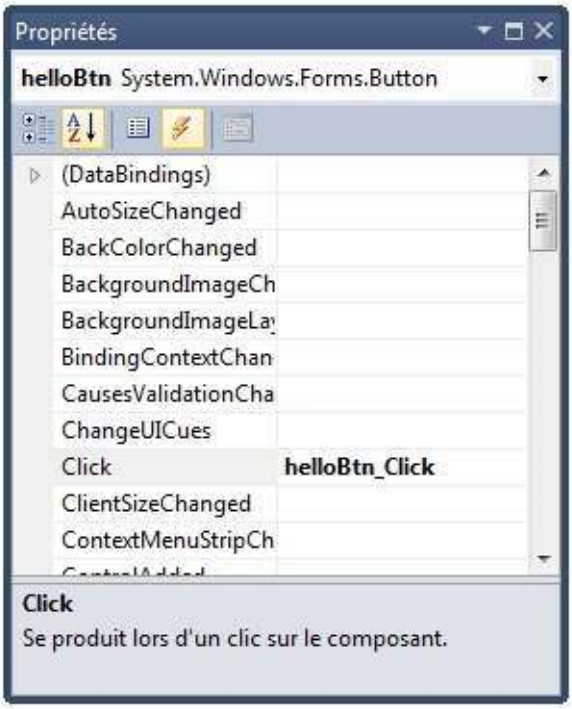
La méthode statique `Show` de la classe `MessageBox` affiche un message à l'utilisateur. Plusieurs surcharges de cette méthode permettent de paramétrer l'apparence du message (texte, titre, boutons, icône).

Voici le résultat d'un clic sur le bouton du formulaire, une fois l'application exécutée.



Gestion des gestionnaires

Dans le concepteur de formulaire, la zone des propriétés permet de gérer les évènements associés à un contrôle (ou au formulaire lui-même). Un clic sur le petit bouton en forme d'éclair affiche la liste de tous les évènements que l'élément peut générer, ainsi que les gestionnaires d'évènements ajoutés.



Le lien entre le contrôle et le gestionnaire se fait dans le fichier `.Designer.cs`. Son contenu est complexe et géré par Visual Studio, mais on peut tout de même le consulter pour y trouver le code ci-dessous.

```
partial class MainForm
{
    // ...

    #region Code généré par le Concepteur Windows Form

    /// <summary>
    /// Méthode requise pour la prise en charge du concepteur - ne modifiez pas
    /// le contenu de cette méthode avec l'éditeur de code.
    /// </summary>
    private void InitializeComponent()
    {
        this.helloBtn = new System.Windows.Forms.Button();
        // ...
        this.helloBtn.Click += new System.EventHandler(this.helloBtn_Click);
        // ...
    }

    #endregion

    private System.Windows.Forms.Button helloBtn;
}
```

L'attribut privé `helloBtn` correspond au bouton ajouté au formulaire. Il est instancié dans la méthode `InitializeComponent`, appelée par le constructeur du formulaire (voir plus haut). Ensuite, on lui ajoute (opérateur `+=`) le gestionnaire `helloBtn_Click` pour l'évènement `Click`.

Principaux contrôles WinForms

L'objectif de ce chapitre est de présenter succinctement quelques contrôles WinForms parmi les plus utilisés.

> D'autres contrôles plus spécialisés seront présentés plus loin.

Nommage des contrôles

Comme indiqué précédemment, il est fortement recommandé de donner un nom parlant à un contrôle immédiatement après son ajout au formulaire. Cela augmente fortement la lisibilité du code utilisant ce contrôle.

On peut aller plus loin et choisir une convention de nommage qui permet d'identifier clairement le type du contrôle. Il n'existe pas de consensus à ce sujet :

- On peut rester générique et suffixer tous les contrôles par `Ctrl`.
- On peut choisir un suffixe différent pour chaque type de contrôle.

L'important est de rester cohérent dans la convention choisie afin que le code soit uniforme.

Affichage de texte

Le contrôle **Label** est dédié à l'affichage d'un texte *non modifiable*.

> On peut donner à un contrôle de ce type un nom finissant par `Lbl`. Exemple : `loginLbl`.

Il dispose d'une propriété **Text** qui permet de récupérer ou de définir le texte affiché par le contrôle.

Saisie de texte

Le contrôle **TextBox** crée une zone de saisie de texte.

> On peut donner à un contrôle de ce type un nom finissant par `TB`. Exemple : `loginTB`.

Voici ses propriétés importantes :

- **Text** permet de récupérer ou de définir la valeur saisie.
- **Multiline** précise si le texte saisi peut comporter une ou plusieurs lignes.
- **ReadOnly** permet, quand elle vaut `true`, d'en faire une zone en lecture seule (saisie impossible).

> Le contrôle **RichTextBox** est une version enrichie de ce contrôle.

Liste déroulante

Le contrôle **ComboBox** définit une liste déroulante.

> On peut donner à un contrôle de ce type un nom finissant par `CB`. Exemple : `countryCB`.

Voici ses propriétés importantes :

- **Items** regroupe ses valeurs sous la forme d'une liste (collection) d'objets. On peut ajouter des valeurs dans la liste déroulante dans le concepteur du formulaire ou via le code.
- **DropDownStyle** permet de choisir le style de la liste. Pour obtenir des valeurs non éditables par l'utilisateur, il faut choisir le style `DropDownList`.
- **SelectedIndex** récupère ou définit l'indice de l'élément actuellement sélectionné. Le premier élément correspond à l'indice 0.
- **SelectedItem** renvoit l'élément actuellement sélectionné sous la forme d'un objet.

```
// Ajoute 3 pays à la liste
countryCB.Items.Add("France");
countryCB.Items.Add("Belgique");
countryCB.Items.Add("Suisse");

// Sélectionne le 2ème élément de la liste
countryCB.SelectedIndex = 1;
```

L'évènement **SelectedIndexChanged** permet de gérer le changement de la valeur sélectionnée de la liste.

```
// Gère le changement de sélection dans la liste déroulante
private void countryCB_SelectedIndexChanged(object sender, EventArgs e)
{
    // On caste l'objet renvoyé par SelectedItem vers le type chaîne
    string selectedValue = (string) countryCB.SelectedItem;
    // ...
}
```

Liste d'éléments

Le contrôle **ListBox** permet de créer une liste d'éléments.

> On peut donner à un contrôle de ce type un nom finissant par `LB`. Exemple : `hardwareLB`.

Voici ses propriétés importantes :

- **Items** regroupe ses valeurs sous la forme d'une liste (collection) d'objets. On peut ajouter des valeurs dans la liste déroulante dans le concepteur du formulaire ou via le code.
- **SelectionMode** permet de choisir si la liste est à sélection simple, multiple ou si la sélection est désactivée.
- **SelectedIndex** récupère ou définit l'indice de l'élément actuellement sélectionné. Le premier élément correspond à l'indice 0.
- **SelectedItems** renvoit les éléments actuellement sélectionnés sous la forme d'une liste d'objets.

```
// Ajoute 4 éléments à la liste
hardwareLB.Items.Add("PC");
hardwareLB.Items.Add("Mac");
hardwareLB.Items.Add("Tablette");
hardwareLB.Items.Add("Smartphone");

// Sélectionner le 1er élément
hardwareLB.SelectedIndex = 0;
```

L'évènement **SelectedIndexChanged** permet de gérer le changement de sélection dans la liste.

```
// Gère le changement de sélection dans la liste
private void hardwareLB_SelectedIndexChanged(object sender, EventArgs e)
{
    // Parcours de la liste des éléments sélectionnés
    foreach (string value in hardwareLB.SelectedItems)
    {
        // ...
    }
}
```

> Le contrôle **ListView** est une version enrichie de ce contrôle.

Bouton

Le contrôle **Button** permet de créer un bouton.

> On peut donner à un contrôle de ce type un nom finissant par `Btn`. Exemple : `connectBtn`.

Voici ses propriétés importantes :

- **Text** récupère ou définit le texte affiché dans le bouton.
- **Enabled** active ou désactive le bouton (clic impossible).
- **DialogResult** définit le résultat renvoyé lors d'un clic sur le bouton quand le formulaire est affichée de manière modale (voir chapitre suivant).

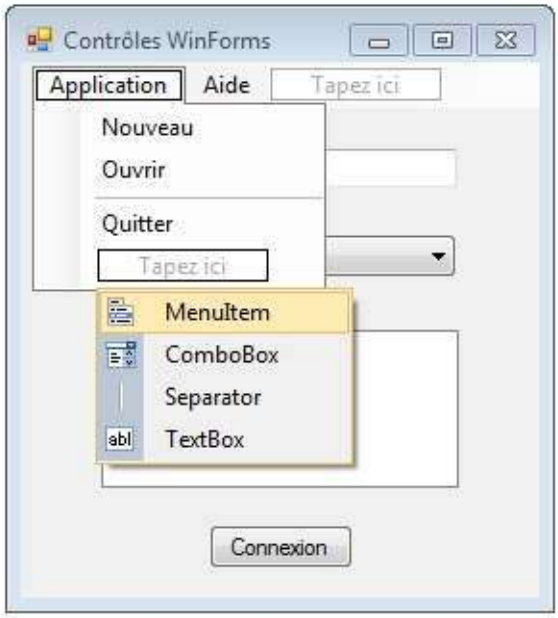
L'évènement **Click** permet de gérer le clic sur le bouton.

```
// Gère le clic sur le bouton de connexion
private void connectBtn_Click(object sender, EventArgs e)
{
    // ...
}
```

> Les contrôles **CheckBox** et **RadioButton** permettent respectivement de créer une case à cocher et un bouton bouton radio.

Barre de menus

Le contrôle **MenuStrip** permet de créer une barre de menus déroulants. Une entrée de menu déroulant peut être une commande (**MenuItem**), une liste déroulante, une zone de texte ou un séparateur.



> Dans le cas d'une barre de menus, on peut conserver le nommage des contrôles initial proposé par Visual Studio.

L'évènement **Click** permet de gérer le clic sur une commande du menu déroulant.

```
// Gère le clic sur la commande Quitter
private void quitterToolStripMenuItem_Click(object sender, EventArgs e)
{
    // ...
}
```

Opérations courantes avec les WinForms

L'objectif de ce chapitre est de rassembler les solutions à des besoins courants lorsqu'on développe une application WinForms.

Positionner un contrôle

Le concepteur de formulaire permet de placer précisément un contrôle sur un formulaire. La barre d'outils **Disposition** offre des fonctionnalités avancées pour le positionnement :

- Alignement de contrôles
- Centrage horizontal et vertical
- Uniformisation de l'espace entre plusieurs contrôles.
- ...

Gérer le redimensionnement d'un formulaire

Une IHM réussie doit adapter sa présentation à la taille de sa fenêtre, notamment lorsque celle-ci est redimensionnée par l'utilisateur.

Interdire le redimensionnement

Une première solution, simple et radicale, consiste à interdire tout redimensionnement du formulaire. Pour cela, il faut modifier les propriétés suivantes du formulaire :

- **FormBorderStyle** : `Fixed3D` (ou une autre valeur commençant par `Fixed`).
- **MaximizeBox** et **MinimizeBox** ; `false`

Ainsi, le formulaire ne sera plus redimensionnable et les icônes pour le minimiser/maximiser ne seront plus affichés.

Gérer le positionnement des contrôles

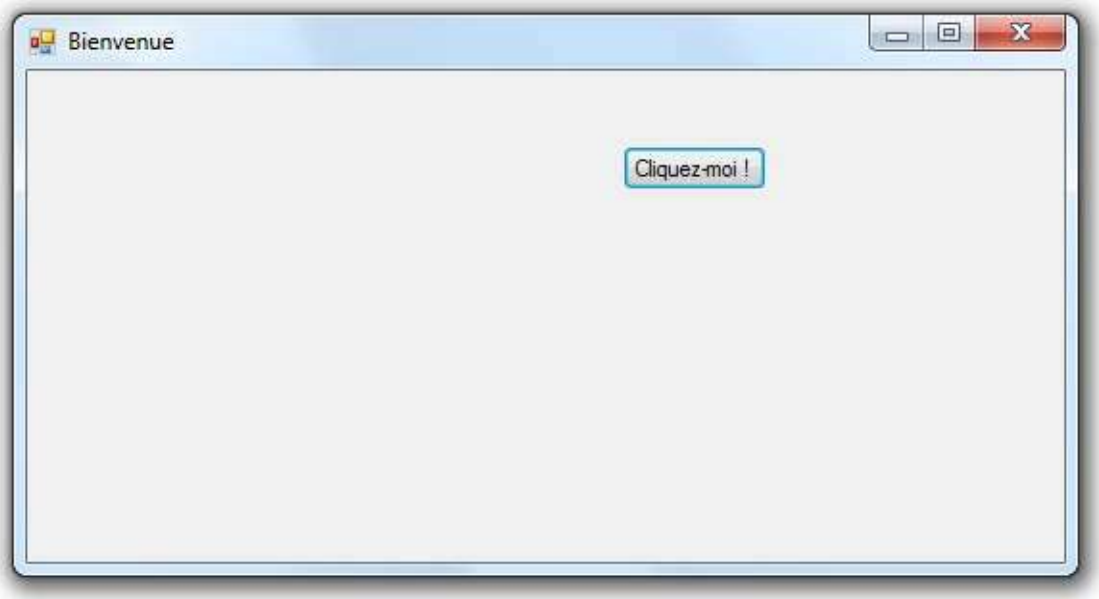
Si le formulaire doit pouvoir être redimensionné, il faut prévoir de quelle manière les contrôles qu'il contient vont être repositionnés pour que le résultat affiché soit toujours harmonieux.

Le positionnement d'un contrôle s'effectue par rapport à son conteneur, qui peut être le formulaire ou un autre contrôle (exemples : **GroupBox**, **Panel**, **TabControl**). Les deux propriétés qui gouvernent le positionnement sont **Anchor** et **Dock**.

- **Anchor** décrit l'ancrage du contrôle par rapport aux bordures de son conteneur. Lorsqu'un contrôle est ancré à une bordure, la distance entre le contrôle et cette bordure restera toujours constante.

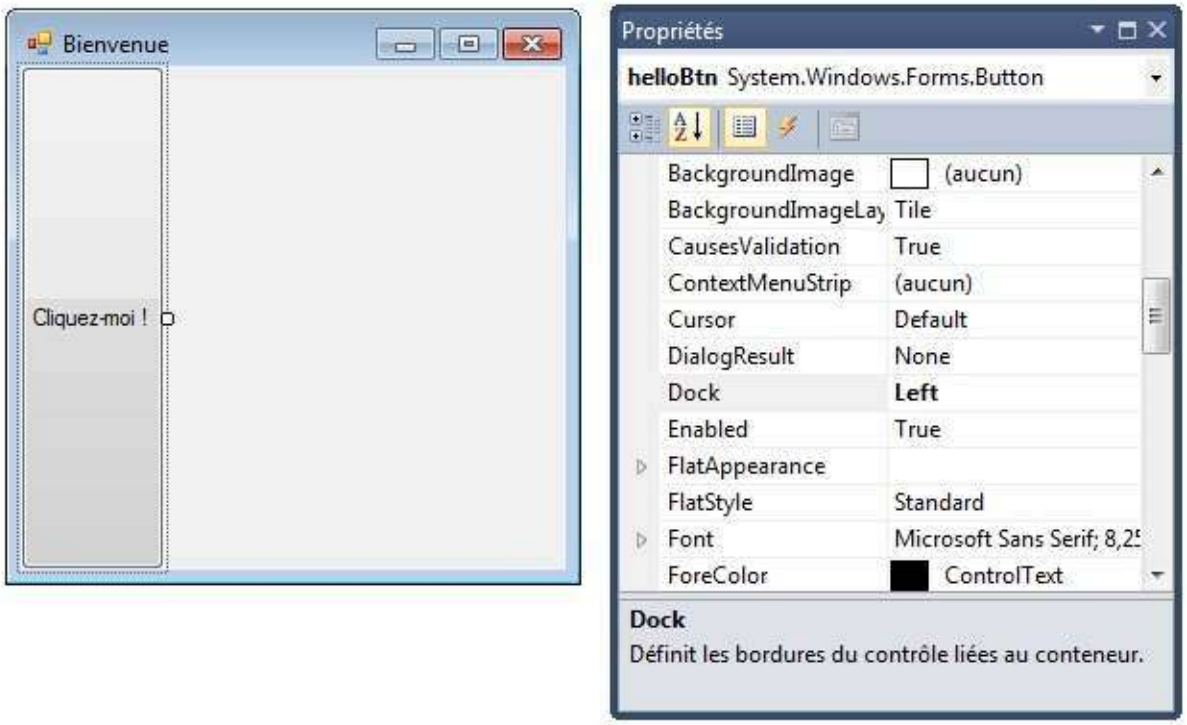
> Par défaut, les contrôles sont ancrés en haut (`Top`) et à gauche (`Left`) de leur conteneur.

Si on donne à un contrôle les valeurs d'ancrage `Bottom` et `Right` , il sera déplacé pour conserver les mêmes distances avec les bordures bas et droite lors d'un redimensionnement de son conteneur. C'est le cas pour le bouton dans l'exemple ci-dessous.



- **Dock** définit la ou les bordures du contrôles directement attachées au conteneur parent. Le contrôle prendra toute la place disponible sur la ou les bordure(s) en question, même en cas de redimensionnement.

Voici l'exemple d'un bouton attaché à la bordure gauche de son conteneur.



Gérer l'arrêt de l'application

Déclencher l'arrêt

Une application dont le programme principal contient une ligne du type `Application.Run(new MainForm())` se termine automatiquement lorsque l'utilisateur décide de fermer le formulaire `MainForm`.

Depuis un gestionnaire d'évènement, le même résultat s'obtient avec la commande `Application.Exit()`.

Confirmer l'arrêt

Pour effectuer une demande de confirmation avant la fermeture, il faut ajouter un gestionnaire pour l'évènement **FormClosing** du formulaire. Dans ce gestionnaire, on peut afficher un message puis récupérer le choix de l'utilisateur.

```
// Gère la fermeture du formulaire par l'utilisateur
private void MainForm_FormClosing(object sender, FormClosingEventArgs e)
{
    if (MessageBox.Show("Etes-vous sûr(e) ?", "Demande de confirmation",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.No)
    {
        e.Cancel = true;
    }
}
```

la méthode `Show` renvoie son résultatsous la forme d'une valeur de type `DialogResult`. Si l'utilisateur répond `Non` (valeur `DialogResult.No`) à la demande de confirmation , la propriété `Cancel` de l'objet `FormClosingEventArgs` est définie à `true` pour annuler la fermeture. Sinon, l'application s'arrête.

> L'appel à `Application.Exit()` depuis un gestionnaire d'évènement déclenche l'évènement **FormClosing** et donc la demande de confirmation.

Afficher un formulaire

Une application WinForms se compose le plus souvent de plusieurs formulaires ayant des rôles différents.

L'affichage d'un formulaire peut se faire de façon modale ou non modale. Un formulaire **modal** doit être fermé ou masqué avant de pouvoir utiliser de nouveau le reste de l'application. C'est le cas des formulaire de type boîte de dialogue qui permettent d'interroger l'utilisateur avant de poursuivre le déroulement de l'application.

Formulaire non modal

On affiche un formulaire non modal en instanciant un objet de la classe correspondante, puis en appelant la méthode `Show` sur cet objet.

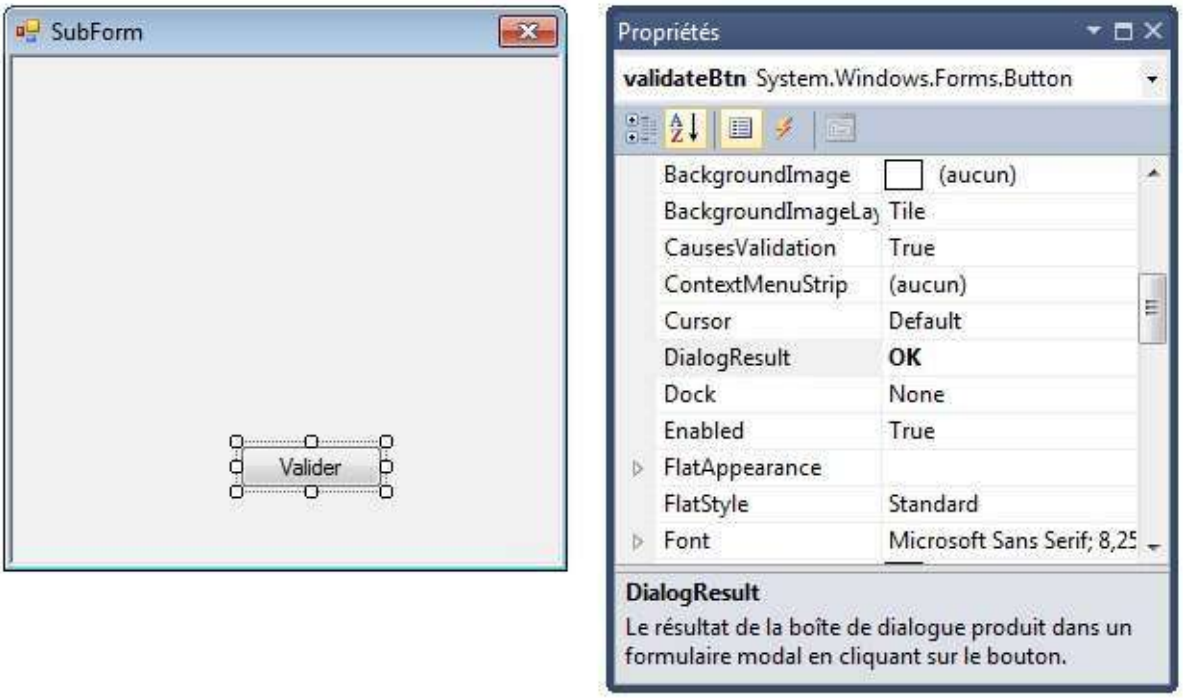
```
// Affiche le formulaire SubForm (non modal)
SubForm subForm = new SubForm();
subForm.Show();
```

Formulaire modal

On affiche un formulaire modal en l'instanciant, puis en appelant sa méthode `ShowDialog`. Comme la méthode `MessageBox.Show`, elle renvoie un résultat de type `DialogResult` qui permet de connaître le choix de l'utilisateur.

```
// Affiche le formulaire SubForm (modal)
SubForm subForm = new SubForm();
if (subForm.ShowDialog() == DialogResult.OK)
{
    // ...
}
```

Pour que cela fonctionne, il faut ajouter au formulaire modal au moins un bouton, puis avoir défini la propriété `DialogResult` de ce bouton. La valeur de cette propriété est renvoyée par la méthode `ShowDialog` lorsque l'utilisateur clique sur le bouton associé du formulaire modal.



Si le formulaire modal contient plusieurs boutons avec une propriété `DialogResult` définie et différente pour chacun d'eux, la valeur renvoyée par `ShowDialog` permet d'identifier celui sur lequel l'utilisateur a cliqué.

Echanger des données entre formulaires

Fournir des données à un formulaire

La manière la plus simple de fournir des données à un formulaire est de modifier son constructeur pour qu'il accepte des paramètres. Dans l'exemple ci-dessous, le constructeur du formulaire utilise la chaîne reçue en paramètre pour définir le texte affiché par un label.

```
public partial class SubForm : Form
{
    public SubForm(string message)
    {
        InitializeComponent();
        inputLbl.Text = message;
    }
    // ...
```

Les formulaires sont des classes C# et peuvent donc comporter des attributs en plus des contrôles. La valeur de ces attributs peut être définie dans le constructeur ou par le biais d'accesseurs en écriture (mutateurs).

Récupérer une donnée depuis un formulaire

Les contrôles contenus dans un formulaire sont gérés sous la forme d'attributs privés. On ne peut donc pas y accéder en dehors de la classe.

Pour pouvoir récupérer une propriété d'un contrôle depuis l'extérieur, on peut ajouter à la classe un accesseur qui renvoie cette valeur. Dans l'exemple ci-dessous, la propriété `Input` renvoie le texte saisi dans la **TextBox** nommée `inputBox`.

```
public partial class SubForm : Form
{
    // ...
    public string Input
    {
        get { return inputBox.Text; }
    }
}
```

Ainsi, on peut récupérer et utiliser la valeur saisie dans le formulaire d'origine. L'exemple ci-dessous modifie le titre de la fenêtre du formulaire appelant.

```
SubForm subForm = new SubForm("Entrez votre login");
if (subForm.ShowDialog() == DialogResult.OK)
{
    string login = subForm.Input;
    this.Text = "Bienvenue, " + login;
}
```

Gérer les erreurs

Dans toute application, des évènements imprévus peuvent se produire : absence d'une ressource externe nécessaire (fichier, base de donnée...), bug, etc. Ces évènements se manifestent par la levée d'une exception. Si elle n'est pas gérée, elle se propage et finit par provoquer l'arrêt brutal de l'application.

Une gestion *a minima* des exceptions consiste à placer la ligne `Application.Run(...)` du programme principal dans un bloc `try/catch`. Cela permet de présenter à l'utilisateur un message informatif en cas d'apparition d'une erreur.

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    try
    {
        Application.Run(new MainForm());
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Erreur", MessageBoxButtons.OK, MessageBoxIcon.Err
or);
    }
}
```

> Ce mécanisme agira en dernier recours : dans certains scénarios, il sera plus pertinent d'intercepter les exceptions au plus près de leur apparition potentielle.

Supprimer manuellement un gestionnaire d'évènement

Le fichier `.Designer.cs` associé à un formulaire est normalement géré automatiquement par Visual Studio. Il est cependant parfois nécessaire d'y intervenir manuellement.

Lorsqu'on a généré des gestionnaires d'évènements pour des contrôles avant de les renommer, on aboutit parfois à des gestionnaires d'évènements obsolètes et inutiles dans le fichier `.cs` d'un formulaire.

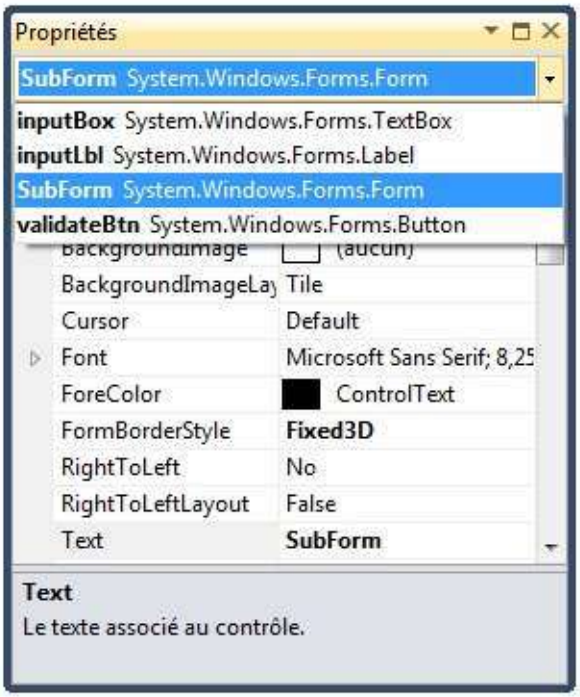
La suppression d'un tel gestionnaire d'évènement se fait en deux étapes :

- Supprimer la ligne qui ajoute le gestionnaire d'évènement au contrôle dans le fichier `.Designer.cs`. Cette ligne est du type `this.nomCtrl.nomEvenement += new System.EventHandler(nomGestionnaire)`.
- Supprimer la méthode correspondant au gestionnaire dans le fichier `.cs`.

Vérifier les noms des contrôles d'un formulaire

Nous avons vu précédemment qu'il était important de renommer systématiquement les contrôles ajoutés à un formulaire.

Pour vérifier tous les noms des contrôles, il suffit d'utiliser la liste déroulante de la zone des propriétés dans le concepteur de formulaire. On détecte immédiatement les contrôles qui n'ont pas été renommés.


Propriétés
SubForm System.Windows.Forms.Form
inputBox System.Windows.Forms.TextBox
inputLbl System.Windows.Forms.Label
SubForm System.Windows.Forms.Form
validateBtn System.Windows.Forms.Button
BackgroundImage (aucun)
BackgroundImageLay Tile
Cursor Default
Font Microsoft Sans Serif; 8,25
ForeColor ControlText
FormBorderStyle Fixed3D
RightToLeft No
RightToLeftLayout False
Text SubForm
Text
Le texte associé au contrôle.

Interactions avec les fichiers

L'objectif de ce chapitre est de présenter plusieurs techniques pour lire et écrire dans des fichiers.

> Les exemples de code de ce chapitre utilisent le framework .NET et ne sont pas liés à la technologie WinForms.

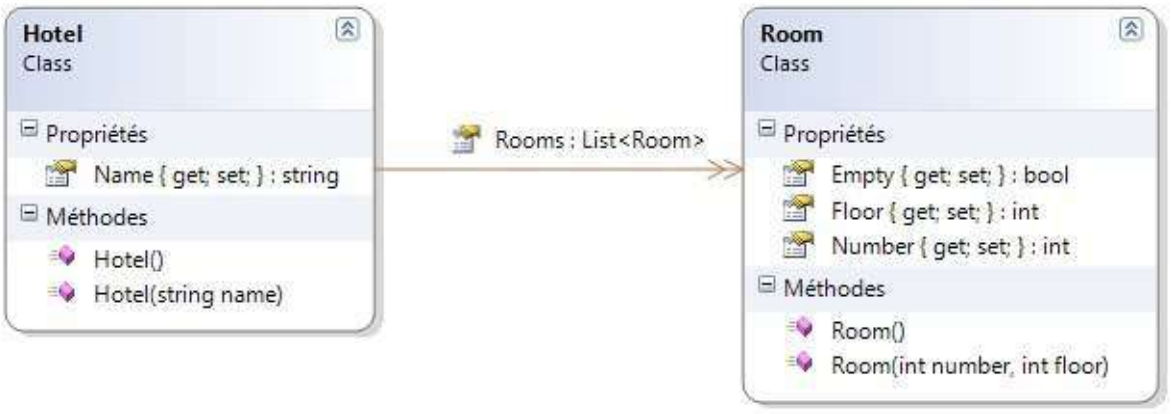
Le concept de sérialisation

La plupart des applications manipulent des données *persistantes* (qui ne disparaissent pas entre deux utilisations de l'application). Pour cela, elles doivent pouvoir sauvegarder et charger leurs données depuis un support de stockage persistant : disque dur ou SSD, clé USB, etc. Une solution simple consiste à y créer un fichier contenant les données de l'application.

On appelle **sérialisation** la transformation d'une information en mémoire sous une forme permettant son stockage persistant ou son transfert.

Contexte d'exemple

Les données métier de note application seront des chambres d'hôtel. Elles sont modélisées sous forme de classes de la manière suivante.



Un hôtel a un nom et comporte une liste de chambres. Chaque chambre possède un numéro, un étage et l'indication de sa disponibilité. Une chambre est vide par défaut.

Voici le code de création des données de test.

```
Hotel hotel = new Hotel("Chelsea Hotel");
hotel.Rooms.Add(new Room(1, 0));
hotel.Rooms.Add(new Room(2, 1));
hotel.Rooms.Add(new Room(3, 1));
hotel.Rooms[2].Empty = false;
```

Sérialisation binaire

La sérialisation binaire consiste à transformer les données en une suite d'octets avec un encodage binaire. Le résultat peut être écrit dans un fichier ou envoyé par le réseau.

Les exemples de code ci-dessous nécessitent l'ajout des directives `using` suivantes.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

Ecriture

Voici un exemple de sérialisation d'un objet vers un fichier binaire.

```
// Sauvegarde l'objet hotel dans le fichier binaire "hotel.dat"
Stream stream = File.Open("hotel.dat", FileMode.Create);
new BinaryFormatter().Serialize(stream, hotel);
stream.Close();
```

Pour que ce code fonctionne, les classes des objets sérialisés (ici `Hotel` et `Room`) doivent indiquer leur caractère sérialisable grâce à l'annotation `[Serializable()]`.

```
[Serializable()]
public class Hotel
{
    // ...
```

```
[Serializable()]
public class Room
{
    // ...
```

> Par défaut, le fichier résultat se trouve dans le même répertoire que l'exécutable de l'application (`bin\Debug` sous Visual Studio).

Lecture

Voici un exemple de désérialisation d'un fichier binaire vers un objet.

```
// Charge le contenu du fichier "hotel.dat" dans l'objet hotel
Stream stream = File.Open("hotel.dat", FileMode.Open);
Hotel hotel = (Hotel) new BinaryFormatter().Deserialize(stream);
stream.Close();
```

Sérialisation XML

La sérialisation XML consiste à transformer les données en une chaîne de caractères au format XML. Ce format a pour avantages d'être lisible par un humain et très interopérable. Son inconvénient majeur est sa verbosité (présence de balises ouvrantes et fermantes pour chaque attribut).

Les exemples de code ci-dessous nécessitent l'ajout des directives `using` suivantes.

```
using System.IO;
using System.Xml;
using System.Xml.Serialization;
```

Ecriture

Voici un exemple de sérialisation d'un objet vers un fichier XML.

```
// Sauvegarde l'objet hotel dans le fichier "hotel.xml"
StreamWriter writer = new StreamWriter("hotel.xml");
new XmlSerializer(typeof(Hotel)).Serialize(writer, hotel);
writer.Close();
```

Pour que ce code fonctionne, les classes des objets sérialisés doivent disposer d'un constructeur sans aucun paramètre.

Le fichier résultat `hotel.xml` contient les données stockées dans les classes métier.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Hotel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Chelsea Hotel</Name>
  <Rooms>
    <Room>
      <Number>1</Number>
      <Floor>0</Floor>
      <Empty>true</Empty>
    </Room>
    <Room>
      <Number>2</Number>
      <Floor>1</Floor>
      <Empty>true</Empty>
    </Room>
    <Room>
      <Number>3</Number>
      <Floor>1</Floor>
      <Empty>false</Empty>
    </Room>
  </Rooms>
</Hotel>
```

> Par défaut, le fichier résultat se trouve dans le même répertoire que l'exécutable de l'application (`bin\Debug` sous Visual Studio).

Lecture

Voici un exemple de désérialisation d'un fichier XML vers un objet.

```
// Charge le contenu du fichier "hotel.xml" dans l'objet hotel
StreamReader reader = new StreamReader("hotel.xml");
Hotel hotel = (Hotel) new XmlSerializer(typeof(Hotel)).Deserialize(reader);
reader.Close();
```

> Vous trouverez plus de détails sur la sérialisation avec le framework .NET à cette adresse et sur la sérialisation XML à cette adresse.

Sérialisation JSON

La sérialisation JSON consiste à transformer les données en une chaîne de caractères au format JSON. Ce format basé sur la notation des objets en JavaScript a les mêmes avantages qu'XML tout en étant moins verbeux. Il est notamment très utilisé pour créer des API web.

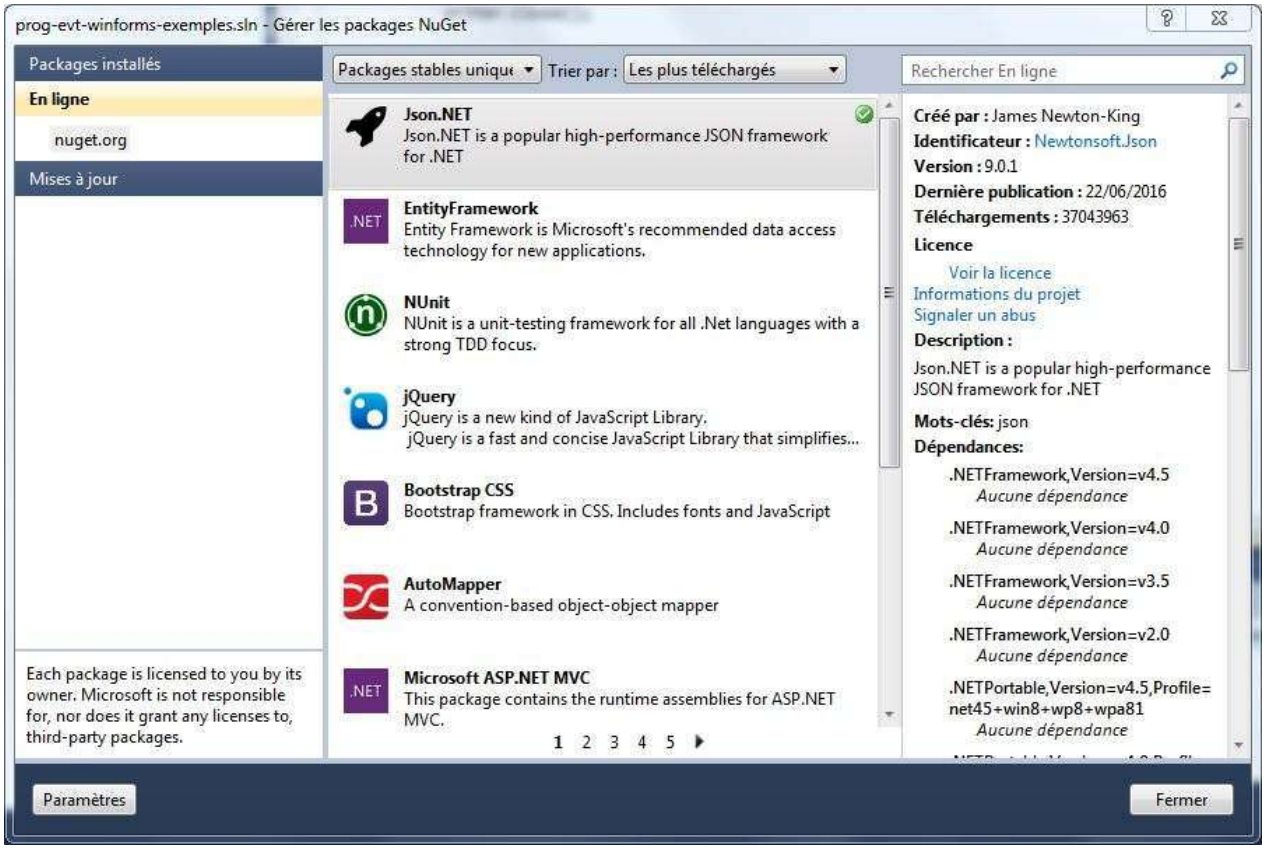
Utilisation d'une librairie externe

Le framework .NET actuel (4.6) ne supporte pas nativement la sérialisation JSON. Pour la mettre en oeuvre, il faut utiliser une librairie externe. La plus connue est Json.NET.

Pour installer cette librairie, nous allons utiliser le gestionnaire de package NuGet. Il s'agit d'une immense collection de librairies répondant à des besoins très variés et la plupart du temps *open source*. En intégrant ces librairies, on gagne du temps tout en évitant de réinventer la roue.

L'installation de NuGet se fait via le menu **Outils/Gestionnaire d'extensions** de Visual Studio.

Une fois NuGet installé, on peut parcourir son contenu (menu **Outils/Gestionnaire de package NuGet**) pour trouver Json.NET, puis l'ajouter à notre projet.



Les exemples de code ci-dessous nécessitent l'ajout des directives `using` suivantes.

```
using System.IO;
using Newtonsoft.Json;
```

Ecriture

Voici un exemple de sérialisation d'un objet vers un fichier JSON.

```
// Sauvegarde l'objet hotel dans le fichier "hotel.json"
StreamWriter writer = new StreamWriter("hotel.json");
new JsonSerializer().Serialize(writer, hotel);
writer.Close();
```

Le fichier résultat `hotel.json` contient les données stockées dans les classes métier.

```
{
    "Name": "Chelsea Hotel",
    "Rooms": [
        {
            "Number": 1,
            "Floor": 0,
            "Empty": true
        },
        {
            "Number": 2,
            "Floor": 1,
            "Empty": true
        },
        {
            "Number": 3,
            "Floor": 1,
            "Empty": false
        }
    ]
}
```

> Par défaut, le fichier résultat se trouve dans le même répertoire que l'exécutable de l'application (`bin\Debug` sous Visual Studio).

Lecture

Voici un exemple de désérialisation d'un fichier JSON vers un objet.

```
// Charge le contenu du fichier "hotel.json" dans l'objet hotel
StreamReader reader = new StreamReader("hotel.json");
Hotel hotel = (Hotel) new JsonSerializer().Deserialize(reader, typeof(Hotel));
reader.Close();
```

WinForms et multithreading

L'objectif de ce chapitre est de comprendre comment faire réaliser plusieurs tâches en parallèle à une application WinForms.

> Il s'agit d'un sujet complexe, uniquement abordé ici dans ses grandes lignes.

La notion de thread

On appelle **thread** (*fil*) un contexte dans lequel s'exécute du code. Chaque application en cours d'exécution utilise au minimum un thread. Une application gérant plusieurs threads est dite *multithread*.

De manière générale, on utilise des threads pour permettre à une application de réaliser plusieurs tâches en parallèle. L'exemple typique est celui d'un navigateur affichant plusieurs onglets tout en téléchargeant un fichier.

Les limites d'une application WinForms monothread

Une application WinForms est basée sur le paradigme de la programmation évènementielle : elle réagit à des évènements provenant du système ou de l'utilisateur. Plus techniquement, elle reçoit et traite en permanence des **messages** provenant du système d'exploitation. Voici quelques exemples de messages :

- Appui sur une touche du clavier.
- Déplacement de la souris.
- Ordre de rafraîchir l'affichage d'une fenêtre.
- ...

Une application WinForms s'exécute dans un thread, appelé thread principal ou thread de l'interface (*UI thread*). Le traitement des messages de l'OS ainsi que l'exécution du code des gestionnaire d'évènement ont lieu dans ce même thread.

Si un gestionnaire d'évènement lance une opération longue (chargement ou sauvegarde réseau, calcul complexe, etc), le traitement des messages de l'OS va ralentir, voire s'arrêter. L'application ne réagira plus aux actions de l'utilisateur et semblera bloquée.

On peut simuler une opération longue en arrêtant le thread courant dans un gestionnaire d'évènement, comme dans l'exemple ci-dessous.

```
private void startMonoBtn_Click(object sender, EventArgs e)
{
    // Opération longue dans le thread principal => blocage de l'application
    infoLbl.Text = "Opération en cours...";
    Thread.Sleep(5000); // Arrête le thread courant pendant 5 secondes
    infoLbl.Text = "Opération terminée";
}
```

Cet exemple ainsi que les suivants nécessitent l'ajout de la directive ci-dessous.

```
using System.Threading;
```

La règle d'or est la suivante : dans une application WinForms, toute opération potentiellement longue doit s'exécuter dans un thread séparé.

Multithreading dans une application WinForms

Le framework .NET permet de manipuler des threads au travers de la classe **Thread**. L'utilisation de cette classe au sein d'une application WinForms pose cependant problème : le code exécuté dans ces threads n'a pas le droit d'accéder aux contrôles des formulaires (c'est un privilège réservé au thread principal). Des solutions de contournement existent mais sont relativement complexes.

Une meilleure alternative consiste à utiliser la classe **BackgroundWorker**. Elle permet de réaliser un traitement dans un thread séparé tout en offrant des mécanismes d'interaction avec le formulaire :

- L'évènement **DoWork** permet de définir le traitement à exécuter dans le thread.
- L'évènement **ProgressChanged** permet de notifier le formulaire de l'avancement du traitement.
- L'évènement **RunWorkerCompleted** signale au formulaire la fin du traitement.

Sa méthode `RunWorkerAsync` démarre un nouveau thread et débute l'exécution du traitement défini dans le gestionnaire de **DoWork**. Les gestionnaires des évènements **ProgressChanged** et **RunWorkerCompleted** peuvent accéder aux contrôles du formulaires afin de mettre à jour celui-ci.

L'exemple de code ci-dessous utilise un **BackgroundWorker** dans lequel on simule une opération longue.

```
private void startMultiBtn_Click(object sender, EventArgs e)
{
    infoLbl.Text = "Opération en cours...";
    worker.RunWorkerAsync(); // Démarre un thread
}

private void worker_DoWork(object sender, DoWorkEventArgs e)
{
    Thread.Sleep(5000); // Arrête le thread courant pendant 5 secondes
}

private void worker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    infoLbl.Text = "Opération terminée";
}
```

Avec cette technique, le thread WinForms principal n'est pas affecté par l'opération en cours et l'application reste réactive.

Multithreading et gestion des erreurs

Si un évènement inattendu se produit dans un thread et provoque la levée d'une exception, celle-ci va entrainer l'arrêt brutal de l'application. Une solution palliative consiste à placer le code du gestionnaire **DoWork** dans un bloc `try/catch`.

```
private void worker_DoWork(object sender, DoWorkEventArgs e)
{
    try
    {
        // Code susceptible de lever des exceptions
        // ...
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Erreur", MessageBoxButtons.OK, MessageBoxIcon.Err
or);
    }
}
```

Pour aller plus loin

Le multithreading est l'un des domaines les plus complexes et les plus intéressants du développement logiciel. Les ressources suivantes pourront vous aider à enrichir vos connaissances en la matière.

- La classe WinForms BackgroundWorker (MSDN)
- Multithreading in WinForms
- Multithreading in .NET

Annexe : exécution de code à intervalles réguliers

Le contrôle WinForms **Timer** permet d'armer une minuterie qui exécute du code à intervalles réguliers.

```
private void countdownBtn_Click(object sender, EventArgs e)
{
    Timer timer = new Timer();
    timer.Tick += new EventHandler(timer_Tick); // timer_Tick est appelé à chaque décl
enchement
    timer.Interval = 1000;                      // Le déclenchement a lieu toutes les
secondes
    timer.Enabled = true;                       // Démarre la minuterie
}

// Code exécuté à chaque déclenchement du timer
void timer_Tick(object sender, EventArgs e)
{
    // ...
}
```

Attention toutefois : l'exécution du code associé au timer se fait dans le thread WinForms principal. Si le traitement associé est trop long, il peut bloquer l'application comme nous l'avons vu plus haut.