Getting the Most Out of the .NET Framework PropertyGrid Control

Mark Rideout
Microsoft Corporation

Applies to:
Microsoft® .NET® Framework
Microsoft® Visual Studio® .NET

Summary: Helps you understand the PropertyGrid control in the Microsoft .NET Framework and how to customize it for your application. (37 printed pages)

Contents

Introducing the PropertyGrid Control

If you have used Microsoft® Visual Basic® or Microsoft Visual Studio .NET then you have used a property browser to browse, view, and edit the properties of one or more objects. The .NET Framework **PropertyGrid** control is core to the property browser that is in Visual Studio .NET. The **PropertyGrid** control displays properties for any object or type, and it retrieves the item's properties, primarily using reflection. (Reflection is a technology that provides type information at run time.)

The following screen shot shows how a **PropertyGrid** looks when placed on a form.

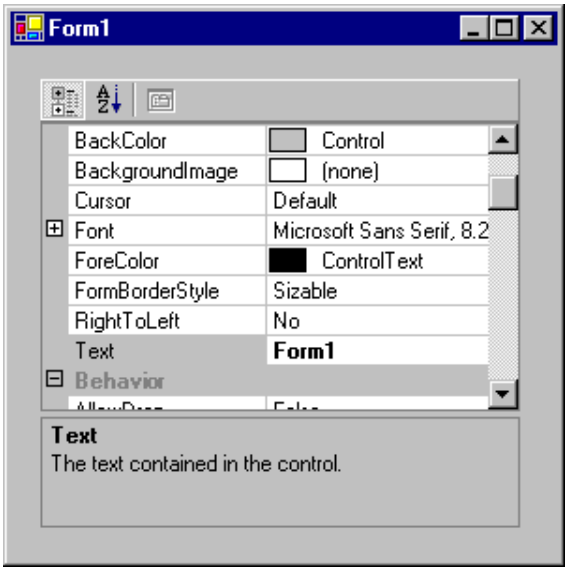

Figure 1. PropertyGrid on a form

The **PropertyGrid** contains the following parts:

- Properties

- Expandable Properties
- Property Category Headings
- Property Descriptions
- Property Editors
- Property Tabs
- Commands Pane (displays designer verbs that a control's designer exposes)

Creating a PropertyGrid Control

To create a **PropertyGrid** control using Visual Studio .NET, you need to add the **PropertyGrid** control to the toolbox, since it is not included by default. From the **Tools** menu, select **Customize Toolbox**. In the dialog box, select the **Framework Components** tab and then select **PropertyGrid**.

If you are compiling your code from the command line, use the **/reference** option and specify System.Windows.Forms.dll.

The following code demonstrates creating a **PropertyGrid** control and adding it to a form.

```
' Visual Basic

Imports System
Imports System.Drawing
Imports System.ComponentModel
Imports System.Windows.Forms
Imports System.Globalization

Public Class OptionsDialog
   Inherits System.Windows.Forms.Form

   Private OptionsPropertyGrid As System.Windows.Forms.PropertyGrid

   Public Sub New()
      MyBase.New()

      OptionsPropertyGrid = New PropertyGrid()
      OptionsPropertyGrid.Size = New Size(300, 250)

      Me.Controls.Add(OptionsPropertyGrid)
      Me.Text = "Options Dialog"
   End Sub
End Class


//C#

using System;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;
using System.Globalization;

public class OptionsDialog : System.Windows.Forms.Form
{
   private System.Windows.Forms.PropertyGrid OptionsPropertyGrid;
   public OptionsDialog()
   {
      OptionsPropertyGrid = new PropertyGrid();
      OptionsPropertyGrid.Size = new Size(300, 250);

      this.Controls.Add(OptionsPropertyGrid);
      this.Text = "Options Dialog";
   }

   [STAThread]
```

```
    static void Main()
    {
       Application.Run(new OptionsDialog());
    }
}
```

Where to Use the PropertyGrid Control

There are numerous places in an application where you might want to provide a richer editing experience by having the user interact with a **PropertyGrid**. One example is an application that has several "settings" or options, some of them complex, that a user can set. You could use radio buttons, combo boxes, or text boxes to represent these options. Instead, this paper will step you though the process of using the **PropertyGrid** control to create an options window for setting application options. The `OptionsDialog` form that you created above will be the start of the options window. Now create a class called `AppSettings` that contains all of the properties that map to the application settings. The settings are much easier to manage and maintain if you create a separate class rather than using individual variables.

```
' Visual Basic

Public Class AppSettings
    Private _saveOnClose As Boolean = True
    Private _greetingText As String = "Welcome to your application!"
    Private _maxRepeatRate As Integer = 10
    Private _itemsInMRU As Integer = 4

    Private _settingsChanged As Boolean = False
    Private _appVersion As String = "1.0"

    Public Property SaveOnClose() As Boolean
        Get
            Return _saveOnClose
        End Get
        Set(ByVal Value As Boolean)
            SaveOnClose = Value
        End Set
    End Property

    Public Property GreetingText() As String
        Get
            Return _greetingText
        End Get
        Set(ByVal Value As String)
            _greetingText = Value
        End Set
    End Property

    Public Property ItemsInMRUList() As Integer
        Get
            Return _itemsInMRU
        End Get
        Set(ByVal Value As Integer)
            _itemsInMRU = Value
        End Set
    End Property

    Public Property MaxRepeatRate() As Integer
        Get
            Return _maxRepeatRate
        End Get
        Set(ByVal Value As Integer)
            _maxRepeatRate = Value
        End Set
    End Property
```

```
    Public Property SettingsChanged() As Boolean
        Get
            Return _settingsChanged
        End Get
        Set(ByVal Value As Boolean)
            _settingsChanged = Value
        End Set
    End Property

    Public Property AppVersion() As String
        Get
            Return _appVersion
        End Get
        Set(ByVal Value As String)
            _appVersion = Value
        End Set
    End Property
End Class


//C#

public class AppSettings{
    private bool saveOnClose = true;
    private string greetingText = "Welcome to your application!";
    private int itemsInMRU = 4;
    private int maxRepeatRate = 10;
    private bool settingsChanged = false;
    private string appVersion = "1.0";

    public bool SaveOnClose
    {
        get { return saveOnClose; }
        set { saveOnClose = value;}
    }
    public string GreetingText
    {
        get { return greetingText; }
        set { greetingText = value; }
    }
    public int MaxRepeatRate
    {
        get { return maxRepeatRate; }
        set { maxRepeatRate = value; }
    }
    public int ItemsInMRUList
    {
        get { return itemsInMRU; }
        set { itemsInMRU = value; }
    }
    public bool SettingsChanged
    {
        get { return settingsChanged; }
        set { settingsChanged = value; }
    }
    public string AppVersion
    {
        get { return appVersion; }
        set { appVersion = value; }
    }
}
```

The **PropertyGrid** on the options window will manipulate this class, so add the class definition to your application project,

either in a new file or at the bottom of the source code for the form.

Selecting an Object

To identify what the **PropertyGrid** displays, set the **PropertyGrid.SelectedObject** property to an object instance. The **PropertyGrid** does the rest. Each time you set **SelectedObject**, the **PropertyGrid** refreshes the properties shown. This provides an easy way to force the refresh of properties, or to switch between objects at run time. You can also call the **PropertyGrid.Refresh** method to refresh properties.

To continue, update the code in the `OptionsDialog` constructor to create an `AppSettings` object and set it to the **PropertyGrid.SelectedObject** property.

```
' Visual Basic

   Public Sub New()
      MyBase.New()

      OptionsPropertyGrid = New PropertyGrid()
      OptionsPropertyGrid.Size = New Size(300, 250)

      Me.Controls.Add(OptionsPropertyGrid)
      Me.Text = "Options Dialog"

      ' Create the AppSettings class and display it in the PropertyGrid.
      Dim appset as AppSettings = New AppSettings()
      OptionsPropertyGrid.SelectedObject = appset
   End Sub


//C#

   public OptionsDialog()
   {
      OptionsPropertyGrid = new PropertyGrid();
      OptionsPropertyGrid.Size = new Size(300, 250);

      this.Controls.Add(OptionsPropertyGrid);
      this.Text = "Options Dialog";

      // Create the AppSettings class and display it in the PropertyGrid.
      AppSettings appset = new AppSettings();
      OptionsPropertyGrid.SelectedObject = appset;
   }
```

Compile and run the application. The following screen shot shows how it should look.

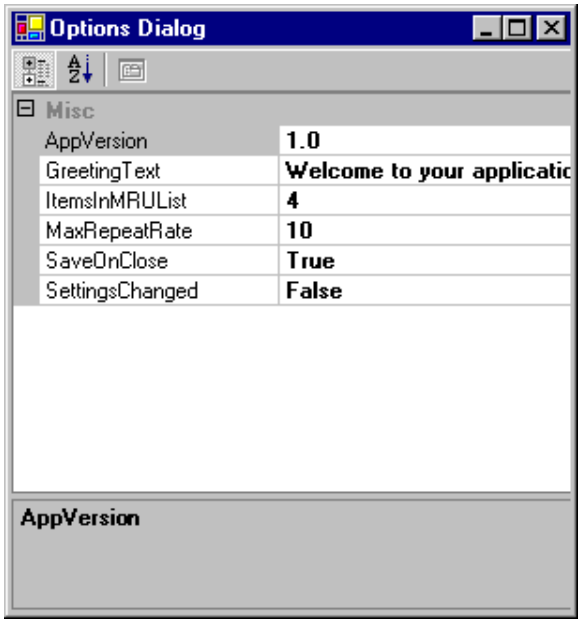

Figure 2. AppSettings class selected in the PropertyGrid

Customizing the PropertyGrid Control

You can modify some visual aspects of the **PropertyGrid** to fit your needs. You might want to change how some properties are displayed, and even choose to not display some properties. How customizable is the **PropertyGrid**?

Changing Visual Aspects of the PropertyGrid

Many visual aspects of the **PropertyGrid** are customizable. Here is a partial list:

- Change the background color, change the font color, or hide the description pane through the **HelpBackColor**, **HelpForeColor**, and **HelpVisible** properties.
- Hide the toolbar through the **ToolbarVisible** property, change its color through the **BackColor** property, and display large toolbar buttons through the **LargeButtons** property.
- Sort the properties alphabetically, and categorized them using the **PropertySort** property.
- Change the splitter color through the **BackColor** property.
- Change the grid line and borders through the **LineColor** property.

For the options window in this example the toolbar is not needed, so set **ToolbarVisible** to **false**. Keep the other default settings.

Changing How Properties Are Displayed

To change how some properties are displayed, you can apply different attributes to the properties. Attributes are declarative tags used to annotate programming elements such as types, fields, methods, and properties that can be retrieved at run time using reflection. Here is a partial list:

- **DescriptionAttribute**. Sets the text for the property that is displayed in the description help pane below the properties. This is a useful way to provide help text for the active property (the property that has focus). Apply this attribute to the `MaxRepeatRate` property.
- **CategoryAttribute**. Sets the category that the property is under in the grid. This is useful when you want a property grouped by a category name. If a property does not have a category specified, then it will be assigned to the **Misc** category. Apply this attribute to all properties.
- **BrowsableAttribute** – Indicates whether the property is shown in the grid. This is useful when you want to hide a property from the grid. By default, a public property is always shown in the grid. Apply this attribute to the `SettingsChanged` property.
- **ReadOnlyAttribute** – Indicates whether the property is read-only. This is useful when you want to keep a property from being editable in the grid. By default, a public property with get and set accessor functions is editable in the grid. Apply this attribute to the `AppVersion` property.
- **DefaultValueAttribute** – Identifies the property's default value. This is useful when you want to provide a default value for a property and later determine if the property's value is different than the default. Apply this attribute to all properties.

- **DefaultPropertyAttribute** – Identifies the default property for the class. The default property for a class gets the focus first when the class is selected in the grid. Apply this attribute to the `AppSettings` class.

Now apply some of these attributes to the `AppSettings` class to change the way the properties are displayed in the **PropertyGrid.**

```
' Visual Basic

<DefaultPropertyAttribute("SaveOnClose")> _
Public Class AppSettings
    Private _saveOnClose As Boolean = True
    Private _greetingText As String = "Welcome to your application!"
    Private _maxRepeatRate As Integer = 10
    Private _itemsInMRU As Integer = 4

    Private _settingsChanged As Boolean = False
    Private _appVersion As String = "1.0"

    <CategoryAttribute("Document Settings"), _
     DefaultValueAttribute(True)> _
    Public Property SaveOnClose() As Boolean
        Get
            Return _saveOnClose
        End Get
        Set(ByVal Value As Boolean)
            SaveOnClose = Value
        End Set
    End Property

    <CategoryAttribute("Global Settings"), _
    ReadOnlyAttribute(True), _
    DefaultValueAttribute("Welcome to your application!")> _
    Public Property GreetingText() As String
        Get
            Return _greetingText
        End Get
        Set(ByVal Value As String)
            _greetingText = Value
        End Set
    End Property

    <CategoryAttribute("Global Settings"), _
    DefaultValueAttribute(4)> _
    Public Property ItemsInMRUList() As Integer
        Get
            Return _itemsInMRU
        End Get
        Set(ByVal Value As Integer)
            _itemsInMRU = Value
        End Set
    End Property

    <DescriptionAttribute("The rate in milliseconds that the text will repeat."), _
    CategoryAttribute("Global Settings"), _
    DefaultValueAttribute(10)> _
    Public Property MaxRepeatRate() As Integer
        Get
            Return _maxRepeatRate
        End Get
        Set(ByVal Value As Integer)
            _maxRepeatRate = Value
        End Set
    End Property
```

```
    <BrowsableAttribute(False),
     DefaultValueAttribute(False)> _
    Public Property SettingsChanged() As Boolean
        Get
            Return _settingsChanged
        End Get
        Set(ByVal Value As Boolean)
            _settingsChanged = Value
        End Set
    End Property

    <CategoryAttribute("Version"), _
    DefaultValueAttribute("1.0"), _
    ReadOnlyAttribute(True)> _
    Public Property AppVersion() As String
        Get
            Return _appVersion
        End Get
        Set(ByVal Value As String)
            _appVersion = Value
        End Set
    End Property
End Class


//C#
[DefaultPropertyAttribute("SaveOnClose")]
public class AppSettings{
    private bool saveOnClose = true;
    private string greetingText = "Welcome to your application!";
    private int maxRepeatRate = 10;
    private int itemsInMRU = 4;

    private bool settingsChanged = false;
    private string appVersion = "1.0";

    [CategoryAttribute("Document Settings"),
    DefaultValueAttribute(true)]
    public bool SaveOnClose
    {
        get { return saveOnClose; }
        set { saveOnClose = value;}
    }

    [CategoryAttribute("Global Settings"),
    ReadOnlyAttribute(true),
    DefaultValueAttribute("Welcome to your application!")]
    public string GreetingText
    {
        get { return greetingText; }
        set { greetingText = value; }
    }

    [CategoryAttribute("Global Settings"),
    DefaultValueAttribute(4)]
    public int ItemsInMRUList
    {
        get { return itemsInMRU; }
        set { itemsInMRU = value; }
    }

    [DescriptionAttribute("The rate in milliseconds that the text will repeat."),
    CategoryAttribute("Global Settings"),
    DefaultValueAttribute(10)]
```

```
        public int MaxRepeatRate
        {
            get { return maxRepeatRate; }
            set { maxRepeatRate = value; }
        }

        [BrowsableAttribute(false),
        DefaultValueAttribute(false)]
        public bool SettingsChanged
        {
            get { return settingsChanged; }
            set { settingsChanged = value; }
        }

        [CategoryAttribute("Version"),
        DefaultValueAttribute("1.0"),
        ReadOnlyAttribute(true)]
        public string AppVersion
        {
            get { return appVersion; }
            set { appVersion = value; }
        }
    }
```

With these attributes applied to the`AppSettings` class, compile and run the application. The following screen shot shows how it should look.

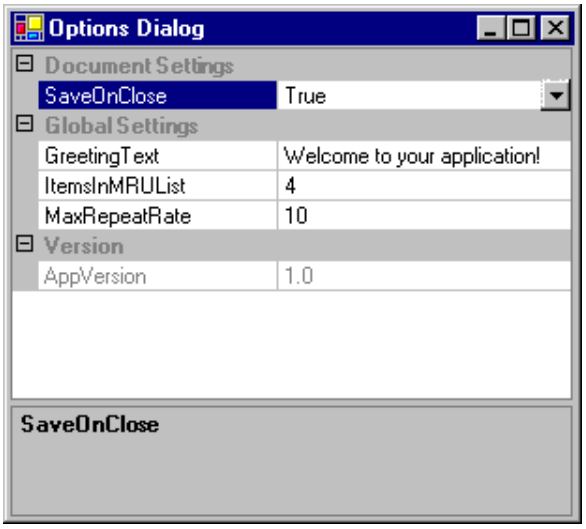

Figure 3. Properties displayed with categories and default values in the PropertyGrid

After working with this version of the options window, you might notice the following things:

- The `SaveOnClose` property gets focus when the window is displayed.
- The `MaxRepeatRate` property displays "The rate in milliseconds that the text will repeat" in the description help pane when selected.
- The `SaveOnClose` property is displayed under the "Document Settings" category. The other properties are displayed under two other categories named "Global Settings" and "Version."
- The `SettingsChanged` property is no longer displayed.
- The `AppVersion` property is read-only. Read-only properties are displayed with dimmed text.
- When the `SaveOnClose` property has a value other than **true**, it is displayed in bold text. The **PropertyGrid** uses bold text to indicate properties that have a non-default value.

Displaying Complex Properties

So far, the options window has displayed simple types like integers, Booleans, and strings. What about more complex types? What if your application needs to track something like window size, document font, or toolbar color? Some data

types provided by the .NET Framework have special display abilities that help make them more usable in the **PropertyGrid**.

Support for Provided Types

First, update the `AppSettings` class to add new properties for window size (**Size** type), window font (**Font** type), and toolbar color (**Color** type).

```
' Visual Basic

<DefaultPropertyAttribute("SaveOnClose")> _
Public Class AppSettings
    Private _saveOnClose As Boolean = True
    Private _greetingText As String = "Welcome to your application!"
    Private _maxRepeatRate As Integer = 10
    Private _itemsInMRU As Integer = 4

    Private _settingsChanged As Boolean = False
    Private _appVersion As String = "1.0"

    Private _windowSize As Size = New Size(100, 100)
    Private _windowFont As Font = New Font("Arial", 8, FontStyle.Regular)
    Private _toolbarColor As Color = SystemColors.Control

    <CategoryAttribute("Document Settings"), _
     DefaultValueAttribute(True)> _
    Public Property SaveOnClose() As Boolean
        Get
            Return _saveOnClose
        End Get
        Set(ByVal Value As Boolean)
            SaveOnClose = Value
        End Set
    End Property

    <CategoryAttribute("Document Settings")> _
    Public Property WindowSize() As Size
        Get
            Return _windowSize
        End Get
        Set(ByVal Value As Size)
            _windowSize = Value
        End Set
    End Property

    <CategoryAttribute("Document Settings")> _
    Public Property WindowFont() As Font
        Get
            Return _windowFont
        End Get
        Set(ByVal Value As Font)
            _windowFont = Value
        End Set
    End Property

    <CategoryAttribute("Global Settings")> _
    Public Property ToolbarColor() As Color
        Get
            Return _toolbarColor
        End Get
        Set(ByVal Value As Color)
            _toolbarColor = Value
        End Set
    End Property
```

```
    <CategoryAttribute("Global Settings"), _
    ReadOnlyAttribute(True), _
    DefaultValueAttribute("Welcome to your application!")> _
    Public Property GreetingText() As String
        Get
            Return _greetingText
        End Get
        Set(ByVal Value As String)
            _greetingText = Value
        End Set
    End Property

    <CategoryAttribute("Global Settings"), _
    DefaultValueAttribute(4)> _
    Public Property ItemsInMRUList() As Integer
        Get
            Return _itemsInMRU
        End Get
        Set(ByVal Value As Integer)
            _itemsInMRU = Value
        End Set
    End Property

    <DescriptionAttribute("The rate in milliseconds that the text will repeat."), _
    CategoryAttribute("Global Settings"), _
    DefaultValueAttribute(10)> _
    Public Property MaxRepeatRate() As Integer
        Get
            Return _maxRepeatRate
        End Get
        Set(ByVal Value As Integer)
            _maxRepeatRate = Value
        End Set
    End Property

    <BrowsableAttribute(False),
     DefaultValueAttribute(False)> _
    Public Property SettingsChanged() As Boolean
        Get
            Return _settingsChanged
        End Get
        Set(ByVal Value As Boolean)
            _settingsChanged = Value
        End Set
    End Property

    <CategoryAttribute("Version"), _
    DefaultValueAttribute("1.0"), _
    ReadOnlyAttribute(True)> _
    Public Property AppVersion() As String
        Get
            Return _appVersion
        End Get
        Set(ByVal Value As String)
            _appVersion = Value
        End Set
    End Property
End Class


//C#

[DefaultPropertyAttribute("SaveOnClose")]
public class AppSettings{
```

```
        private bool saveOnClose = true;
        private string greetingText = "Welcome to your application!";
        private int maxRepeatRate = 10;
        private int itemsInMRU = 4;

        private bool settingsChanged = false;
        private string appVersion = "1.0";

        private Size windowSize = new Size(100,100);
        private Font windowFont = new Font("Arial", 8, FontStyle.Regular);
        private Color toolbarColor = SystemColors.Control;

        [CategoryAttribute("Document Settings"),
        DefaultValueAttribute(true)]
        public bool SaveOnClose
        {
            get { return saveOnClose; }
            set { saveOnClose = value;}
        }

        [CategoryAttribute("Document Settings")]
        public Size WindowSize
        {
            get { return windowSize; }
            set { windowSize = value;}
        }

        [CategoryAttribute("Document Settings")]
        public Font WindowFont
        {
            get {return windowFont; }
            set { windowFont = value;}
        }

        [CategoryAttribute("Global Settings")]
        public Color ToolbarColor
        {
            get { return toolbarColor; }
            set { toolbarColor = value; }
        }

        [CategoryAttribute("Global Settings"),
        ReadOnlyAttribute(true),
        DefaultValueAttribute("Welcome to your application!")]
        public string GreetingText
        {
            get { return greetingText; }
            set { greetingText = value; }
        }

        [CategoryAttribute("Global Settings"),
        DefaultValueAttribute(4)]
        public int ItemsInMRUList
        {
            get { return itemsInMRU; }
            set { itemsInMRU = value; }
        }

        [DescriptionAttribute("The rate in milliseconds that the text will repeat."),
        CategoryAttribute("Global Settings"),
        DefaultValueAttribute(10)]
        public int MaxRepeatRate
        {
            get { return maxRepeatRate; }
            set { maxRepeatRate = value; }
```

```
    }

    [BrowsableAttribute(false),
    DefaultValueAttribute(false)]
    public bool SettingsChanged
    {
        get { return settingsChanged; }
        set { settingsChanged = value; }
    }

    [CategoryAttribute("Version"),
    DefaultValueAttribute("1.0"),
    ReadOnlyAttribute(true)]
    public string AppVersion
    {
        get { return appVersion; }
        set { appVersion = value; }
    }
}
```

The following screen shot shows how the new properties look in the **PropertyGrid**.

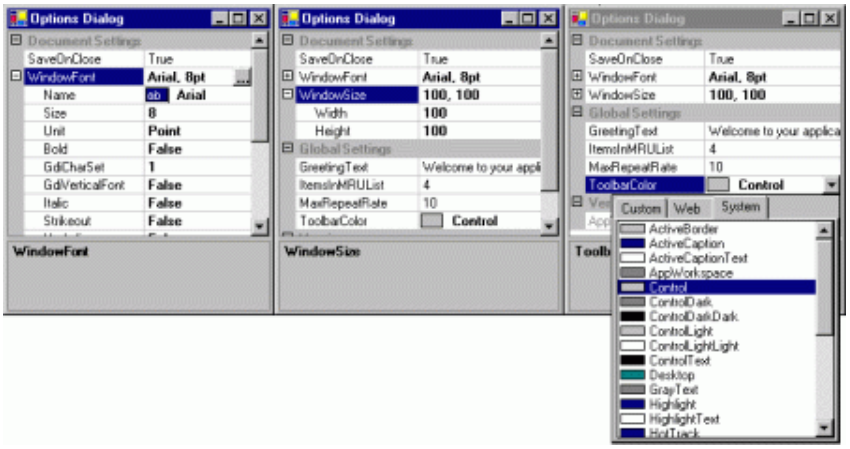

Figure 4. .NET Framework data types displayed in the PropertyGrid

Notice that the `WindowFont` property has an ellipsis ("...") button that displays a font selection dialog when pressed. In addition, the property can be expanded to display more **Font** properties. Some of the **Font** properties provide a drop-down list of values and details about the font. The `WindowSize` property can be expanded to show more properties of the **Size** type. And finally, notice that the `ToolbarColor` property has a swatch of the selected color and a custom drop-down list to select different colors. For these data types and others, the .NET Framework provides additional classes to make editing in the **PropertyGrid** easier.

Support for Custom Types

Now add two more properties to the `AppSettings` class, one called `DefaultFileName` and the other called `SpellCheckOptions`. The `DefaultFileName` property gets or sets a string and the `SpellCheckOptions` property gets or sets an instance of the `SpellingOptions` class.

The `SpellingOptions` class is a new class that will manage the application's spell checking properties. There is no hard and fast rule about when to create a separate class to manage an object's properties; it depends upon your overall class design. Add the `SpellingOptions` class definition to your application project, either in a new file or at the bottom of the source code for the form.

```
' Visual Basic

<DescriptionAttribute("Expand to see the spelling options for the application.")> _
Public Class SpellingOptions
    Private _spellCheckWhileTyping As Boolean = True
```

```
    Private _spellCheckCAPS As Boolean = False
    Private _suggestCorrections As Boolean = True

    <DefaultValueAttribute(True)> _
    Public Property SpellCheckWhileTyping() As Boolean
        Get
            Return _spellCheckWhileTyping
        End Get
        Set(ByVal Value As Boolean)
            _spellCheckWhileTyping = Value
        End Set
    End Property

    <DefaultValueAttribute(False)> _
    Public Property SpellCheckCAPS() As Boolean
        Get
            Return _spellCheckCAPS
        End Get
        Set(ByVal Value As Boolean)
            _spellCheckCAPS = Value
        End Set
    End Property

    <DefaultValueAttribute(True)> _
    Public Property SuggestCorrections() As Boolean
        Get
            Return _suggestCorrections
        End Get
        Set(ByVal Value As Boolean)
            _suggestCorrections = Value
        End Set
    End Property
End Class


//C#

[DescriptionAttribute("Expand to see the spelling options for the application.")]
public class SpellingOptions{
    private bool spellCheckWhileTyping = true;
    private bool spellCheckCAPS = false;
    private bool suggestCorrections = true;

    [DefaultValueAttribute(true)]
    public bool SpellCheckWhileTyping
    {
        get { return spellCheckWhileTyping; }
        set { spellCheckWhileTyping = value; }
    }

    [DefaultValueAttribute(false)]
    public bool SpellCheckCAPS
    {
        get { return spellCheckCAPS; }
        set { spellCheckCAPS = value; }
    }
    [DefaultValueAttribute(true)]
    public bool SuggestCorrections
    {
        get { return suggestCorrections; }
        set { suggestCorrections = value; }
    }
}
```

Compile and run the options window application again. The following screen shot shows what it should look like.

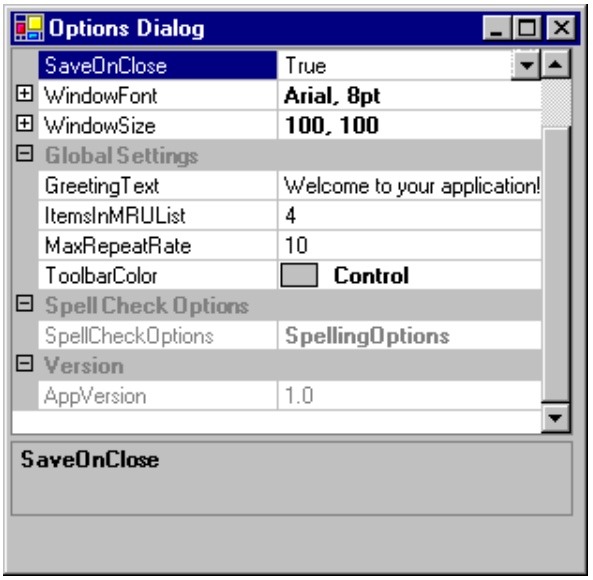

Figure 5. Custom data types without a type converter, displayed in the PropertyGrid

Notice how the `SpellcheckOptions` property looks. Unlike the .NET Framework types, it doesn't expand or display a custom string representation. What if you want to provide the same editing experience that the .NET Framework types have to your own complex type? The .NET Framework types use the **TypeConverter** and **UITypeEditor** classes to provide most of the **PropertyGrid** editing support, and you can use them too.

Adding Expandable Property Support

To get the **PropertyGrid** to expand the `SpellingOptions` property, you will need to create a **TypeConverter**. A **TypeConverter** provides a way to convert from one type to another. The **PropertyGrid** uses the **TypeConverter** to convert your object type to a **String**, which it uses to display the object value in the grid. During editing, the **TypeConverter** converts back to your object type from a **String**. The .NET Framework provides the **ExpandableObjectConverter** class to make this easier.

To provide expandable object support

1. Create a class that inherits from **ExpandableObjectConverter**.

```
' Visual Basic

Public Class SpellingOptionsConverter
    Inherits ExpandableObjectConverter
End Class


//C#

public class SpellingOptionsConverter:ExpandableObjectConverter
{  }
```

2. Override the **CanConvertTo** method and return **true** if the `destinationType` parameter is the same type as the class that uses this type converter (the `SpellingOptions` class in your example); otherwise, return the value of the base class **CanConvertTo** method.

```
' Visual Basic

Public Overloads Overrides Function CanConvertTo( _
                                ByVal context As ITypeDescriptorContext, _
                                ByVal destinationType As Type) As Boolean
    If (destinationType Is GetType(SpellingOptions)) Then
        Return True
    End If
```

```
    Return MyBase.CanConvertFrom(context, destinationType)
End Function
```

```
//C#

public override bool CanConvertTo(ITypeDescriptorContext context,
                                  System.Type destinationType)
{
    if (destinationType == typeof(SpellingOptions))
        return true;

    return base.CanConvertTo(context, destinationType);
}
```

3. Override the **ConvertTo** method and ensure that the `destinationType` parameter is a **String** and that the value is the same type as the class that uses this type converter (the `SpellingOptions` class in your example). If either case is **false**, return the value of the base class **ConvertTo** method; otherwise return a string representation of the value object. The string representation needs to separate each property of your class with a unique delimiter. Since the whole string will be displayed in the **PropertyGrid** you will want to choose a delimiter that doesn't detract from the readability; commas usually work well.

```
' Visual Basic

Public Overloads Overrides Function ConvertTo( _
                            ByVal context As ITypeDescriptorContext, _
                            ByVal culture As CultureInfo, _
                            ByVal value As Object, _
                            ByVal destinationType As System.Type) _
                    As Object
    If (destinationType Is GetType(System.String) _
        AndAlso TypeOf value Is SpellingOptions) Then

        Dim so As SpellingOptions = CType(value, SpellingOptions)

        Return "Check while typing: " & so.SpellCheckWhileTyping & _
               ", check CAPS: " & so.SpellCheckCAPS & _
               ", suggest corrections: " & so.SuggestCorrections
    End If
    Return MyBase.ConvertTo(context, culture, value, destinationType)
End Function
```

```
//C#

public override object ConvertTo(ITypeDescriptorContext context,
                                 CultureInfo culture,
                                 object value,
                                 System.Type destinationType)
{
    if (destinationType == typeof(System.String) &&
         value is SpellingOptions){

        SpellingOptions so = (SpellingOptions)value;

        return "Check while typing:" + so.SpellCheckWhileTyping +
               ", check CAPS: " + so.SpellCheckCAPS +
               ", suggest corrections: " + so.SuggestCorrections;
    }
```

```
    return base.ConvertTo(context, culture, value, destinationType);
}
```

4. (Optional) You can enable editing of the object's string representation in the grid by specifying that your type converter can convert from a string. To do this, first override the **CanConvertFrom** method and return **true** if the source **Type** parameter is of type **String**; otherwise, return the value of the base class **CanConvertFrom** method.

```
' Visual Basic

Public Overloads Overrides Function CanConvertFrom( _
                           ByVal context As ITypeDescriptorContext, _
                           ByVal sourceType As System.Type) As Boolean
    If (sourceType Is GetType(String)) Then
        Return True
    End If
    Return MyBase.CanConvertFrom(context, sourceType)
End Function
```

```
//C#

public override bool CanConvertFrom(ITypeDescriptorContext context,
                              System.Type sourceType)
{
    if (sourceType == typeof(string))
        return true;

    return base.CanConvertFrom(context, sourceType);
}
```

5. To enable editing of the object's base class, you also need to override the **ConvertFrom** method and ensure that the value parameter is a **String**. If it is not a **String**, return the value of the base class **ConvertFrom** method; otherwise return a new instance of your class (the `SpellingOptions` class in your example) based on the value parameter. You will need to parse the values for each property of your class from the value parameter. Knowing the format of the delimited string you created in the **ConvertTo** method will help you perform the parsing.

```
' Visual Basic

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

    If (TypeOf value Is String) Then
        Try
            Dim s As String = CStr(value)
            Dim colon As Integer = s.IndexOf(":")
            Dim comma As Integer = s.IndexOf(",")

            If (colon <> -1 AndAlso comma <> -1) Then
                Dim checkWhileTyping As String = s.Substring(colon + 1, _
                                                 (comma - colon - 1))

                colon = s.IndexOf(":", comma + 1)
                comma = s.IndexOf(",", comma + 1)

                Dim checkCaps As String = s.Substring(colon + 1, _
                                                 (comma - colon - 1))
```

```
                colon = s.IndexOf(":", comma + 1)

                Dim suggCorr As String = s.Substring(colon + 1)

                Dim so As SpellingOptions = New SpellingOptions()

                so.SpellCheckWhileTyping = Boolean.Parse(checkWhileTyping)
                so.SpellCheckCAPS = Boolean.Parse(checkCaps)
                so.SuggestCorrections = Boolean.Parse(suggCorr)

                Return so
            End If
        Catch
            Throw New ArgumentException( _
                "Can not convert '" & CStr(value) & _
                                "' to type SpellingOptions")

        End Try
    End If
    Return MyBase.ConvertFrom(context, culture, value)
End Function
```

```
//C#

public override object ConvertFrom(ITypeDescriptorContext context,
                            CultureInfo culture, object value)
{
    if (value is string) {
        try {
            string s = (string) value;
            int colon = s.IndexOf(':');
            int comma = s.IndexOf(',');

            if (colon != -1 && comma != -1) {
                string checkWhileTyping = s.Substring(colon + 1 ,
                                                (comma - colon - 1));

                colon = s.IndexOf(':', comma + 1);
                comma = s.IndexOf(',', comma + 1);

                string checkCaps = s.Substring(colon + 1 ,
                                                (comma - colon -1));

                colon = s.IndexOf(':', comma + 1);

                string suggCorr = s.Substring(colon + 1);

                SpellingOptions so = new SpellingOptions();

                so.SpellCheckWhileTyping =Boolean.Parse(checkWhileTyping);
                so.SpellCheckCAPS = Boolean.Parse(checkCaps);
                so.SuggestCorrections = Boolean.Parse(suggCorr);

                return so;
            }
        }
        catch {
            throw new ArgumentException(
                "Can not convert '" + (string)value +
                                 "' to type SpellingOptions");
        }
    }
```

```
        return base.ConvertFrom(context, culture, value);
    }
```

6. Now that you have a type converter class, you need to identify the target class that will use it. You do this by applying the **TypeConverterAttribute** to the target class (the `SpellingOptions` class in your example).

```
' Visual Basic

' The TypeConverter attribute applied to the SpellingOptions class.
<TypeConverter(GetType(SpellingOptionsConverter)), _
DescriptionAttribute("Expand to see the spelling options for the application.")> _
Public Class SpellingOptions
   ...
End Class


//C#

// The TypeConverter attribute applied to the SpellingOptions class.
 [TypeConverterAttribute(typeof(SpellingOptionsConverter)),
DescriptionAttribute("Expand to see the spelling options for the application.")]
public class SpellingOptions{ ... }
```

Compile and run the options window application again. The following screen shot shows how the options window should now look.

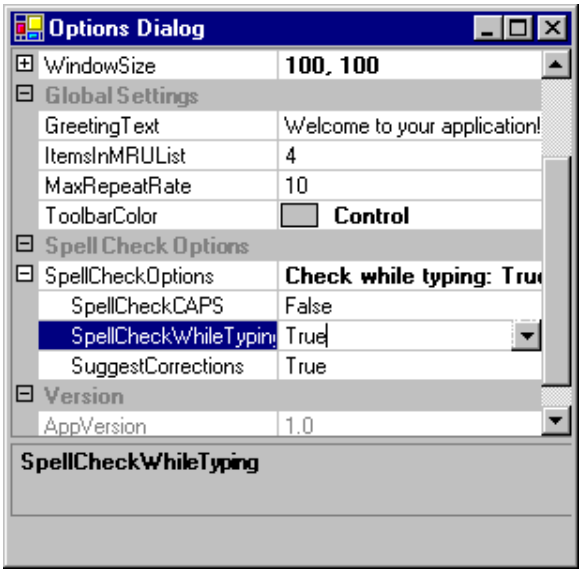

Figure 6. Custom data types with a type converter, displayed in the PropertyGrid

> **Note** If you only want the expandable object support, but not the custom string representation, you can simply apply the **TypeConverterAttribute** to your class. Specify **ExpandableObjectConverter** as the type converter type.

Adding Domain List and Simple Drop-down Property Support

For properties that return an enumeration based upon the **Enum** type, the **PropertyGrid** automatically displays the enumeration values in a drop-down list. The **EnumConverter** also provides this functionality. For you own properties, you might want to provide a list of valid values to the user, sometimes called a pick list or domain list, with types not based upon **Enum**. This is the case when the domain values are not known until run time, or when the values can change.

Modify the options window to provide a domain list of default file names that the user can choose from. You have already added the `DefaultFileName` property to the `AppSettings` class. The next step is to display the drop-down for the property in the **PropertyGrid**, in order to provide the domain list.

To provide simple drop-down property support

1. Create a class that inherits from a type converter class. Since the `DefaultFileName` property is of **String** type, you can inherit from **StringConverter**. If a type converter for the type of your property doesn't exist, you can inherit from **TypeConverter**; in this case it isn't necessary.

```
' Visual Basic

Public Class FileNameConverter
    Inherits StringConverter
End Class


//C#

public class FileNameConverter: StringConverter
{  }
```

2. Override the **GetStandardValuesSupported** method and return **true** to indicate that this object supports a standard set of values that can be picked from a list.

```
' Visual Basic

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


//C#

public override bool GetStandardValuesSupported(
                           ITypeDescriptorContext context)
{
    return true;
}
```

3. Override the **GetStandardValues** method and return a **StandardValuesCollection** filled with your standard values. One way to create a **StandardValuesCollection** is to provide an array of values in the constructor. For the options window application you can use a **String** array filled with proposed default file names.

```
' Visual Basic

Public Overloads Overrides Function GetStandardValues( _
                    ByVal context As ITypeDescriptorContext) _
                 As StandardValuesCollection

    Return New StandardValuesCollection(New String() {"New File", _
                                                      "File1", _
                                                      "Document1"})
End Function


//C#

public override StandardValuesCollection
                    GetStandardValues(ITypeDescriptorContext context)
{
    return new StandardValuesCollection(new string[]{"New File",
                                                     "File1",
                                                     "Document1"});
}
```

4. (Optional) If you want the user to be able to type in a value that is not in the drop-down list, override the **GetStandardValuesExclusive** method and return **false**. This basically changes the drop-down list style to a combo box style.

```
' Visual Basic

Public Overloads Overrides Function GetStandardValuesExclusive( _
               ByVal context As ITypeDescriptorContext) As Boolean
    Return False
End Function


//C#

public override bool GetStandardValuesExclusive(
                           ITypeDescriptorContext context)
{
    return false;
}
```

5. Now that you have your own type converter class for displaying a drop-down list, you need to identify the target that will use it. In this case the target is the `DefaultFileName` property, since the type converter is specific to the property. Apply the **TypeConverterAttribute** to the target property.

```
' Visual Basic

' The TypeConverter attribute applied to the DefaultFileName property.
<TypeConverter(GetType(FileNameConverter)), _
 CategoryAttribute("Document Settings")> _
Public Property DefaultFileName() As String
    Get
        Return _defaultFileName
    End Get
    Set(ByVal Value As String)
        _defaultFileName = Value
    End Set
End Property


//C#

// The TypeConverter attribute applied to the DefaultFileName property.
 [TypeConverter(typeof(FileNameConverter)),
 CategoryAttribute("Document Settings")]
public string DefaultFileName
{
    get{ return defaultFileName; }
    set{ defaultFileName = value; }
}
```

Compile and run the options window application again. The following screen shot shows how the options window should now look. Notice how the `DefaultFileName` property displays.

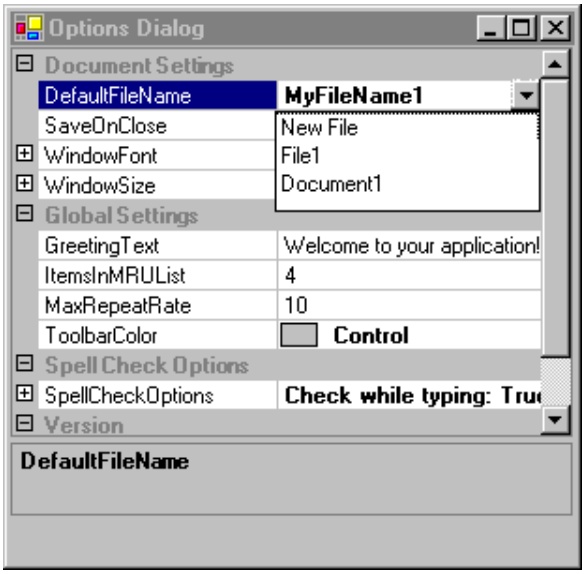

Figure 7. Displaying a drop-down domain list in the PropertyGrid

Providing a Custom UI for Your Properties

As mentioned earlier, the .NET Framework types use the **TypeConverter** and **UITypeEditor** classes (along with other classes) to provide **PropertyGrid** editing support. The Support for Custom Types section looked at using a **TypeConverter**; you can also use the **UITypeEditor** class to perform your own **PropertyGrid** customization.

You can provide a small graphical representation along with the property value in the **PropertyGrid**, similar to what is provided for the **Image** and **Color** classes. To do this for your customization, inherit from **UITypeEditor**, override **GetPaintValueSupported** and return **true**. Next, override the **UITypeEditor.PaintValue** method, and in your method, use the **PaintValueEventArgs.Graphics** parameter to draw your graphic. Finally, apply the **Editor** attribute to the class or property that uses your **UITypeEditor** class.

The following screen shot shows what your results might look like.

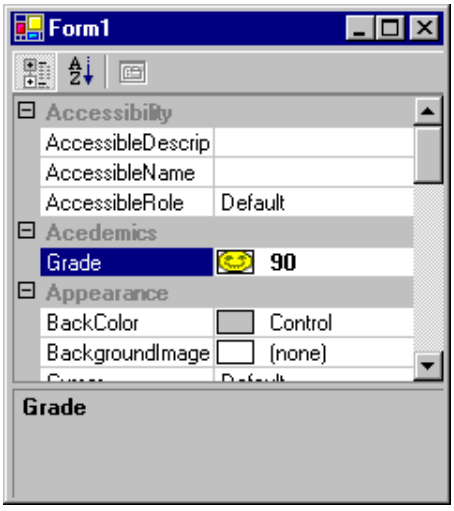

Figure 8. Displaying a custom graphic for a property in the PropertyGrid

You can also provide your own drop-down list control, similar to what the **Control.Dock** property uses to give the user docking selections. To do this, inherit from **UITypeEditor**, override **GetEditStyle**, and return a **UITypeEditorEditStyle** enum value, such as **DropDown**. Your custom drop-down control must inherit from **Control** or a **Control**-derived class (like **UserControl**). Next, override the **UITypeEditor.EditValue** method. Use the **IServiceProvider** parameter to call the **IServiceProvider.GetService** method to get an **IWindowsFormsEditorService** instance. Finally, call the **IWindowsFormsEditorService.DropDownControl** method to show your custom drop-down list control. Remember to apply the **Editor** attribute to the class or property that uses your **UITypeEditor** class.

The following screen shot shows what your results might look like.

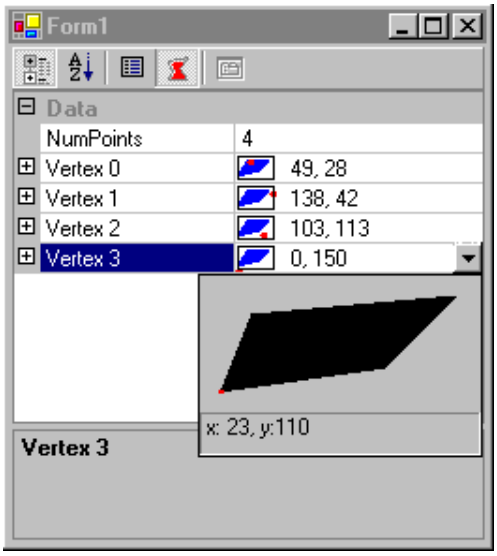

Figure 9. Displaying a custom drop-down control for a property in the PropertyGrid

In addition to using the **TypeEditor** and **UITypeEditor** classes, the **PropertyGrid** can also be customized to display additional property tabs. Property tabs inherit from the **PropertyTab** class. You might have seen a custom **PropertyTab** if you've used the property browser in Microsoft Visual C#™ .NET—the **Events** tab (the button with lightning bolt) is a custom **PropertyTab**. The following screen shot shows another example of a custom **PropertyTab** is shown below. This **PropertyTab** provides the ability to create a custom button shape by editing bounding points of a button.

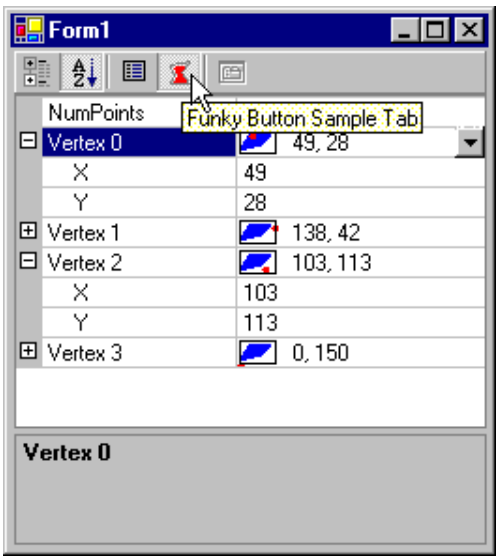

Figure 10. Displaying a custom tab in the PropertyGrid

You can find more information about using the **UITypeEditor** class to customize the **PropertyGrid**, along with code examples of the above custom user interfaces, Shawn Burke's article, Make Your Components Really RAD with Visual Studio .NET Property Browser.

Conclusion

The **ProperyGrid** control provided by the .NET Framework provides a rich editing experience that you can use to enhance your user interface. With your new knowledge on how easy the **PropertyGrid** is to customize, you will be using the **PropertyGrid** throughout your applications. In addition, since the Visual Studio .NET property browser is based upon the **PropertyGrid**, you can use these techniques to provide a richer design time experience.