INFORMATIQUE
TECHNIQUE

Expert

C# 8

Développez des applications Windows avec Visual Studio 2019

En téléchargement

</> des exemples de code

+ QUIZ

Version en ligne **OFFERTE !** pendant 1 an

Jérôme HUGON

eni

Table des matières

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence de l'ouvrage **EI8C19VIS** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Chapitre 3
Introduction au langage C#

Chapitre 4
La création de types

Chapitre 5
L'héritage

Chapitre 6
Types génériques

Chapitre 7
Délégués, événements et expressions lambda

Chapitre 8
Création de formulaires

Chapitre 9
Implémentation de gestionnaires d'événements

Chapitre 10
Validation de la saisie

Chapitre 11
Création de contrôles utilisateurs

Chapitre 12
Création d'applications UWP

Chapitre 13
Débogage

Chapitre 14
Gestion des exceptions

Chapitre 15
Monitoring

Chapitre 19
Présentation de LINQ

Chapitre 20
LINQ to Entities

Chapitre 21
LINQ to SQL

Chapitre 22
LINQ to XML

Chapitre 23
Le système de fichiers

Chapitre 24
Sérialisation

Chapitre 25
Expressions régulières

Chapitre 26
Multithreading

Chapitre 27
Globalisation et localisation

Chapitre 28
Sécurité

Chapitre 29
Pour aller plus loin

Chapitre 30
Assemblages et configurations

Chapitre 31
Déploiement

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);
```

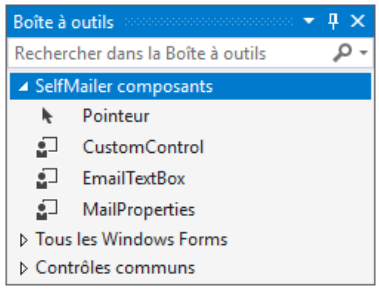
permet d'appeler la méthode de validation de la classe de base. Ainsi, il y a une première validation par la classe de base puis il y a validation du contrôle par le code supplémentaire.

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 :