Chapitre 11
Création de contrôles utilisateurs

1. Introduction

Le développement d'applications est principalement basé sur les contrôles ; ils fournissent des fonctionnalités distinctes sous une forme visuelle permettant à l'utilisateur d'interagir avec eux. Tous ces contrôles dérivent à un niveau plus ou moins lointain de la classe de base `System.Windows.Forms.Control`. Visual Studio propose l'intégration de contrôles tiers par l'ajout à la boîte à outils. Mais si le besoin est très spécifique, il est possible de créer ses propres contrôles.

La classe de base des contrôles, `Control`, fournit les fonctionnalités de base qui sont nécessaires, notamment pour les entrées utilisateurs via le clavier et la souris. Cela implique donc des propriétés, des méthodes et des événements communs à tous les contrôles. Néanmoins, cette classe de base ne fournit pas la logique d'affichage du contrôle.

Il existe trois modes de création de contrôle :

– Les contrôles personnalisés.
– L'héritage de contrôles.
– Les contrôles utilisateurs.

La création de contrôles s'inscrit dans le principe de réutilisation du code. La logique est créée en un seul endroit et peut être utilisée plusieurs fois. L'avantage est d'autant plus important pour la maintenance d'application car pour changer le comportement de ce contrôle, il n'y aura qu'un fichier à modifier.

2. Les contrôles personnalisés

Ces contrôles offrent les plus grandes possibilités de personnalisation tant au niveau graphique que logique. Un contrôle personnalisé hérite directement de la classe `Control`. Il est donc nécessaire d'écrire toute la logique d'affichage ce qui, suivant le résultat attendu, peut être une phase très longue et compliquée. Les méthodes, propriétés et événements doivent également être définis par le développeur.

La classe de base `Control` expose l'événement `Paint`. C'est celui-ci qui est levé lorsque le contrôle est généré et cela implique l'exécution du gestionnaire de l'événement par défaut `OnPaint`. Cette méthode reçoit un paramètre unique du type `PaintEventArgs` contenant les informations requises sur la surface de dessin du contrôle. Le type `PaintEventArgs` possède deux propriétés, Graphics du type `System.Drawing.Graphics` et `ClipRectangle` du type `System.Drawing.Rectangle`. Pour ajouter la logique de dessin au contrôle, il faut surcharger la méthode `OnPaint` et y ajouter le code de dessin :

```
protected override void OnPaint
                        (System.Windows.Forms.PaintEventArgs e)
{
    // Code de dessin du contrôle
}
```

La propriété `Graphics` de l'objet `PaintEventArgs` représente la surface du contrôle tandis que la propriété `ClipRectangle` représente la zone devant être dessinée. Lors de la première représentation du contrôle, la propriété `ClipRectangle` représente les limites du contrôle. Ces limites peuvent ensuite être modifiées, par exemple si un contrôle au-dessus en cache une partie de telle sorte que le contrôle ait besoin d'être redessiné. La partie `ClipRectangle` représentera la région à modifier.

Créez un dossier **Controls** à la racine du projet et ajoutez une nouvelle classe nommée **CustomControl** définie de la manière suivante :

```
using System.Drawing;

namespace SelfMailer.Controls
{
    public class CustomControl : System.Windows.Forms.Control
    {
        protected override void OnPaint
                        (System.Windows.Forms.PaintEventArgs e)
        {
            Rectangle R = new Rectangle(0, 0,
                            this.Size.Width, this.Size.Height);
            e.Graphics.FillRectangle(Brushes.Green, R);
        }
    }
}
```

Dans le constructeur du formulaire **MailServerSettings**, ajoutez le code d'instanciation du contrôle personnalisé :

```
Controls.CustomControl C = new Controls.CustomControl();
C.Location = new System.Drawing.Point(0, 0);
C.Size = this.Size;
this.Controls.Add(C);
```

Ce code instancie un nouveau contrôle du type `CustomControl`, lui affecte la position en haut à gauche et définit sa taille à celle du formulaire. Pour finir, le contrôle est ajouté à la collection des contrôles du formulaire.

Lancez l'application ([F5]) et ouvrez le formulaire des paramètres de serveur mail pour voir que le contrôle, qui représente un simple rectangle vert, remplit le formulaire comme une couleur de fond.

Remarque

Les possibilités de dessin avec GDI+ seront abordées plus loin dans cet ouvrage, à la section Le dessin avec GDI+ du chapitre Pour aller plus loin.

Il suffit ensuite d'ajouter les membres requis pour la logique du contrôle afin de le finaliser.

3. L'héritage de contrôles

Si le but est d'étendre les fonctionnalités d'un contrôle existant, que ce soit un contrôle du Framework .NET ou d'un éditeur tiers, la manière la plus rapide est d'hériter de ce contrôle. Le nouveau contrôle possède ainsi tous les membres et la représentation visuelle de sa classe parente. Il n'y a plus qu'à rajouter la logique de traitement. Au même titre que les contrôles personnalisés, il reste possible de surcharger la méthode `OnPaint` pour modifier l'aspect visuel du contrôle.

Si une application comporte plusieurs formulaires qui requièrent un e-mail comme champ de saisie, il serait préférable de créer un contrôle héritant de la classe `TextBox` et d'y implémenter la logique de validation puis d'ajouter ce contrôle aux formulaires de manière à ne pas répéter le code de validation dans chacun d'eux.

La création d'un contrôle hérité se fait de la même manière qu'un contrôle personnalisé, en créant une classe qui va hériter du contrôle ayant le comportement de base souhaité. Créez la classe `EmailTextBox` dans le dossier **Controls** et faites-la hériter de la classe `TextBox` :

```
public class EmailTextBox : System.Windows.Forms.TextBox
{
}
```

Ajoutez une surcharge de la méthode `OnValidating` pour effectuer les vérifications sur le format et ajoutez le code de la méthode `FromEmail_Validating` du formulaire **MailServerSettings** :

```
protected override void
          OnValidating(System.ComponentModel.CancelEventArgs e)
{
    base.OnValidating(e);
    string pattern = @"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\." +
                     @"[0-9]{1,3}\.[0-9]{1,3}\.)|" +
                     @"(([a-zA-Z0-9\-]+\.)+))" +
                     @"([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";
    Regex reg = new Regex(pattern);
    if (!reg.IsMatch(this.Text))
    {
        this.BackColor = Color.Bisque;
        e.Cancel = true;
```

```
    }
    else
        this.BackColor = this.PreviousBackColor;
}
```

Des modifications sont à apporter car on n'accède plus à la propriété `Text` du contrôle à partir du formulaire. Il faut donc remplacer :

```
this.FromEmail.Text
```

par un accès direct :

```
this.Text
```

La première instruction :

```
base.OnValidating(e);
```

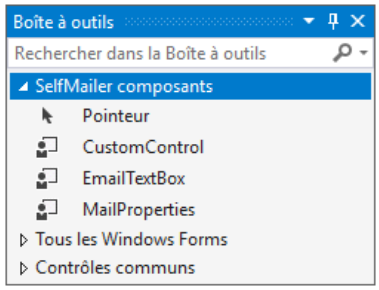
permet d'appeler la méthode de validation de la classe de base. Ainsi, il y a une première validation par la classe de base puis il y a validation du contrôle par le code supplémentaire.

La dernière chose notable est que le composant **ErrorProvider** n'est plus au même niveau. Pour signaler à l'utilisateur que le champ est invalide, au lieu d'afficher une icône, la couleur de fond du contrôle est modifiée. La propriété `PreviousBackColor`, initialisée avec la couleur de fond de base dans le constructeur, permet de la conserver afin de la réaffecter au contrôle en cas de succès de la validation :

```
protected Color PreviousBackColor { get; set; }

public EmailTextBox()
{
    this.PreviousBackColor = this.BackColor;
}
```

Le gestionnaire d'événements `FromEmail_Validating` est devenu inutile puisque la validation se fait au sein du contrôle. De plus, vous pouvez supprimer le contrôle de type `TextBox` pour la saisie de l'e-mail de l'expéditeur et le remplacer par un contrôle de type `EmailTextBox` qui a été ajouté par Visual Studio dans la boîte à outils sous le groupe **SelfMailer composants** (où **SelfMailer** représente le nom du projet).

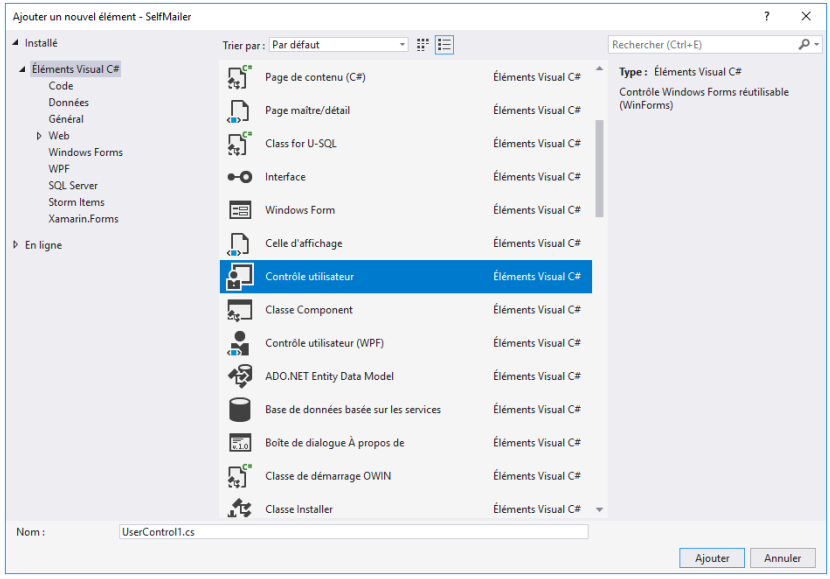


Lancez l'application ([F5]) pour tester la fonctionnalité.

4. Les contrôles utilisateurs

Le but d'un contrôle utilisateur est de regrouper de manière logique des contrôles afin d'obtenir une entité réutilisable. La création se fait par l'ajout au projet d'un **Contrôle utilisateur** depuis la fenêtre d'ajout d'un nouvel élément.

Ajoutez un contrôle utilisateur nommé **MailProperties** dans le dossier **Controls** du projet :



Chapitre 4
Les bases du langage

1. Introduction

Comme tous les langages de programmation, C# impose certaines règles au développeur. Ces règles se manifestent au travers de la syntaxe du langage mais elles couvrent le large spectre fonctionnel proposé par C#. Avant d'explorer en profondeur les fonctionnalités du langage au chapitre suivant, nous étudierons donc les notions essentielles et fondamentales de C# : la création de données utilisables par une application et le traitement de ces données.

2. Les variables

Les données utilisables dans un programme C# sont représentées par des variables. Une variable est un espace mémoire réservé auquel on assigne arbitrairement un nom et dont le contenu est une valeur dont le type est fixé. On peut manipuler ce contenu dans le code en utilisant le nom de la variable.

2.1 Nommage des variables

La spécification du langage C# établit quelques règles à prendre en compte lorsque l'on nomme une variable :

– Le nom d'une variable ne peut comporter que des chiffres, des caractères de l'alphabet latin accentués ou non, le caractère ç ou les caractères spéciaux _ et μ.

– Le nom d'une variable ne peut en aucun cas commencer par un chiffre. Il peut en revanche en comporter un ou plusieurs à toute autre position.

– Le langage est sensible à la casse, c'est-à-dire qu'il fait la distinction entre majuscules et minuscules : les variables `unevariable` et `uneVariable` sont donc différentes.

– La longueur d'un nom de variable est virtuellement illimitée : le compilateur ne remonte pas d'erreur lorsqu'il est composé de 30 000 caractères ! Il n'est évidemment pas recommandé d'avoir des noms de variables aussi longs, le maximum en pratique étant plus souvent de l'ordre de la trentaine de caractères.

– Le nom d'une variable ne peut pas être un mot-clé du langage. Il est toutefois possible de préfixer un mot-clé par un caractère autorisé ou par le caractère @ pour utiliser un nom de variable similaire.

Les noms de variables suivants sont acceptés par le compilateur C# :

– `maVariable`

– `maVariableNumero1`

– `@void` (`void` est un mot-clé de C#)

– `µn3_VàRïãbl3`

De manière générale, il est préférable d'utiliser des noms de variables explicites, c'est-à-dire permettant de savoir à quoi correspond la valeur stockée dans la variable, comme `nomClient`, `montantAchat` ou `ageDuCapitaine`.

2.2 Type des variables

Une des caractéristiques de C# est la notion de typage statique : chaque variable correspond à un type de données et ne peut en changer. De plus, ce type doit être déterminable au moment de la compilation.

2.2.1 Types valeurs et types références

Les différents types utilisables en C# peuvent être décomposés en deux familles : les types valeurs et les types références. Cette notion peut être déconcertante au premier abord, puisqu'une variable représente justement une donnée et donc une valeur. Ce concept est en fait lié à la manière dont est stockée l'information en mémoire.

Lorsque l'on utilise une variable de type valeur, on accède directement à la zone mémoire stockant la donnée. Au moment de la création d'une variable de type valeur, une zone mémoire de la taille correspondant au type est allouée. Chaque octet de cette zone est automatiquement initialisé avec la valeur binaire 00000000. Notre variable aura donc une suite de 0 pour valeur binaire.

Dans le cas d'une variable de type référence, le comportement est différent. La zone mémoire allouée à notre variable contient une adresse mémoire à laquelle est stockée la donnée. On passe donc par un intermédiaire pour accéder à notre donnée. L'adresse mémoire est initialisée avec la valeur spéciale `null`, qui ne pointe sur rien, tandis que la zone mémoire contenant les données n'est pas initialisée. Elle sera initialisée lorsque la variable sera instanciée. Dans le même temps, l'adresse mémoire stockée dans notre variable sera mise à jour.

Une variable de type référence pourra donc ne contenir aucune donnée, tandis qu'une variable de type valeur aura forcément une valeur correspondant à une suite de 0 binaires.

Cette différence de fonctionnement a une conséquence importante : la copie de variable se fait par valeur ou par référence, ce qui signifie qu'une variable de type valeur sera effectivement copiée, tandis que pour un type référence, c'est l'adresse que contient la variable qui sera copiée, et il sera donc possible d'agir sur la donnée réelle indifféremment à partir de chacune des variables pointant sur ladite donnée.

2.2.2 Types intégrés

La plateforme .NET embarque plusieurs milliers de types différents utilisables par les développeurs. Parmi ces types, nous en avons une quinzaine que l'on peut considérer comme fondamentaux : ce sont les types intégrés (aussi nommés types primitifs). Ce sont les types de base à partir desquels sont construits les autres types de la BCL ainsi que ceux que le développeur crée dans son propre code. Ils permettent de définir des variables contenant des données très simples.

Ces types ont comme particularité d'avoir chacun un alias intégré à C#.

Types numériques

Ces types permettent de définir des variables numériques entières ou décimales. Ils couvrent des plages de valeurs différentes et ont chacun une précision spécifique. Certains types seront donc plus adaptés pour les calculs entiers, d'autres pour les calculs dans lesquels la précision décimale est très importante, comme les calculs financiers.

Les différents types numériques sont énumérés ci-dessous avec leurs alias ainsi que les plages de valeurs qu'ils couvrent.

Type	Alias C#	Plage de valeurs couverte	Taille en mémoire
`System.Byte`	`byte`	0 à 255	1 octet
`System.SByte`	`sbyte`	-128 à 127	1 octet
`System.Int16`	`short`	-32768 à 32767	2 octets
`System.UInt16`	`ushort`	0 à 65535	2 octets
`System.Int32`	`int`	-2147483648 à 2147483647	4 octets
`System.UInt32`	`uint`	0 à 4294967295	4 octets
`System.Int64`	`long`	-9223372036854775808 à 9223372036854775807	8 octets
`System.UInt64`	`ulong`	0 à 18446744073709551615	8 octets
`System.Single`	`float`	±1,5e-45 à ±3,4e38	4 octets

Type	Alias C#	Plage de valeurs couverte	Taille en mémoire
`System.Double`	`double`	±5,0e-324 à ±1,7e308	8 octets
`System.Decimal`	`decimal`	±1,0e-28 à ±7,9e28	16 octets

Les types numériques primitifs sont tous des types valeurs. Une variable de type numérique et non initialisée par le développeur aura pour valeur par défaut 0.

Remarque

.NET 4 a apporté le type `System.Numerics.BigInteger` afin de manipuler des entiers d'une taille arbitraire. Ce type est aussi un type valeur, mais il ne fait pas partie des types intégrés.

Les valeurs numériques utilisées au sein du code peuvent être définies à l'aide de trois bases numériques :

– **Décimale** (base 10) : c'est la base numérique utilisée par défaut.

  Exemple : 280514

– **Héxadécimale** (base 16) : elle est très utilisée dans le monde du Web, pour la définition de couleurs, et plus généralement pour encoder des valeurs numériques dans un format plus condensé. **Une valeur héxadécimale est écrite en la préfixant par 0x.**

  Exemple : 0x0447C2

– **Binaire** (base 2) : cette base est celle qui permet de s'approcher au plus près de la manière dont la machine interprète le code compilé. L'utilisation du binaire en C# peut par conséquent se révéler précieuse dans le cadre d'optimisations (décalages de bits au lieu de multiplications par 2, stockage de multiples données au sein d'une même variable, etc.). **Les valeurs binaires sont préfixées par 0b.**

  Exemple : 0b01000100011111000010

La lecture de valeurs numériques peut s'avérer difficile dans certains cas, notamment lorsqu'elles sont représentées sous forme binaire. L'équipe en charge des évolutions de C# a intégré, dans la septième version du langage, la possibilité d'ajouter des séparateurs dans les valeurs numériques afin d'améliorer la lisibilité globale. Le caractère utilisé pour cela est _ :

– 280_514
– 0x04_47_C2
– 0b0100_0100_0111_1100_0010

Types textuels

Il existe dans la BCL deux types permettant de manipuler des caractères Unicode et des chaînes de caractères Unicode : `System.Char` et `System.String`. Ces types ont respectivement pour alias `char` et `string`.

Le type `char` est un type valeur encapsulant les mécanismes nécessaires au traitement d'un caractère Unicode. Par conséquent, une variable de type char est stockée en mémoire sur deux octets.

Les valeurs de type `char` doivent être encadrées par les caractères ' ' : 'a'.

Certains caractères ayant une signification particulière pour le langage doivent être utilisés avec le caractère d'échappement \ afin d'être correctement interprétés. D'autres caractères n'ayant pas de représentation graphique doivent être aussi déclarés avec des séquences spécifiques. Le tableau suivant résume les séquences qui peuvent être utilisées.

Séquence d'échappement	Caractère associé
\'	Simple quote '
\"	Double quote "
\\	Backslash \
\a	Alerte sonore
\b	Retour arrière
\f	Saut de page
\n	Saut de ligne