

Création d'un popup

par Anthony DE DECKER ([Accueil](#))

Date de publication :

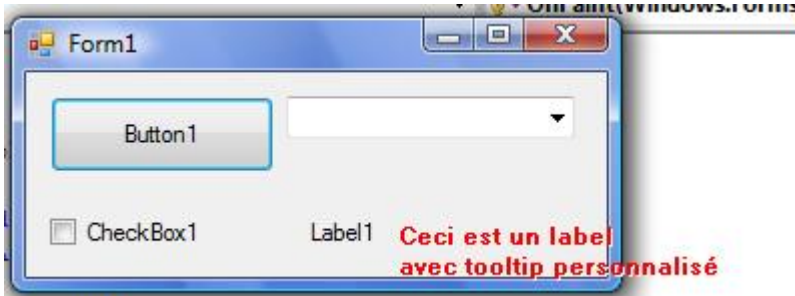
Dernière mise à jour : 03/03/2008

Ce tutoriel présente l'utilisation d'un popup dans le cadre de la création d'un tooltip personnalisé.

I - Introduction.....	3
II - Présentation du Popup.....	4
II-A - Définition.....	4
II-B - Qu'en tirons nous ?.....	4
III - Gestion du Popup.....	5
III-A - Définir le parent.....	5
III-B - Afficher le Popup.....	5
III-C - Masquer le Popup.....	6
III-D - Interaction avec l'utilisateur.....	6
III-E - Interaction avec le créateur.....	6
III-F - Illustration.....	6
III-F-1 - Qu'allons nous faire ?.....	6
III-F-2 - Comment ?.....	7
III-F-3 - Allons plus loin.....	9
IV - Utilisation dans le cadre d'un Tooltip personnalisé.....	11
IV-A - Introduction.....	11
IV-B - Définition du Popup.....	11
IV-C - Création du ToolTip.....	14
IV-C-1 - Définition de la classe d'extension du ToolTip.....	14
IV-C-2 - Création du ToolTip.....	15
IV-C-3 - Déclaration d'un TypeConverter pour la propriété d'extension.....	17
IV-C-4 - Résultat.....	20
IV-F - Exemple d'utilisation.....	21
IV-E - Limites.....	21
V - Remerciements.....	23

I - Introduction

Dans ce tutoriel, nous allons créer un Tooltip personnalisé utilisant pour son affichage un Popup indépendant de l'application dans laquelle il est utilisé.

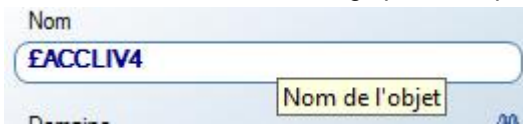


Ce tutoriel est basé sur l'utilisation du Framework 2.0 et du langage VB.Net

II - Présentation du Popup

II-A - Définition

Le popup est par définition un fenêtre s'affichant en premier plan (au dessus de toutes les autres) et destinée à afficher de l'information. Il s'agit par exemple du petit texte d'aide qui s'affiche lorsque la souris survole un **Control** :



Le popup peut toutefois interagir avec l'utilisateur ou un autre control. C'est par exemple le cas de la **Dropdownlist** des **Combobox** qui accepte des actions de la part de l'utilisateur et les répercute sur le **Control Combobox**.

De cette définition, il ressort qu'un popup est un objet disposant d'une méthode d'affichage et potentiellement de méthodes lui permettant l'interaction utilisateur/control parent.

En .NET, il est possible de créer un **popup à partir de n'importe quel objet " affichable "** (capable de se dessiner à l'écran).

On trouve ainsi des exemples de popup directement basés sur des objets héritant de **NativeWindows**. Ce type de popup nécessite d'implémenter "à la main" la méthode d'affichage de l'objet.

*A titre personnel, et partant du principe que tout objet **Control** gère déjà son dessin via la méthode **.OnPaint**, je préfère utiliser soit l'objet **Control**, soit un objet qui en hérite pour la création de popup personnalisé.*

Le comportement d'un popup doit également respecter certaines règles d'affichage définies par l'application (ou l'objet) qui le contient. Par exemple dans le cas d'une application MDI, le popup ne doit pas s'afficher en dehors de la fenêtre MDI de cette application. Dans tous les cas, le popup est au moins soumis aux limites fixées par le "bureau" windows.

II-B - Qu'en tirons nous ?

De la définition du popup, il ressort que pour créer un popup, nous devons savoir :

- lui fixer des limites en définissant son parent (bureau, application MDI, formulaire, panel, etc...),
- afficher une fenêtre pour ce popup,
- dessiner le contenu de cette fenêtre,
- interagir avec l'utilisateur ou l'objet ayant créé le popup,
- et bien sûr, une fois le popup inutile, masquer (ou détruire) cette fenêtre.

III - Gestion du Popup

La gestion de l'affichage du popup (définition du parent et affichage proprement dit) et de sa destruction est réalisé via les API mises à disposition par windows.

⚠ Il s'agit ici d'utiliser des API, donc du code non managé par le Framework. Ceci implique une vigilance toute particulière quant à la libération des ressources utilisées par ces API.

III-A - Définir le parent

La définition du parent de la fenêtre popup est réalisée via l'API **SetParent** :

Utilisation de SetParent

```
Private Declare Function SetParent Lib "user32" ( _
    ByVal hWndChild As IntPtr, _
    ByVal hWndNewParent As IntPtr) As Integer

Private Sub MySubSetParent()
    ' Positionnement du parent indispensable
    If MyParent Is Nothing Then
        ' Le parent est le "bureau"
        SetParent(MyBase.Handle, IntPtr.Zero)
    Else
        ' Le popup doit rester dans les limites du parent
        SetParent(MyBase.Handle, MyParent.Handle)
    End If
End Sub
```

*Nota : ici on considère que MyParent est un objet disposant d'un handle, potentiellement un **Control** passé via la sub **new()** du popup (comme nous le verrons ultérieurement).*

III-B - Afficher le Popup

L'affichage de la fenêtre popup est réalisé via l'API **ShowWindow** :

Utilisation de ShowWindow

```
Private Declare Function ShowWindow Lib "user32" ( _
    ByVal hWnd As IntPtr, _
    ByVal nCmdShow As Integer) As Integer
Private Const SW_SHOWNOACTIVATE As Integer = 4

Private Sub MySubShowWindow()
    ' Affichage
    ShowWindow(MyBase.Handle, SW_SHOWNOACTIVATE)
End Sub
```

*Nota : le **SW_SHOWNOACTIVATE** indique ici que la fenêtre popup doit être affichée sans être activée.*

L'utilisation du **ShowWindow** déclenchera bien l'affichage du Popup.

Cependant, celui-ci étant une fenêtre comme une autre, il apparaîtra logiquement dans la barre des tâches, ce qui est assez déplaisant pour un popup.

Pour interdire cet affichage en barre des tâches, il est nécessaire de modifier le style étendu (**ExStyle**) de notre popup. Ceci est réalisé en surchargeant la propriété **CreateParams** du control :

Interdire l'affichage en barre des tâches

Interdire l'affichage en barre des tâches

```

Private Const WS_EX_TOOLWINDOW As Integer = &H80
Protected Overrides ReadOnly Property CreateParams() _
    As System.Windows.Forms.CreateParams
    Get
        Dim p As CreateParams = MyBase.CreateParams
        p.ExStyle = WS_EX_TOOLWINDOW
        p.Parent = IntPtr.Zero
        Return p
    End Get
End Property
    
```

*Nota : **WS_EX_TOOLWINDOW** permet de ne pas afficher le popup en barre des tâches. Je vous renvoie ici pour plus d'informations (très appréciables) sur les styles étendus : <http://msdn2.microsoft.com/en-us/library/aa930455.aspx>*

III-C - Masquer le Popup

Masquer une fenêtre peut être effectué via l'API **ShowWindow** en utilisant le flag **SW_HIDE** (0). Toutefois, dans le cas d'un popup, si son affichage devient inutile, la destruction de la fenêtre est plus appropriée. Cette destruction est réalisée via l'API **DestroyWindow** :

Utilisation de DestroyWindow

```

Private Declare Function DestroyWindow Lib "user32.dll" ( _
    ByVal hWnd As IntPtr) As Boolean

Private Sub MySubDestroyWindow()
    ' Destruction de la fenêtre
    DestroyWindow(MyBase.Handle)
End Sub
    
```

III-D - Interaction avec l'utilisateur

L'interaction avec l'utilisateur dépend de l'objet que l'on affiche sous forme de popup. Ainsi, comme nous le verrons, un popup basé sur l'utilisation d'un **Button** interagira avec l'utilisateur de la même façon qu'un **Button** positionné sur une **Form** (ou tout type de **Container**).

III-E - Interaction avec le créateur

Le popup peut interagir avec l'objet l'ayant créé soit par l'utilisation d'évènements ou directement par l'utilisation de méthode ou propriété public de cet objet. Ces deux possibilités seront mises en évidence dans les exemples suivants.

III-F - Illustration

III-F-1 - Qu'allons nous faire ?

Pour illustrer la création d'un popup simple, nous allons utiliser un popup issu d'un **Control** de type **Button**. Ce popup sera créé lors d'un clic sur le bouton d'un formulaire. Un second clic sur ce même bouton entraînera la fermeture du popup. Un clic sur le popup "bouton" changera la couleur de fond du formulaire.

III-F-2 - Comment ?

Pour ce faire, nous allons créer une **Class** héritant de **Button** dans laquelle nous allons gérer l'affichage et la destruction du popup (ce que nous avons vu précédemment) :

```

Option Strict On
Option Explicit On

Public Class UltraBasicPopup
    Inherits Button

#Region "Constantes"
    Private Const WS_EX_TOOLWINDOW As Integer = &H80
    Private Const SW_SHOWNOACTIVATE As Integer = 4
#End Region

    Protected Overrides ReadOnly Property CreateParams() _
        As System.Windows.Forms.CreateParams
    Get
        Dim p As CreateParams = MyBase.CreateParams
        p.ExStyle = WS_EX_TOOLWINDOW
        p.Parent = IntPtr.Zero
        Return p
    End Get
End Property

Private Declare Function SetParent Lib "user32" ( _
    ByVal hWndChild As IntPtr, _
    ByVal hWndNewParent As IntPtr) As Integer
Private Declare Function ShowWindow Lib "user32" ( _
    ByVal hWnd As IntPtr, _
    ByVal nCmdShow As Integer) As Integer
Private Declare Function DestroyWindow Lib "user32.dll" ( _
    ByVal hWnd As IntPtr) As Boolean
End Class
    
```

A ce niveau, nous ne pouvons pas piloter l'affichage ou la destruction du popup. Nous allons donc ajouter deux méthodes publiques :

- **ShowPopup** : qui déclenchera l'affichage
- **HidePopup** : qui détruira la fenêtre.

Avant de pouvoir afficher, nous devons toutefois définir la taille de notre popup ainsi que sa position. Nous fixerons la position du popup à partir de la position du control ayant demandé son affichage (et qui devra être passé en paramètre de la méthode **ShowPopup**) et nous mettrons également à disposition une propriété publique **Opened** permettant à ce control de déterminer si le popup est ouvert ou fermé.

```

#Region "declaration"
    Private cControlParent As Control
    Private blnOpened As Boolean
#End Region

#Region "Propriétés"
    Public ReadOnly Property Opened() As Boolean
    Get
        Return blnOpened
    End Get
End Property
#End Region

Private Sub ClosePopup()
    DestroyWindow(MyBase.Handle)
End Sub
    
```

```

        blnOpened = False
    End Sub

    Private Sub OpenPopup()
        SetParent(MyBase.Handle, IntPtr.Zero)
        ' Affichage
        ShowWindow(MyBase.Handle, SW_SHOWNOACTIVATE)
        blnOpened = True
    End Sub

    Public Sub ShowPopup(ByVal IControlParent As Control)

        cControlParent = IControlParent
        Me.SetLocation()
        Me.Size = New Size(150, 30)
        Me.Region = New Region( _
            New Rectangle(3, 3, Me.Size.Width - 6, Me.Size.Height - 6))
        Me.Text = "Youpi quel beau popup !"
        Me.OpenPopup()

    End Sub

    Public Sub HidePopup()

        Me.ClosePopup()

    End Sub

    Private Sub SetLocation()

        Dim rParent As Rectangle = _
Me.cControlParent.RectangleToScreen(Me.cControlParent.ClientRectangle)
        Dim ptLocation As New Point( _
            rParent.X + rParent.Width + 10, _
            rParent.Y + rParent.Height + 10)
        Me.Location = ptLocation

    End Sub
    
```

Pour les curieux : Pourquoi avoir redéfini la **Region** du bouton ?

Tout simplement car le **Button** de base ne peint pas l'intégralité de son **Bounds** dans sa méthode **OnPaint**. Tout ce qui se trouve en dehors de la bordure du bouton est peint lors dans la méthode **OnPaintBackGround** et la couleur utilisée est déterminée à partir du control parent du bouton.

Comme ici notre **Button** n'a pas de control parent, ce qui se trouve en dehors de la bordure du **Button** ne peut pas être dessiné correctement.

Il ne nous reste plus qu'à modifier la couleur de fond de la **Form** contenant le control ayant demandé l'affichage via la méthode **OnClick** :

```

    Private cColor1 As Color = Color.Transparent
    Private cColor2 As Color = Color.Red
    Protected Overrides Sub OnClick( _
        ByVal e As EventArgs)

        MyBase.OnClick(e)

        Dim f As Form = cControlParent.FindForm
        If f Is Nothing Then Exit Sub

        Dim c As Color = f.BackColor

        If c = cColor1 Or cColor1 = Color.Transparent Then
            cColor1 = f.BackColor
            f.BackColor = cColor2
        Else
            f.BackColor = cColor1
        End If
    End Sub
    
```

End Sub

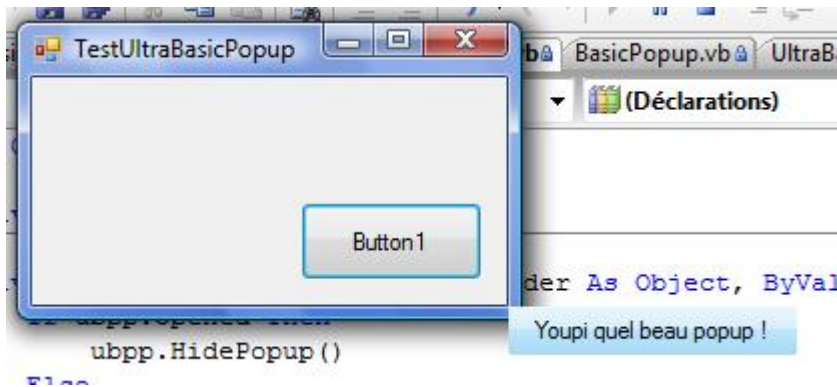
Créons maintenant une **Form** contenant un **Button** qui affiche ou détruit le popup :

```
Public Class TestUltraBasicPopup
    Private ubpp As New UltraBasicPopup

    Private Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) _
Handles Button1.Click
        If ubpp.Opened Then
            ubpp.HidePopup()
        Else
            ubpp.ShowPopup(Me.Button1)
        End If
    End Sub
End Class
```

Résultat :

Lors d'un clic sur le bouton du formulaire, le popup apparaît :



Un clic sur le popup entraîne le changement de couleur de la form.

III-F-3 - Allons plus loin

Ceux qui ont testé le code plus haut ont sans doute constaté les limites de ce Popup (d'où son nom !). En effet, notre popup ne disparaît pas tant que le formulaire est ouvert ou que le bouton de ce formulaire n'a pas été à nouveau cliqué.

Pour pallier cela, il convient d'ajouter une demande de fermeture du popup lors de la désactivation du formulaire :

```
Public Class TestUltraBasicPopup
    Private ubpp As New UltraBasicPopup

    Private Sub Button1_Click _
(ByVal sender As Object, ByVal e As System.EventArgs) Handles Button1.Click
        If ubpp.Opened Then
            ubpp.HidePopup()
        Else
            ubpp.ShowPopup(Me.Button1)
        End If
    End Sub

    Private Sub TestUltraBasicPopup_Deactivate _
(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Deactivate
```

```
If ubpp.Opened Then ubpp.HidePopup()  
End Sub  
  
End Class
```

C'est mieux mais on constate que retailer la **Form** ou la déplacer n'a pas d'impact sur le popup.

Dans l'article [Principe du Hook et utilisation d'un Hook souris](#) nous verrons comment l'utilisation d'un Hook sur la souris nous permettra de résoudre ces problèmes.

IV - Utilisation dans le cadre d'un Tooltip personnalisé

IV-A - Introduction

Nous allons ici créer un Tooltip (hérité de **Component**) qui utilisera un popup pour l'affichage.

Ce Tooltip permettra, pour chaque control de sa **Form** parente, de définir un texte à afficher lors du passage de la souris sur ce **Control**.

Pour gérer l'affichage, le popup utilisera les propriétés suivantes :

- **.BackColor** : définissant la couleur de fond
- **.Font** : définissant la police à utiliser
- **.ForeColor** : définissant la couleur du texte
- **.ShadowForeColor** : définissant la couleur de l'ombre du texte
- **.Text** : définissant le texte

Pour définir ces propriétés, le **Tooltip** étendra chaque **Control** en lui adjoignant une propriété **BasicToolTip**.

IV-B - Définition du Popup

A partir de ce que nous avons vu précédemment, nous définissons une classe BasicPopup :

Définition du Popup pour le Tooltip

```
Option Strict On
Option Explicit On
Imports System.ComponentModel
<DesignTimeVisible(False)> _
Public Class BasicPopup
    Inherits Control

    #Region "Constantes"
        Private Const WS_EX_TOOLWINDOW As Integer = &H80
        Private Const SW_SHOWNOACTIVATE As Integer = 4
    #End Region

    Protected Overrides ReadOnly Property CreateParams() _
        As System.Windows.Forms.CreateParams
        Get
            Dim p As CreateParams = MyBase.CreateParams
            p.ExStyle = WS_EX_TOOLWINDOW
            p.Parent = IntPtr.Zero
            Return p
        End Get
    End Property

    Private Declare Function SetParent Lib "user32" ( _
        ByVal hWndChild As IntPtr, _
        ByVal hWndNewParent As IntPtr) As Integer
    Private Declare Function ShowWindow Lib "user32" ( _
        ByVal hWnd As IntPtr, _
        ByVal nCmdShow As Integer) As Integer
    Private Declare Function DestroyWindow Lib "user32.dll" ( _
        ByVal hWnd As IntPtr) As Boolean

    #Region "declaration"
        Private cControlParent As Control
        Private blnOpened As Boolean
        Private cFormParent As Form
        Private strText As String
        Private cBackColor As Color
    #End Region
End Class
```

Définition du Popup pour le Tooltip

```

Private cShadowForeColor As Color = Color.WhiteSmoke
#End Region

#Region "Graphique"
Public Shadows Property BackColor() As Color
    Get
        Return cBackColor
    End Get
    Set(ByVal value As Color)
        cBackColor = value
    End Set
End Property
Public Property ShadowForeColor() As Color
    Get
        Return cShadowForeColor
    End Get
    Set(ByVal value As Color)
        cShadowForeColor = value
    End Set
End Property

Protected Overrides Sub OnPaint( _
ByVal e As PaintEventArgs)

    MyBase.OnPaint(e)

    Dim bufferImage As Bitmap
    bufferImage = New Bitmap(Me.Width, Me.Height)
    Dim gd As Graphics = Graphics.FromImage(bufferImage)
    gd.SmoothingMode = Drawing2D.SmoothingMode.HighQuality
    gd.TextRenderingHint = Drawing.Text.TextRenderingHint.AntiAlias

    If Not Me.BackColor = Color.Transparent Then
        gd.FillRectangle(New SolidBrush(Me.BackColor), Me.ClientRectangle)
    End If

    Dim intShadowOffset As Integer = CInt(Me.Font.Size / 20)

    gd.DrawString(strText, Me.Font, New SolidBrush(Me.ShadowForeColor), _
New Point(intShadowOffset, intShadowOffset))
    gd.DrawString(strText, Me.Font, New SolidBrush(Me.ForeColor), New Point(0, 0))
    e.Graphics.DrawImage(bufferImage, 0, 0)

    bufferImage.Dispose()
    gd.Dispose()

End Sub

Protected Overrides Sub OnPaintBackground( _
ByVal pevent As PaintEventArgs)
    ' rien
End Sub
#End Region

Private Sub ClosePopup()

    DestroyWindow(MyBase.Handle)
    blnOpened = False

End Sub

Private Sub OpenPopup()

    blnOpened = True

    ' Positionnement du parent indispensable
    If cFormParent.MdiParent Is Nothing Then
        SetParent(MyBase.Handle, IntPtr.Zero)
    Else
        ' Le popup doit rester dans les limites de la form mdi
    
```

Définition du Popup pour le Tooltip

```

        SetProperty(MyBase.Handle, cFormParent.MdiParent.Handle)
    End If

    ' Affichage
    ShowWindow(MyBase.Handle, SW_SHOWNOACTIVATE)

End Sub

Public Sub HidePopup()

    If Not blnOpened Then Exit Sub

    Me.ClosePopup()

End Sub

Public Sub ShowPopup(ByVal IControlParent As Control, ByVal IText As String)

    If blnOpened Then Exit Sub

    strText = IText

    cControlParent = IControlParent

    cFormParent = IControlParent.FindForm()

    If (Me.Handle.Equals(IntPtr.Zero)) Then
        MyBase.CreateControl()
    End If

    Me.Size = TextRenderer.MeasureText(strText, Me.Font)
    Me.Width += 1
    Me.Height += 1

    SetLocation()

    Me.OpenPopup()

End Sub

Private Sub SetLocation()

    Dim ptLocation As Point = Control.MousePosition

    Dim rParent As Rectangle = _
Me.cControlParent.RectangleToScreen(Me.cControlParent.ClientRectangle)

    If ptLocation.X > rParent.X And ptLocation.X < rParent.X + rParent.Width Then
        ptLocation.X = rParent.X + rParent.Width + 5
    Else
        If ptLocation.Y > rParent.Y And _
ptLocation.Y < rParent.Y + rParent.Height Then
            ptLocation.Y = rParent.Y + rParent.Height + 5
        End If
    End If

    Me.Location = ptLocation

End Sub
End Class

```

Le popup héritant de **Control**, nous n'aurions besoin d'ajouter que la propriété **.ShadowForeColor**, les propriétés **.Text**, **.ForeColor**, **.BackColor** et **.Font** étant directement héritées de **Control**.

Cependant, la propriété **.BackColor** de base est masquée pour pouvoir la définir à **Color.Transparent** sans devoir positionner le bit **SupportsTransparentBackColor**.

Ceci est en effet sans intérêt car le **Paint** du **Control** est entièrement géré.

Nota :

Nous passons dans le **OnPaint** par un **Graphics.DrawString**, car La qualité du rendu du **TextRenderer.DrawText** est plus que déplorable dans ce cadre d'utilisation.

L'attribut **DesignTimeVisible(False)** permet de rendre le **Control** non disponible dans la boîte à outils du designer.

IV-C - Création du Tooltip

IV-C-1 - Définition de la classe d'extension du Tooltip

Il s'agit ici de créer une classe d'objet contenant l'ensemble des informations nécessaires à l'affichage du popup : **.ForeColor**, **.Text**, etc ...

Cette classe sera utilisée pour étendre les propriétés de chacun des **Controls** du formulaire.
(Nous verrons cela dans le paragraphe suivant)

Classe d'extension des controls utilisée par le Tooltip

```

Option Strict On
Option Explicit On
Imports System.ComponentModel
Imports System.ComponentModel.Design.Serialization
<Serializable()> _
Public Class BasicToolTipExtendedProperty

    Private strText As String
    Private cBackColor As Color = Color.Transparent
    Private cForeColor As Color = Color.Red
    Private cShadowForeColor As Color = Color.WhiteSmoke
    Private fFont As Font = New Font("Microsoft Sans Serif", 12, FontStyle.Bold, _
GraphicsUnit.Pixel)

    ' Ajouter System.Design aux références
    <Editor(GetType(System.ComponentModel.Design.MultilineStringEditor), _
GetType(System.Drawing.Design.UITypeEditor))> _
    Public Property Text() As String
        Get
            Return strText
        End Get
        Set(ByVal value As String)
            strText = value
        End Set
    End Property

    ''' <summary>
    ''' Couleur de fond du Tooltip
    ''' Color.Transparent définit un Tooltip sans couleur de fond.
    ''' </summary>
    ''' <remarks></remarks>
    <Description("Couleur de fond du Tooltip." & ControlChars.CrLf & _
"Color.Transparent définit un Tooltip sans couleur de fond.")> _
    Public Property BackColor() As Color
        Get
            Return cBackColor
        End Get
        Set(ByVal value As Color)
            cBackColor = value
        End Set
    End Property

    ''' <summary>
    ''' Couleur du texte du Tooltip
    ''' </summary>
    ''' <remarks></remarks>
    <Description("Couleur du texte du Tooltip.")> _
    Public Property ForeColor() As Color
        Get
            Return cForeColor
        End Get
        Set(ByVal value As Color)
            cForeColor = value
        End Set
    End Property

```

Classe d'extension des controls utilisée par le Tooltip

```

''' <summary>
''' Couleur de l'ombre du texte du Tooltip
''' </summary>
''' <remarks></remarks>
<Description("Couleur de l'ombre du texte du Tooltip.")> _
Public Property ShadowForeColor() As Color
    Get
        Return cShadowForeColor
    End Get
    Set(ByVal value As Color)
        cShadowForeColor = value
    End Set
End Property
''' <summary>
''' Police du texte du Tooltip
''' </summary>
''' <remarks></remarks>
<Description("Police du texte du Tooltip.")> _
Public Property Font() As Font
    Get
        Return fFont
    End Get
    Set(ByVal value As Font)
        fFont = value
    End Set
End Property

''' <summary>
''' Texte affiché par le Tooltip
''' </summary>
''' <remarks></remarks>
<Description("Texte affiché par le Tooltip.")> _
Public Sub New(ByVal text As String, _
ByVal foreColor As Color, _
ByVal shadowForeColor As Color, _
ByVal backColor As Color, _
ByVal font As Font)
    strText = text
    cBackColor = backColor
    cForeColor = foreColor
    cShadowForeColor = shadowForeColor
    fFont = font
End Sub

Public Sub New()
End Sub

End Class
    
```

L'attribut **Serializable** entraînera la sérialisation de l'objet dans le fichier ressource de la **Form** utilisant le Tooltip. Il n'a rien d'obligatoire.

L'attribut **Editor** de la propriété **.Text** permettra l'utilisation dans le designer d'un control d'édition multilignes pour cette propriété.

IV-C-2 - Création du Tooltip

Notre Tooltip doit étendre les propriétés de chaque **Control** de sa **Form** parente --> il s'agit en fait d'ajouter une propriété **BasicToolTip** à chacun de ces **Controls** qui permettra de définir le texte et l'apparence du Tooltip. Pour se faire, il suffit d'utiliser l'interface **IExtenderProvider** et l'attribut **ProvideProperty**.

Extension

```

<ProvideProperty("BasicToolTip", GetType(Control))> _
Public Class BasicToolTip
    Inherits Component
    Implements IExtenderProvider
    
```

L'attribut **ProvideProperty** permet de définir la propriété à adjoindre aux différents composants à étendre.

Nota : L'utilisation de plusieurs **ProvideProperty** est possible.

Pour chacune des propriétés d'extension, il est nécessaire de définir des méthodes **GetNomDeLaProperty** et **SetNomDeLaProperty**.

 **Le nom de la propriété est sensible à la case.**

Dans notre cas, nous utiliserons les méthodes suivantes pour mettre à disposition la propriété **BasicToolTip** :

Get/Set de l'extension

```
<DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)> _
Public Function GetBasicToolTip(ByVal control As Control) As BasicToolTipExtendedProperty
...
End Function

Public Sub SetBasicToolTip(ByVal control As Control, _
ByVal BTTEP As BasicToolTipExtendedProperty)
...
End Sub
```

Tous les composants de la **Form** ne doivent pas être étendus. La limitation de l'extension est effectuée via l'implémentation de **IExtenderProvider.CanExtend**.

Dans notre cas, le ToolTip n'étendra que les composants de type **Control** :

Limitation de l'extension

```
Function CanExtend(ByVal target As Object) As Boolean Implements IExtenderProvider.CanExtend
If TypeOf target Is Control Then
Return True
End If
Return False
End Function
```

A ce niveau, nous avons tous les éléments pour définir notre ToolTip :

BasicToolTip

```
Option Strict On
Option Explicit On
Imports System.ComponentModel
<ProvideProperty("BasicToolTip", GetType(Control))> _
Public Class BasicToolTip
Inherits Component
Implements IExtenderProvider

Function CanExtend(ByVal target As Object) As Boolean _
Implements IExtenderProvider.CanExtend
If TypeOf target Is Control And Not TypeOf target Is BasicToolTip Then
Return True
End If
Return False
End Function
<DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)> _
Public Function GetBasicToolTip(ByVal control As Control) _
As BasicToolTipExtendedProperty

If slControl.ContainsKey(control) Then
Return CType(slControl(control), BasicToolTipExtendedProperty)
Else
Return New BasicToolTipExtendedProperty
End If

End Function
```

BasicToolTip

```

Public Sub SetBasicToolTip(ByVal control As Control, _
    ByVal BTTEP As BasicToolTipExtendedProperty)

    If (control Is Nothing) Then
        Throw New ArgumentNullException("control")
    End If

    If BTTEP.Text Is String.Empty Then
        Me.RemoveControl(control)
    Else
        If slControl.ContainsKey(control) Then
            Me.RemoveControl(control)
        End If
        Me.AddControl(control, BTTEP)
    End If

End Sub

Private bppPopup As New BasicPopup
Private slControl As New Hashtable

Private Sub AddControl(ByVal c As Control, _
    ByVal BTTEP As BasicToolTipExtendedProperty)
    slControl.Add(c, BTTEP)
    AddHandler c.MouseEnter, AddressOf ShowPopup
    AddHandler c.MouseLeave, AddressOf HidePopup
    ' gestion des controls possédant eux même un popup --> combo par ex
    AddHandler c.MouseCaptureChanged, AddressOf HidePopup
End Sub

Private Sub RemoveControl(ByVal c As Control)
    slControl.Remove(c)
    RemoveHandler c.MouseEnter, AddressOf ShowPopup
    RemoveHandler c.MouseLeave, AddressOf HidePopup
    RemoveHandler c.MouseCaptureChanged, AddressOf HidePopup
End Sub

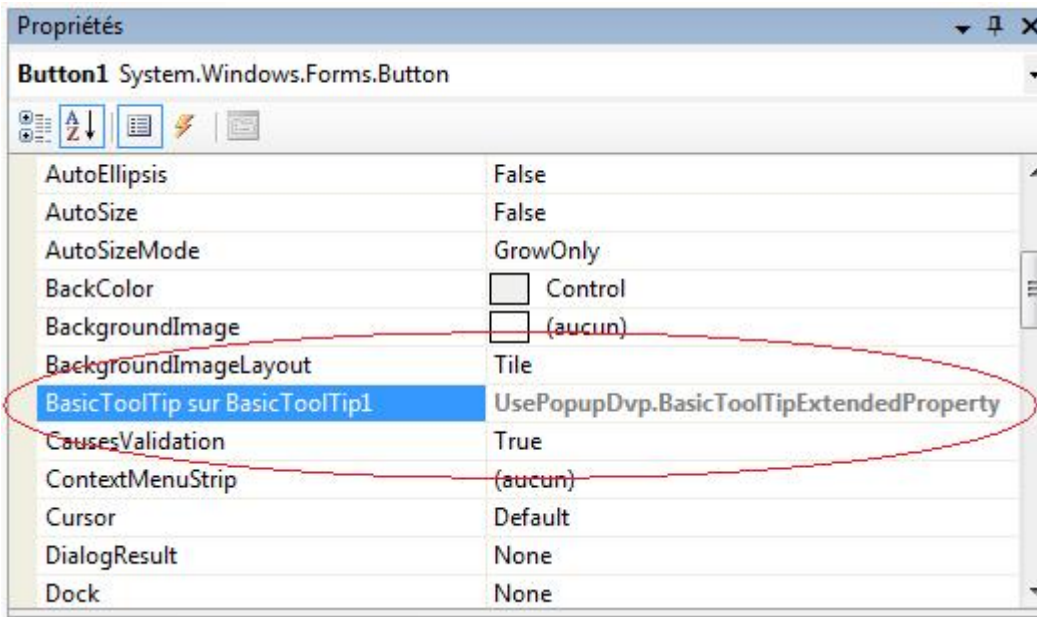
Private Sub ShowPopup(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim c As Control = CType(sender, Control)

    Dim BTTEP As BasicToolTipExtendedProperty = _
    CType(slControl(c), BasicToolTipExtendedProperty)
    bppPopup.BackColor = BTTEP.BackColor
    bppPopup.ForeColor = BTTEP.ForeColor
    bppPopup.ShadowForeColor = BTTEP.ShadowForeColor
    bppPopup.Font = BTTEP.Font
    bppPopup.ShowPopup(c, BTTEP.Text)
End Sub
Private Sub HidePopup(ByVal sender As Object, ByVal e As System.EventArgs)
    bppPopup.HidePopup()
End Sub

End Class
    
```

IV-C-3 - Déclaration d'un TypeConverter pour la propriété d'extension

Créons un formulaire et ajoutons un **Button** ainsi que notre composant **BasicToolTip**. Sur les propriétés du **Button**, nous obtenons ceci :



Nous ne pouvons pas exploiter notre extension !

Pour ce faire, il est nécessaire de définir un **TypeConverter** pour notre type **BasicToolTipExtendedProperty**.

TypeConverter pour BasicToolTipExtendedProperty

```

Option Strict On
Option Explicit On
Imports System.ComponentModel
Imports System.ComponentModel.Design.Serialization
Public Class BasicToolTipExtendedPropertyConverter
    Inherits TypeConverter

    Public Overloads Overrides Function ConvertFrom( _
        ByVal context As ITypeDescriptorContext, _
        ByVal cultureInfo As Globalization.CultureInfo, _
        ByVal value As Object) As Object

        If (TypeOf value Is String) Then

            Dim s() As String
            s = CType(value, String).Split(CChar(";"))
            If s.Length <> 4 Then Throw New Exception("Nombre d'arguments incorrects")
            Try
                Return New BasicToolTipExtendedProperty _
                    (s(0).ToString, _
                    Color.FromName(s(1)), _
                    Color.FromName(s(2)), _
                    Color.FromName(s(3)), _
                    CType(New FontConverter().ConvertFromString(s(4)), Font) _
                    )
            Catch
                Throw New InvalidCastException(value.ToString)
            End Try
        Else
            Return MyBase.ConvertFrom(context, cultureInfo, value)
        End If

    End Function

    Public Overloads Overrides Function ConvertTo( _
        ByVal context As ITypeDescriptorContext, _
        ByVal cultureInfo As Globalization.CultureInfo, _
        ByVal value As Object, ByVal destinationType _
            As Type) As Object

        If destinationType Is Nothing Then Throw New ArgumentNullException()
    
```

TypeConverter pour BasicToolTipExtendedProperty

```

    If destinationType Is GetType(String) And _
value.GetType Is GetType(BasicToolTipExtendedProperty) Then
    Dim item As BasicToolTipExtendedProperty = _
CType(value, BasicToolTipExtendedProperty)
    Return String.Format("{0};{1};{2};{3};{4}", _
        New Object() { _
            New StringConverter().ConvertToString(item.Text), _
            New ColorConverter().ConvertToString(item.ForeColor), _
            New ColorConverter().ConvertToString(item.ShadowForeColor), _
            New ColorConverter().ConvertToString(item.BackColor), _
            New FontConverter().ConvertToString(item.Font)})
    End If

    If destinationType Is GetType(InstanceDescriptor) And _
value.GetType Is GetType(BasicToolTipExtendedProperty) Then
    Dim itemtemp As BasicToolTipExtendedProperty = _
CType(value, BasicToolTipExtendedProperty)
    Dim instance As InstanceDescriptor = _
        New InstanceDescriptor( _
            GetType(BasicToolTipExtendedProperty).GetConstructor(New Type() _
                {GetType(String), GetType(Color), GetType(Color), GetType(Color)}, _
                GetType(Font)}), _
            New Object() { _
                itemtemp.Text, _
                itemtemp.ForeColor, _
                itemtemp.ShadowForeColor, _
                itemtemp.BackColor, _
                itemtemp.Font})
    Return instance
    End If

Return MyBase.ConvertTo(context, cultureInfo, value, destinationType)

End Function

Public Overloads Overrides Function CanConvertFrom( _
    ByVal context As ITypeDescriptorContext, _
    ByVal sourceType As Type) As Boolean

    If (sourceType.Equals(GetType(String))) Then
        Return True
    Else
        Return MyBase.CanConvertFrom(context, sourceType)
    End If

End Function

Public Overloads Overrides Function CanConvertTo( _
    ByVal context As ITypeDescriptorContext, _
    ByVal destinationType As Type) As Boolean

    If (destinationType.Equals(GetType(BasicToolTipExtendedProperty))) Then
        Return True
    Else
        Return MyBase.CanConvertTo(context, destinationType)
    End If

End Function

Public Overloads Overrides Function GetPropertiesSupported( _
    ByVal context As ITypeDescriptorContext) As Boolean
    Return True
End Function

Public Overloads Overrides Function GetProperties( _
    ByVal context As ITypeDescriptorContext, _
    ByVal value As Object, _
    ByVal attribute() As Attribute) _
As PropertyDescriptorCollection

Return TypeDescriptor.GetProperties(value)
    
```

TypeConverter pour BasicToolTipExtendedProperty

```

End Function

End Class
    
```

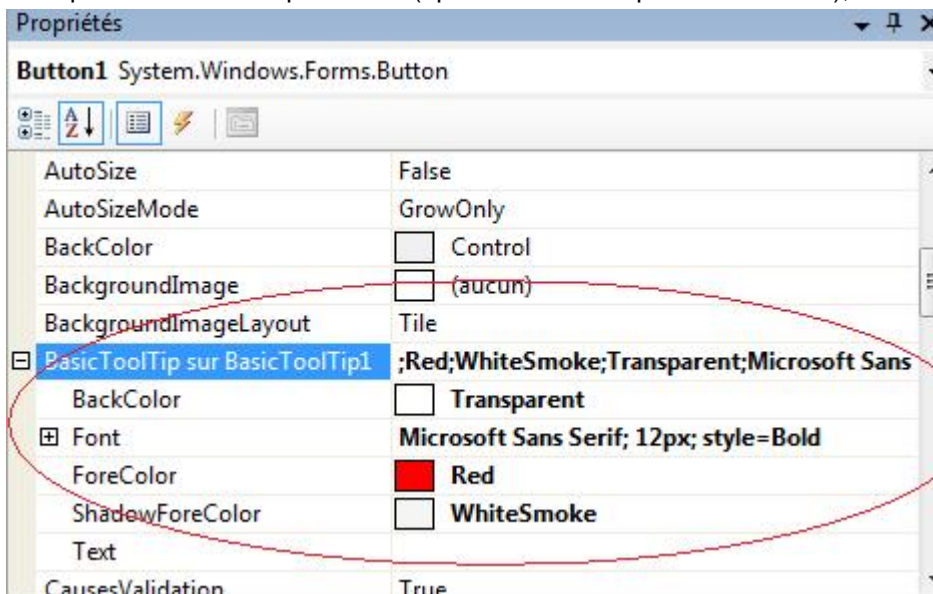
Nota : l'explication de la création d'un TypeConverter n'est pas détaillée ici car hors cadre de cet article. Une fois ce **TypeConverter** créé, nous devons informer le designer de son existence pour la propriété **.BasicToolTip** mise à disposition par notre objet **BasicToolTip**. Pour cela, nous utilisons l'attribut **TypeConverter** :

Utilisation de l'attribut TypeConverter

```

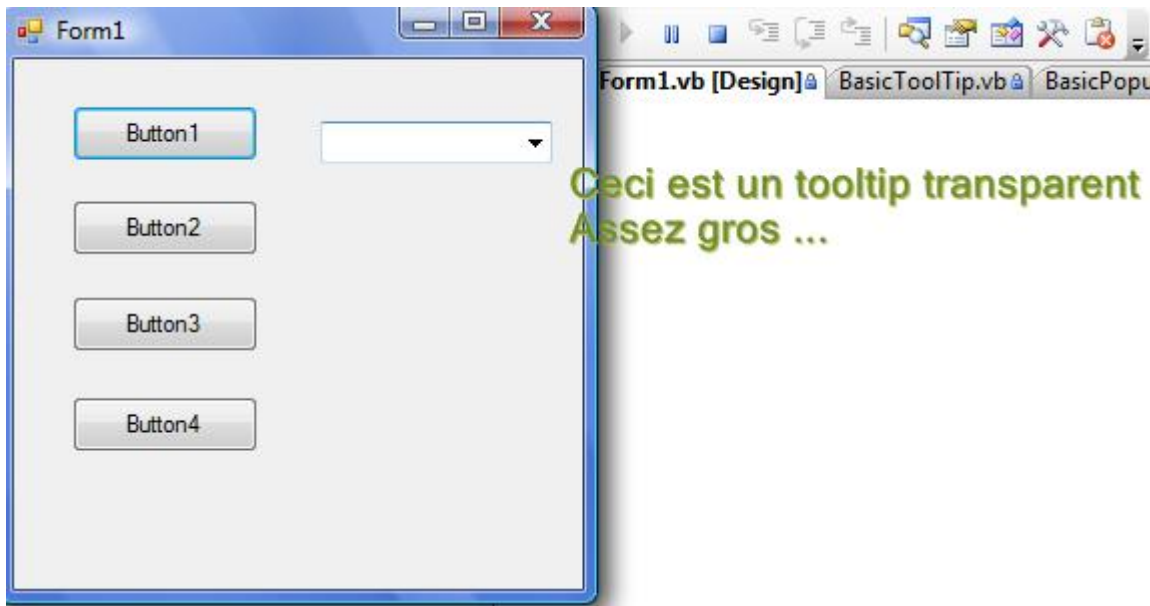
<ProvideProperty("BasicToolTip", GetType(Control))> _
Public Class BasicToolTip
    Inherits Component
    Implements IExtenderProvider
    ...
    <TypeConverter(GetType(BasicToolTipExtendedPropertyConverter)), _
    DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)> _
    Public Function GetBasicToolTip(ByVal control As Control) As BasicToolTipExtendedProperty
    
```

En reprenant notre test précédent (après avoir fermé puis réouvert VS), nous obtenons :



IV-C-4 - Résultat

Nous pouvons maintenant utiliser le ToolTip est obtenir, par exemple, lors de l'entrée de la souris sur un **Control** ceci :



IV-F - Exemple d'utilisation

Un exemple d'utilisation ici : [UsePopup.zip](#)

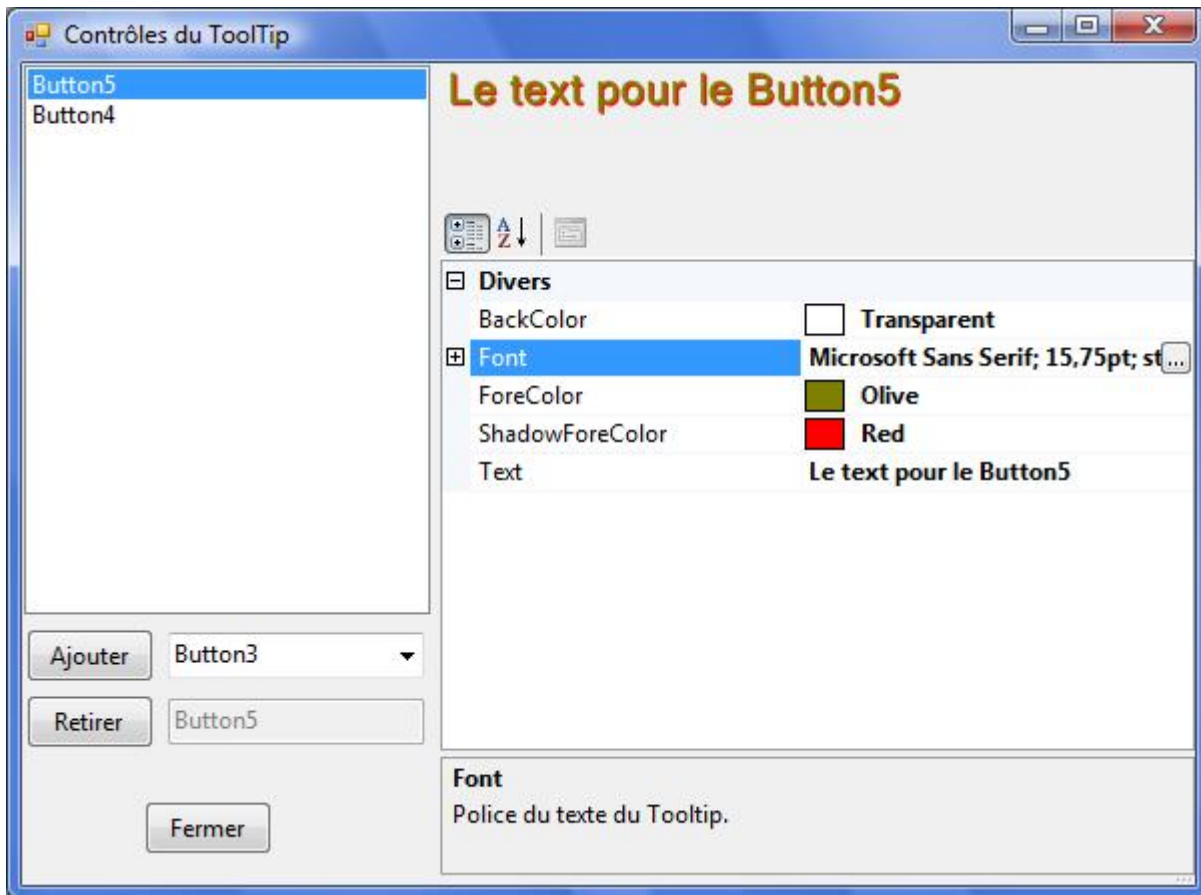
IV-E - Limites

Notre ToolTip bien que fonctionnel présente quelques défauts.

En effet, l'apparition ou la disparition du Popup est quelque peu "abrupte" et il faut reconnaître que l'utilisation de l'Extender est assez moyenne (même si c'est la solution utilisée par le ToolTip classique).

Pour ma part, il me semble plus logique de gérer l'ajout et le retrait des **Controls** d'une collection de **Controls** géré par le ToolTip.

Ceci nécessite toutefois de définir un éditeur de collection personnalisé et c'est ce que nous verrons dans un article à venir sur l'utilisation du **UIEditor** qui nous permettra d'obtenir lors du Design Time :



V - Remerciements

Merci à **Aspic** pour la relecture de cet article.