ComponentOne

PropertyGrid for WPF

ComponentOne, a division of GrapeCity

201 South Highland Avenue, Third Floor

Pittsburgh, PA 15206 • USA

Internet: info@ComponentOne.com

Web site: http://www.componentone.com

Sales

E-mail: sales@componentone.com

Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

This manual was produced using ComponentOne Doc-To-Help™.

Table of Contents

ComponentOne PropertyGrid for WPF Overview

ComponentOne PropertyGrid™ for WPF is a WPF version of the popular **PropertyGrid** control that ships as part of the .NET **WinForms** platform. It allows you to easily edit any class and includes more than 10 built-in editors. Using **ComponentOne PropertyGrid™ for WPF**, users can browse and edit properties on any .NET object.

For a list of the latest features added to **ComponentOne Studio for WPF**, visit What's New in Studio for WPF.

Getting Started

Get started with the following topics:

Help with ComponentOne Studio for WPF

Getting Started

For information on installing **ComponentOne Studio for WPF**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with Studio for WPF.

What's New

For a list of the latest features added to **ComponentOne Studio for WPF**, visit What's New in Studio for WPF.

Key Features

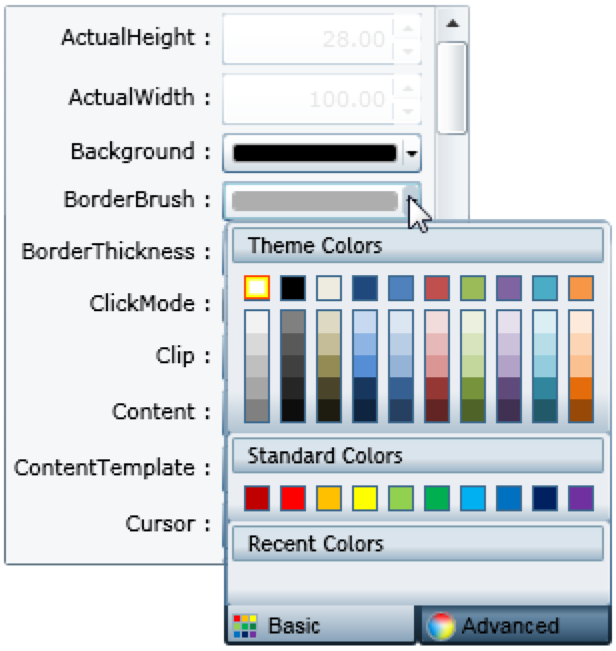
ComponentOne PropertyGrid for WPF allows you to create customized, rich applications. Make the most of **PropertyGrid for WPF** by taking advantage of the following key features:

- **Familiar Properties Window**

  The **C1PropertyGrid** control presents a familiar interface and the built-in editors make it easy to edit properties.

- **Run-time Property Editing**

  You can add the **C1PropertyGrid** control to your WPF application as part of your design and use it to allow end-users to edit properties directly.



- **Dynamic Display**

  It's as simple as adding the **C1PropertyGrid** control to your form, setting one property, and the UI is automatically built for editing objects.

PropertyGrid for WPF Quick Start

Like the standard Microsoft **PropertyGrid** control, the C1PropertyGrid control works based on a **SelectedObject** property. Once this property is set, the control displays the object's public properties and allows the user to edit them. In this quick start guide, you will define a WPF application, and C1PropertyGrid to create a user interface to display and edit information in a control.

Step 1 of 3: Creating the C1PropertyGrid Application

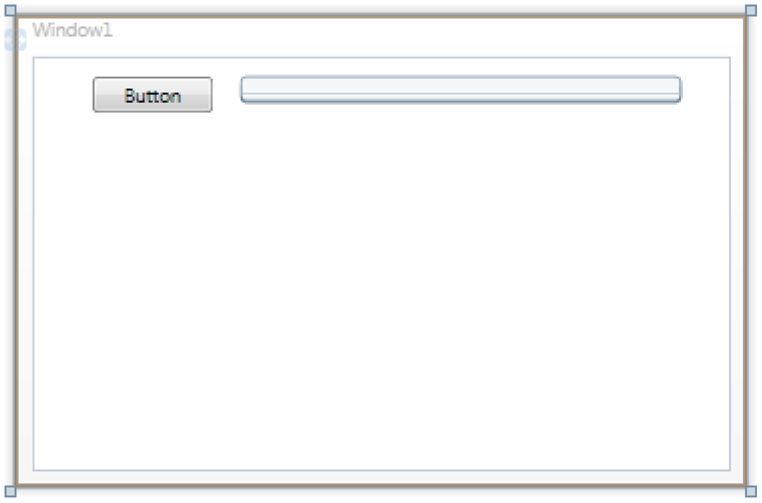
In this step you'll create a WPF application using **PropertyGrid for WPF**. When you add a C1PropertyGrid control to your application, you'll have a complete, functional standard Properties window-like interface that users can use to browse and edit properties and/or methods on any .NET object. To set up your project and add a C1PropertyGrid control to your application, complete the following steps:

1. Create a new WPF project in Visual Studio.
2. In the Solution Explorer, right-click the **References** item and choose **Add Reference**. Select the **C1.WPF**, **C1.WPF.Extended**, and **WPFToolkit** assemblies and click **OK** to add references to your project.
3. Navigate to the Visual Studio Toolbox and double-click the **C1PropertyGrid** icon to add the control to the window.
4. In the Visual Studio Toolbox, double-click the **Button** item to add a standard button control to the application. You will use the C1PropertyGrid control to set and view the button's properties.
5. Resize the Window and resize and reposition the **C1PropertyGrid** and **Button** controls in the Window.
6. Select the **Button** control and in the Properties window set its **Name** to "TestButton".
7. Select the **C1PropertyGrid** control and in the Properties window set its **Height** to **290** and **Width** to **250**.

   The XAML will appear similar to the following:

```
<Button Height="23" HorizontalAlignment="Left" Margin="48.75,12,0,0"
Name="TextButton" VerticalAlignment="Top" Width="75">Button</Button>
<c1:C1PropertyGrid Margin="130,12,30,12" Name="c1PropertyGrid1"
xmlns:c1="http://schemas.componentone.com/winfx/2006/xamld" />
```

   The page's Design view should now look similar to the following image (with the C1PropertyGrid control selected on the form):



You've successfully set up your application's user interface, but the C1PropertyGrid control contains no content. In the next step you'll set the C1PropertyGrid control to display certain properties of the **Button** control, and then you'll add code to your application to add functionality to the control.

Step 2 of 3: Customizing the C1PropertyGrid Application

In the last step you created a WPF application and added a **Button** and the C1PropertyGrid control to the application. In this step you'll customize the C1PropertyGrid control to display specific properties of the **Button** control.

To customize and connect the C1PropertyGrid control to the **Button** control, complete the following steps:

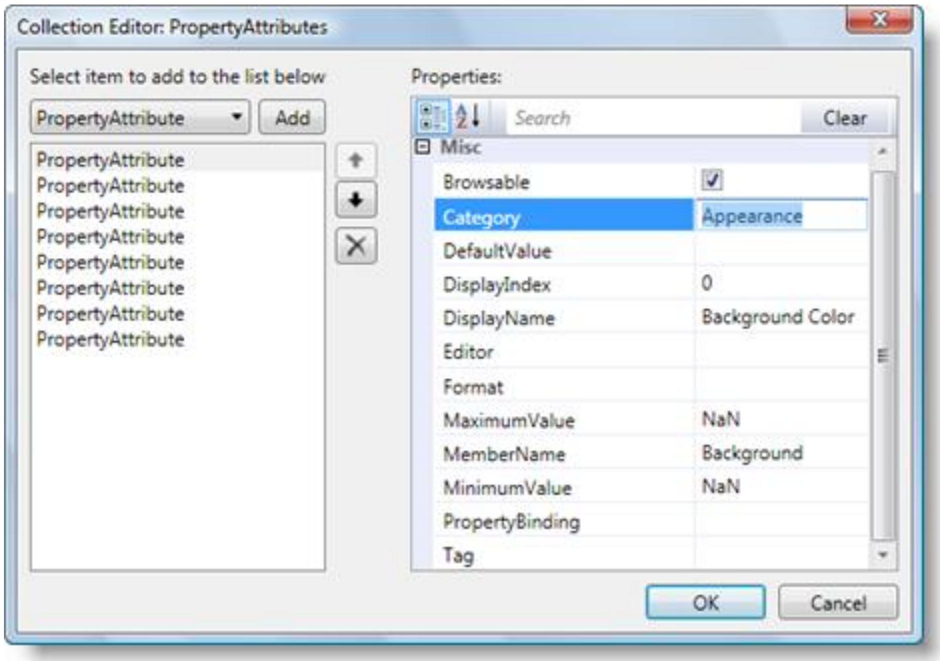
1. Switch to XAML view. You'll bind the C1PropertyGrid control to the **Button** control in XAML then customize the controls.
2. Add `SelectedObject="{Binding ElementName=TextButton, Mode=OneWay}"` to the `<c1:C1PropertyGrid>` tag so it appears similar to the following:

   ```
   <c1:C1PropertyGrid Margin="130,12,30,12" Name="c1PropertyGrid1"
   xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
   SelectedObject="{Binding ElementName=TextButton, Mode=OneWay}"/>
   ```

   You will notice in Design view that the C1PropertyGrid control now reflects all of the button's properties. In the next steps you'll customize the C1PropertyGrid control so that only certain properties are displayed.
3. In the Properties window uncheck the AutoGenerateProperties check box. Now every property will no longer be displayed – only those that you specify.
4. Locate the PropertyAttributes collection in the Properties window, and click the ellipsis button next to the item. The **Property Attribute Collection Editor** dialog box will appear.
5. In the **Property Attribute Collection Editor** dialog box, click the **Add** button. Repeat this step seven more times to create a total of eight **PropertyAttribute** items.
6. Set the following Properties in the right-side Properties pane for the items you just added:

PropertyAttribute	Category	DisplayName	MemberName
[0] PropertyAttribute	Appearance	Background Color	Background
[1] PropertyAttribute	Appearance	Border Color	BorderBrush
[2] PropertyAttribute	Appearance	Visibility	Visibility
[3] PropertyAttribute	Size	Button Height	Height
[4] PropertyAttribute	Size	Button Width	Width
[5] PropertyAttribute	Text	Button Text	Content
[6] PropertyAttribute	Text	Text Color	Foreground
[7] PropertyAttribute	Text	Text Size	FontSize

   The **Category** identifies what section the item appears in. The **DisplayName** indicates the name displayed for the item. The **MemberName** indicates the actual name of the member.


Collection Editor: PropertyAttributes

Select item to add to the list below

PropertyAttribute Add

PropertyAttribute
PropertyAttribute
PropertyAttribute
PropertyAttribute
PropertyAttribute
PropertyAttribute
PropertyAttribute
PropertyAttribute

Properties:

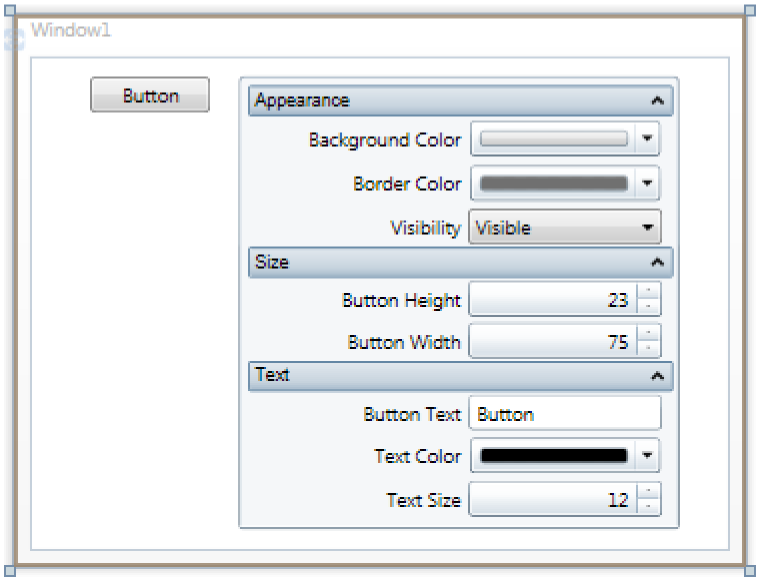
Search Clear

Misc

Browsable	✓
Category	Appearance
DefaultValue	
DisplayIndex	0
DisplayName	Background Color
Editor	
Format	
MaximumValue	NaN
MemberName	Background
MinimumValue	NaN
PropertyBinding	
Tag	

OK Cancel

7. Click the **OK** button to close the **Property Attribute Collection Editor** dialog box and change the settings. The page should now look similar to the following image at design time:


Window1

Button

Appearance

Background Color	
Border Color	
Visibility	Visible

Size

Button Height	23
Button Width	75

Text

Button Text	Button
Text Color	
Text Size	12

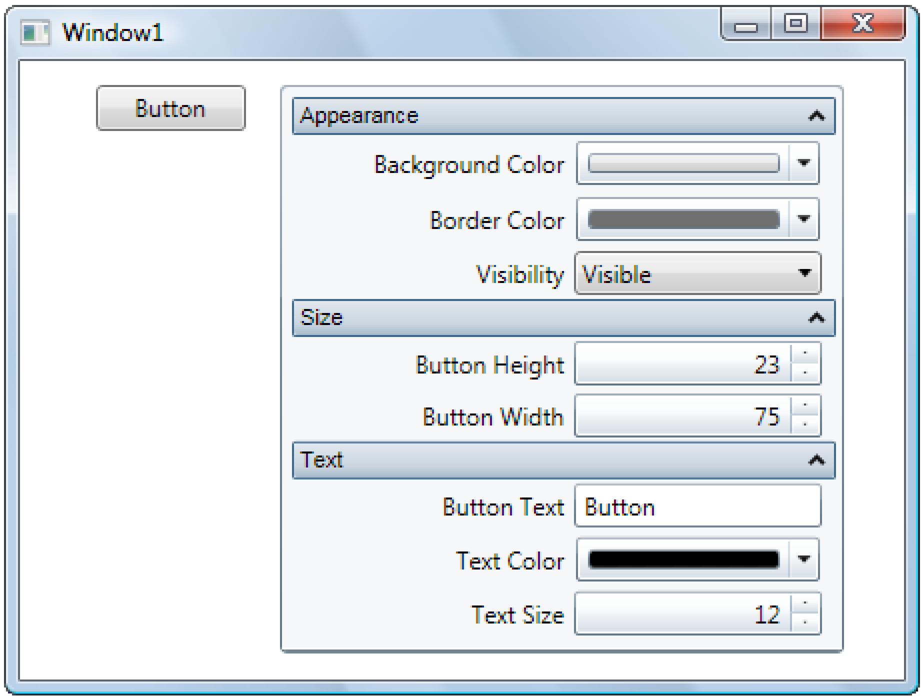
In this step you customized the C1PropertyGrid control to display specific properties of the **Button** control. In the next step, you'll run the application and view some of the possible run-time interactions.

Step 3 of 3: Running the C1PropertyGrid Application

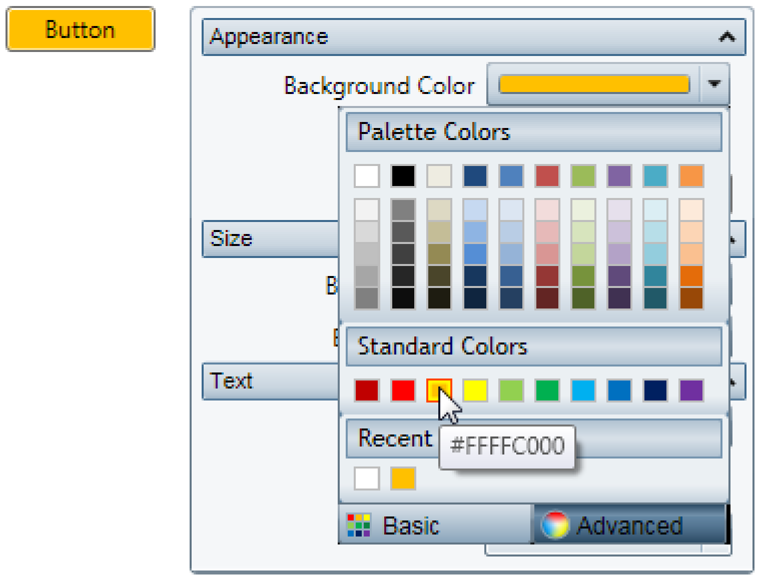
Now that you've created a WPF application and customized the application's appearance, the only thing left to do is run your application. To run your application and observe **PropertyGrid for WPF**'s run-time behavior complete the following steps:

1. From the **Debug** menu, select **Start Debugging** to view how your application will appear at run time.

   The application will appear similar to the following:

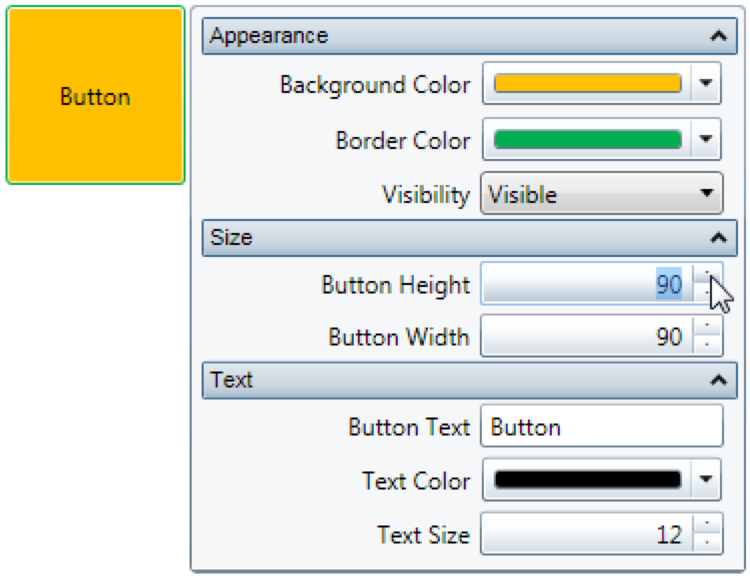


2. Click the **Background Color** drop-down arrow and pick a color, for example, orange, from the color picker that appears. The button's background color will change to your selection:



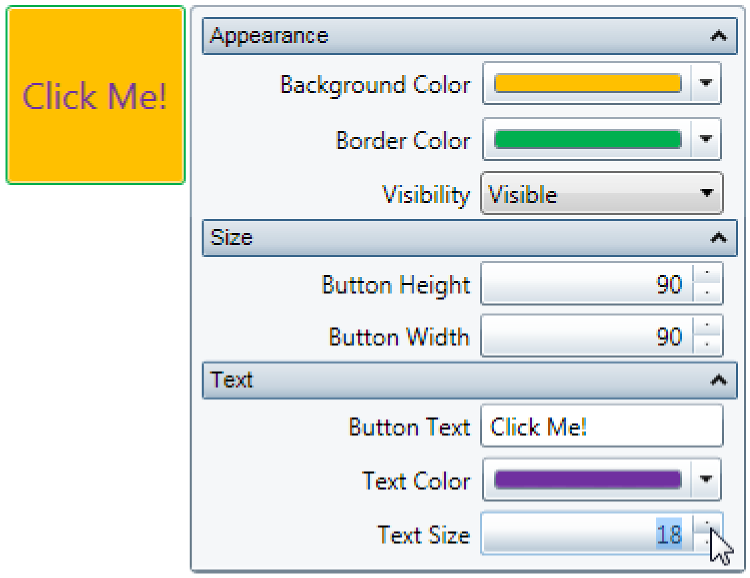
3. Click the **Border Color** drop-down arrow and pick a color, for example green, from the color picker that appears.

4. Change the size of the button by entering values in the **Button Height** and **Button Width** numeric boxes. For example enter **90** for both values. The application will appear similar to the following:



Appearance
Background Color 🟨
Border Color 🟩
Visibility Visible
Size
Button Height 90
Button Width 90
Text
Button Text Button
Text Color ⬛
Text Size 12

5. Enter a string, for example "Click Me!" in the **Button Text** box.
6. Click the **Text Color** drop-down arrow and pick a color, for example purple, from the color picker that appears.
7. Click the **Up** or **Down** arrow next to the **Text Size** value to change the size of the text that appears on the button control. For example, set the value to **18**. The application will appear similar to the following:



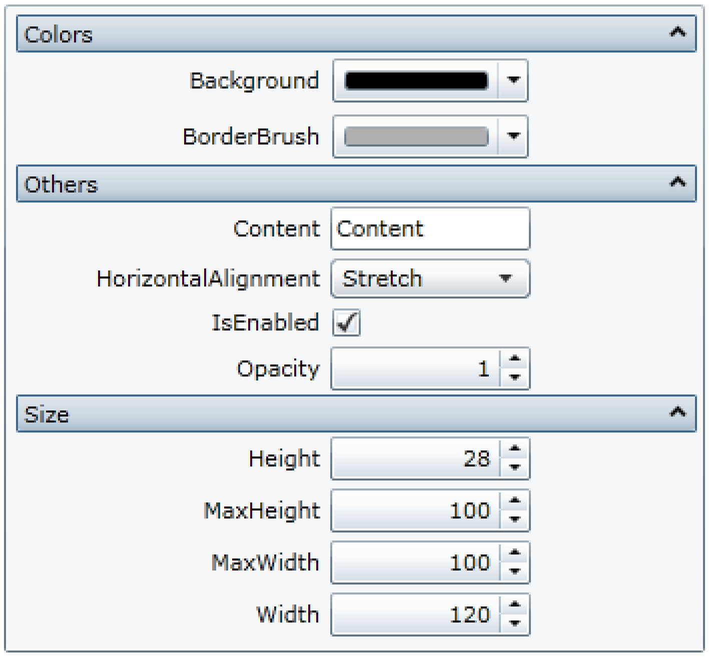
Appearance
Background Color 🟨
Border Color 🟩
Visibility Visible
Size
Button Height 90
Button Width 90
Text
Button Text Click Me!
Text Color 🟪
Text Size 18

Congratulations! You've completed the **ComponentOne PropertyGrid for WPF** quick start. In this quick start you added the C1PropertyGrid and **Button** controls to a page, linked the C1PropertyGrid control to the **Button**, customized the controls, and view the run-time interactions possible with **PropertyGrid for WPF**.

Working with PropertyGrid for WPF

ComponentOne PropertyGrid for WPF includes the C1PropertyGrid control, a simple control that lets users easily edit any class with more than 10 built-in editors. When you add the C1PropertyGrid control to a XAML window, it exists as a complete control which you can further customize.

The Basic C1PropertyGrid control appears similar to the Microsoft **PropertyGrid** control. It appears as a window in which properties and methods of an object can be edited at run time:



Colors

Background	
BorderBrush	

Others

Content	Content
HorizontalAlignment	Stretch
IsEnabled	✓
Opacity	1

Size

Height	28
MaxHeight	100
MaxWidth	100
Width	120

You can customize what appears in the control or have the content be automatically generated from an object's properties. You can customize this grid by added headers to organize content. You can also customize the appearance of the control as you could any other WPF control.

Basic Properties

ComponentOne PropertyGrid for WPF includes several properties that allow you to set the functionality of the control. Some of the more important properties are listed below. Note that you can see PropertyGrid for WPF Appearance Properties (page 15) for more information about properties that control appearance.

The following properties let you customize the C1PropertyGrid control:

Property	Description
AutoGenerateMethods	Gets or sets a value indicating whether the **C1PropertyGrid** should try to get the methods of the **SelectedObject** using reflection.
AutoGenerateProperties	Gets or sets a value indicating whether the **C1PropertyGrid** should try to get the properties of the **SelectedObject** using reflection.

AvailableEditors	The list of currently available editors to use in the **C1PropertyGrid**. The default list includes editors for: bool, Color, Brush, ColorPalette, Enum, Numeric, String, Image and Uri values.
CategoryContainers	The list of category containers (if defined) created to show each category
CategoryContainerStyle	Gets or sets the **Style** applied to all the generated category containers.
DefaultCategoryName	The name used for the default category in which the properties are placed.
DisabledCuesVisibility	Gets a value indicating whether the disabled visuals of the control are visible.
EditorWidth	Gets or sets the width of generated editors for the **SelectedObject**'s properties.
FocusCuesVisibility	Gets a value indicating whether the focus visuals of the control are visible.
LabelStyle	Gets or sets the **Style** applied to all the generated labels for the **SelectedObject**'s properties.
LabelWidth	Gets or sets the width of generated labels for the **SelectedObject**'s properties.
MethodAttributes	Gets or sets the list of method attributes used to configure the visualization of the **SelectedObject**'s methods.
MethodBoxes	Gets the **MethodBoxes** generated to show the **SelectedObject**'s methods on the **C1PropertyGrid**.
MethodsPanel	Gets or sets the template that defines the panel that controls the layout of methods.
PropertiesPanel	Gets or sets the template that defines the panel that controls the layout of properties.
PropertyAttributes	Gets or sets the list of property attributes used to configure the visualization of the **SelectedObject**'s properties.
PropertyBoxes	Gets the **PropertyBoxes** generated to show the **SelectedObject**'s properties on the **C1PropertyGrid**.
PropertySort	Gets or sets the type of sorting the **C1PropertyGrid** uses to display properties.
SelectedObject	Gets or sets the object for which the **C1PropertyGrid** displays properties. See The Selected Object (page 11) for more information.

Basic Events

ComponentOne PropertyGrid for WPF includes several events that allow you to set interaction and customize the control. Some of the more important events are listed below.

The following events let you customize the C1PropertyGrid control:

Event	**Description**
CategoryContainerAdded	Fired when a category container is added to the **C1PropertyGrid**.
CategoryContainerRemoved	Fired when a category container is removed from the **C1PropertyGrid**.

IsMouseOverChanged	Event raised when the **IsMouseOver** property has changed.
MethodBoxChanged	Fired when the generated list of method editors for the **SelectedObject** changes.
MethodBoxesLoaded	Fired when the all the method boxes in **C1PropertyGrid** were loaded.
PropertyBoxAdded	Fired when a property box is added to the **C1PropertyGrid**.
PropertyBoxChanged	Fired when the generated list of property editors for the **SelectedObject** changes.
PropertyBoxesLoaded	Fired when the all the property boxes in **C1PropertyGrid** were loaded.
PropertyBoxRemoved	Fired when a property box is removed from the **C1PropertyGrid**.
ValueChanged	Fired when the value of a property in the current **SelectedObject** changes.

The Selected Object

The SelectedObject property determines the object for which the C1PropertyGrid control displays properties to edit. You can set the SelectedObject property to any object. For example, you can connect the C1PropertyGrid control to a control as in the PropertyGrid for WPF Quick Start (page 3) or you can bind the control to a class as in the Binding C1PropertyGrid to a Class (page 21) topic.

In XAML, you would use a binding statement to connect the C1PropertyGrid control to an object. For example, the following C1PropertyGrid control is linked to a button object:

```
<c1:C1PropertyGrid Margin="244,152,186,168" SelectedObject="{Binding
ElementName=button, Mode=OneWay}"/>
```

You can also set the SelectedObject property in Design view in Blend by selecting the square Advanced Properties icon next to the SelectedObject item in the Properties window and selecting **Data Binding**. The **Create Data Binding** dialog box will appear allowing you to choose an object to bind to.

Automatically Generating Properties and Methods

By default when you set the SelectedObject property to bind to an object, the members listed in the C1PropertyGrid control will be automatically generated. This is because the AutoGenerateProperties property is set to **True** by default. You can set the SelectedObject property to **False** if you do not want properties to be automatically generated. You might do so, for example, if you want only a few properties to be editable or you want to customize the way properties appear in the C1PropertyGrid window.

While properties are visible by default, however, methods are not. By default the AutoGenerateMethods property is set to **False** and methods are not automatically displayed. If you choose, you can automatically generate methods by setting the AutoGenerateMethods property to **True**.

Sorting Members in C1PropertyGrid

By default properties and methods are listed alphabetically in the C1PropertyGrid control, similar to the **Alphabetic** view in the Visual Studio Properties window. However, you can customize the way members are listed by setting the PropertySort property. The C1PropertyGrid control can sort the properties in any of the following ways:

- **Alphabetical**: the properties are sorted in an alphabetical list. This is the default setting and appears similar to the **Alphabetic** view in the Visual Studio Properties window.

- **Categorized**: the categories are displayed in an alphabetical list, the properties in each category are displayed in no particular order (the order in which are retrieved from the **SelectedObject**).
- **CategorizedAlphabetical**: the categories are displayed in an alphabetical list; the properties in each category are displayed in alphabetical order. This appears similar to the **Categorized** view in the Visual Studio Properties window.
- **CategorizedCustom**: the categories are displayed in an alphabetical list; the properties inside each category are displayed in a custom order defined by the user using the **Display.Order** attribute.
- **Custom**: the properties are displayed in custom order defined by the user using the **Display.Order** attribute.
- **NoSort**: properties are displayed in the order in which they are retrieved from the **SelectedObject**.

Set the PropertySort property to one of the above options to customize the way the property grid is sorted.

Built-in Editors

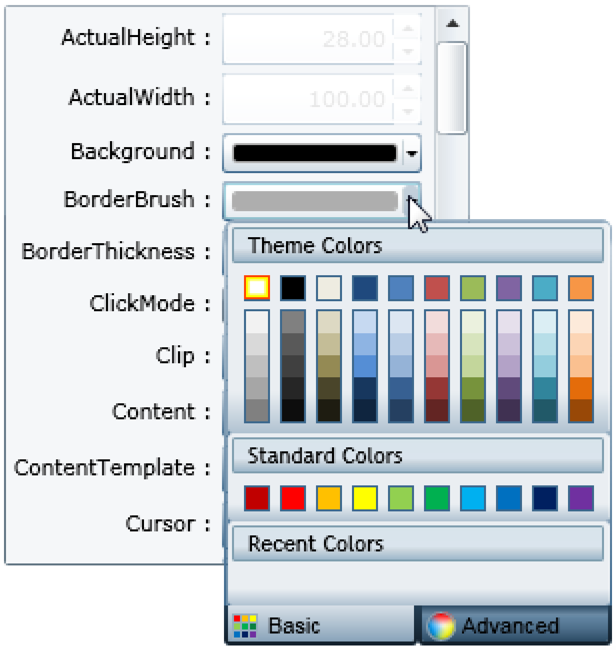
The C1PropertyGrid control includes several built-in editors. If you do not specify an editor, C1PropertyGrid will display each member with a default editor. You can also specify editors, and if needed, create your own custom editor. The AvailableEditors property controls what editors are available in the C1PropertyGrid control.

Because the C1PropertyGrid control does not have built-in editors for complex objects, complex object will be shown in the default editor (StringEditor) and the string representation (tostring()) of the complex object will be shown. If you need to show complex objects, you may want to create a custom editor. For more information about creating a custom editor, see Creating Custom Editors (page 31).

Built-in editors include:

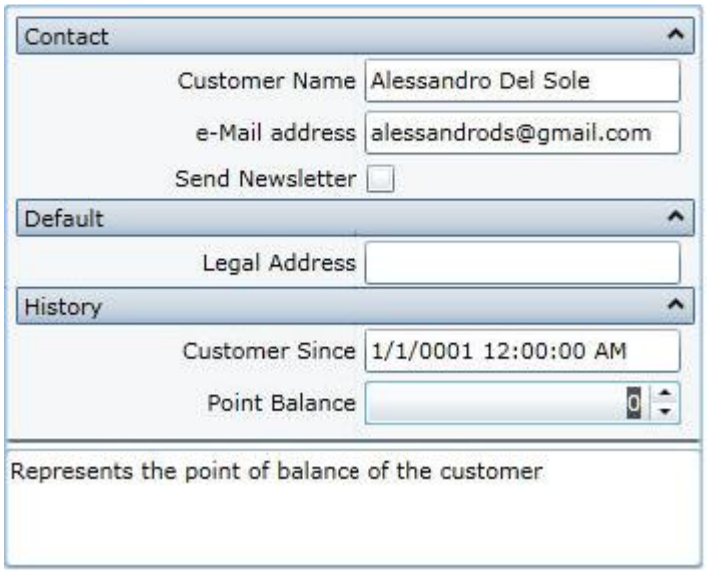
Editor	**Description**
BoolEditor	Default editor used by C1PropertyGrid to edit bool values.
BrushEditor	Default editor used by C1PropertyGrid to edit Brush values.
ColorEditor	Default editor used by C1PropertyGrid to edit color values.
ColorPaletteEditor	Default editor used by C1PropertyGrid to edit color palette values.
EnumEditor	Default editor used by C1PropertyGrid to edit Enum values.
ImageSourceEditor	Default editor used by C1PropertyGrid to edit image values.
NumericEditor	Default editor used by C1PropertyGrid to edit numeric values.
StringEditor	Default editor used by C1PropertyGrid to edit String values.
UriEditor	Default editor used by C1PropertyGrid to edit Uri values.

For example, color values are editable at run time in the following image:



Showing Property Descriptions

By default the C1PropertyGrid control does not display descriptions for properties users can edit at run time. However, in some case, you may want to display additional information about the properties being edited. The C1PropertyGrid control allows you to show a description for each property, similar to the Description pane in the Visual Studio Properties window. You can make property descriptions visible by setting the ShowDescription property to **True**. This will add a description area at the bottom of the C1PropertyGrid control, showing the description for the property currently focused. The description, which can be added using the **Display.Description** attribute, appears similar to the following image:



Contact

Customer Name	Alessandro Del Sole
e-Mail address	alessandrods@gmail.com
Send Newsletter	☐

Default

Legal Address	

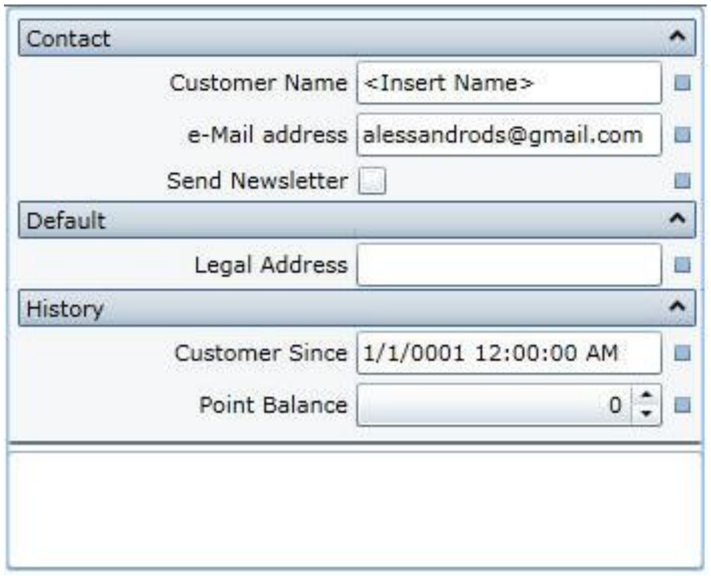
History

Customer Since	1/1/0001 12:00:00 AM
Point Balance	0

Represents the point of balance of the customer

Resetting Property Values

C1PropertyGrid provides a reset button that allows users to reset a changed property to its default value (specified using the **DefaultValue** attribute). The reset button is small rectangle next to the property editor:



By default, the **Reset** button is not shown, but you can make the **Reset** button visible by setting the ShowResetButton property to **True**. It's important to note that the **SelectedObject** should implement the **INotifyPropertyChanged** interface so it can reflect the updated value; this is because bound properties must notify value changes to the editor so it can refresh values.

Supported PropertyGrid Attributes

You can use attributes to customize the display and content of items displayed in the property grid. The C1PropertyGrid control supports several attributes, including the following:

- **Display.Name**: sets the text label displayed for each property
- **Display.Order**: used to specify the order in which properties are shown (when using custom sort)
- **Display.Description**: used to set a description for the property; this description will be shown at the bottom of the C1PropertyGrid, when the property that is being edited gets the focus.
- **DefaultValue**: used to set a default value for the property; the default value will be applied when the property has no other value (it was not initialized or has a null value) or when the **Reset** button is pressed.
- **Editor**: used to set a custom editor for the current property (override the editor assigned by default).
- **MaximumValue**: used to set a maximum value for a property, the MaximumValue will be taken into consideration only for those editors where it makes sense (the numeric editor, for example).
- **MinimumValue**: used to set a minimum value for a property, the MinimumValue will be taken into consideration only for those editors where it makes sense (the numeric editor, for example).
- **Browsable**: if set to false, the property will not be shown in the C1PropertyGrid.
- **Category**: used to set the category in which the property will be shown.

- **PropertyBinding**: used to override the default binding between the property and its editor; this is useful, for example, if you want to set your own converter or if you don't want the default validation for properties.
- **DisplayFormat**: used to specify a format to show the property value, the **DisplayFormat** will be taken into consideration only for those editors where it makes sense (the numeric editor, for example).
- **ReadOnly**: if set to **True**, the property will be shown in the C1PropertyGrid but it won't be possible to change its value.

C1PropertyGrid Layout and Appearance

The following topics detail how to customize the C1PropertyGrid control's layout and appearance. You can use built-in layout options to lay your controls out in panels such as Grids or Canvases. Themes allow you to customize the appearance of the grid and take advantage of WPF's XAML-based styling. You can also use templates to format and lay out the control and to customize the control's actions.

PropertyGrid for WPF Appearance Properties

ComponentOne PropertyGrid for WPF includes several properties that allow you to customize the appearance of the control. You can change the color, border, and height of the control. The following topics describe some of these appearance properties.

Color Properties

The following properties let you customize the colors used in the control itself:

Property	Description
Background	Gets or sets a brush that describes the background of a control. This is a dependency property.
Foreground	Gets or sets a brush that describes the foreground color. This is a dependency property.

Alignment Properties

The following properties let you customize the control's alignment:

Property	Description
HorizontalAlignment	Gets or sets the horizontal alignment characteristics applied to this element when it is composed within a parent element, such as a panel or items control. This is a dependency property.
VerticalAlignment	Gets or sets the vertical alignment characteristics applied to this element when it is composed within a parent element such as a panel or items control. This is a dependency property.

Border Properties

The following properties let you customize the control's border:

Property	Description
BorderBrush	Gets or sets a brush that describes the border background of a control. This is a dependency property.
BorderThickness	Gets or sets the border thickness of a control. This is a dependency property.

Size Properties

The following properties let you customize the size of the **C1PropertyGrid** control:

Property	Description
Height	Gets or sets the suggested height of the element. This is a dependency property.
MaxHeight	Gets or sets the maximum height constraint of the element. This is a dependency property.
MaxWidth	Gets or sets the maximum width constraint of the element. This is a dependency property.
MinHeight	Gets or sets the minimum height constraint of the element. This is a dependency property.
MinWidth	Gets or sets the minimum width constraint of the element. This is a dependency property.
Width	Gets or sets the width of the element. This is a dependency property.

ComponentOne ClearStyle Technology

ComponentOne ClearStyle™ technology is a new, quick and easy approach to providing Silverlight and WPF control styling. ClearStyle allows you to create a custom style for a control without having to deal with the hassle of XAML templates and style resources.

Currently, to add a theme to all standard WPF controls, you must create a style resource template. In Microsoft Visual Studio this process can be difficult; this is why Microsoft introduced Expression Blend to make the task a bit easier. Having to jump between two environments can be a bit challenging to developers who are not familiar with Blend or do not have the time to learn it. You could hire a designer, but that can complicate things when your designer and your developers are sharing XAML files.

That's where ClearStyle comes in. With ClearStyle the styling capabilities are brought to you in Visual Studio in the most intuitive manner possible. In most situations you just want to make simple styling changes to the controls in your application so this process should be simple. For example, if you just want to change the row color of your data grid this should be as simple as setting one property. You shouldn't have to create a full and complicated-looking template just to simply change a few colors.

How ClearStyle Works

Each key piece of the control's style is surfaced as a simple color property. This leads to a unique set of style properties for each control. For example, a **Gauge** has **PointerFill** and **PointerStroke** properties, whereas a **DataGrid** has **SelectedBrush** and **MouseOverBrush** for rows.

Let's say you have a control on your form that does not support ClearStyle. You can take the XAML resource created by ClearStyle and use it to help mold other controls on your form to match (such as grabbing exact colors). Or let's say you'd like to override part of a style set with ClearStyle (such as your own custom scrollbar). This is also possible because ClearStyle can be extended and you can override the style where desired.

ClearStyle is intended to be a solution to quick and easy style modification but you're still free to do it the old fashioned way with ComponentOne's controls to get the exact style needed. ClearStyle does not interfere with those less common situations where a full custom design is required.

ClearStyle Properties

The following table lists all of the ClearStyle-supported properties in the C1PropertyGrid control as well as a description of the property:

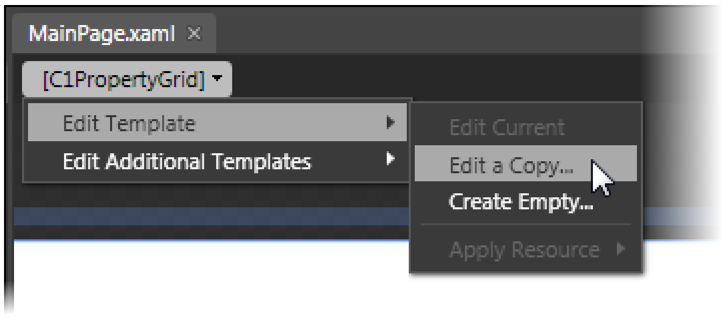
Property	Description
Background	Gets or sets a brush that describes the background of a control. The default **Background** color is LightBlue.
CategoryBackground	A brush used to define the appearance of the control's category background.
CategoryForeground	A brush used to define the appearance of the control's category foreground.
MouseOverBrush	A brush used to define the appearance of the control, when the control is moused over.
PressedBrush	A brush used to define the appearance of the control, when the control is pressed.

PropertyGrid Templates

One of the main advantages to using a WPF control is that controls are "lookless" with a fully customizable user interface. Just as you design your own user interface (UI), or look and feel, for WPF applications, you can provide your own UI for data managed by **ComponentOne PropertyGrid for WPF**. Extensible Application Markup Language (XAML; pronounced "Zammel"), an XML-based declarative language, offers a simple approach to designing your UI without having to write code.

Accessing Templates

You can access templates in Microsoft Expression Blend by selecting the C1PropertyGrid control and, in the menu, selecting **Edit Control Parts (Templates)**. Select **Edit a Copy** to create an editable copy of the current template or **Create Empty**, to create a new blank template.



Note: If you create a new template through the menu, the template will automatically be linked to that template's property. If you manually create a template in XAML you will have to link the appropriate template property to the template you've created.

Note that you can use the Template property to customize the template.

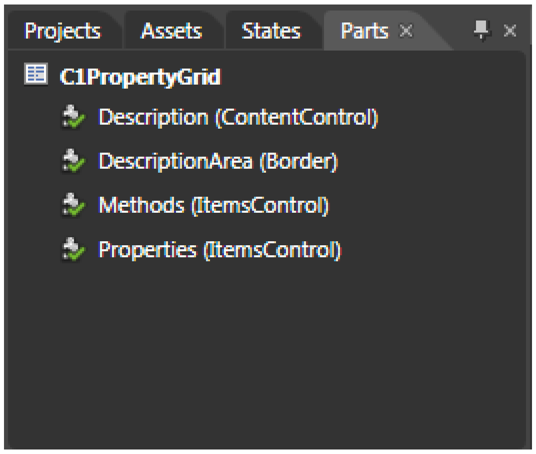
PropertyGrid Styles

ComponentOne PropertyGrid for WPF's **C1PropertyGrid** control provides several style properties that you can use to change the appearance of the control. Some of the included styles are described in the table below:

Style	**Description**
CategoryContainerStyle	Gets or sets the **Style** applied to all the generated category containers.
FontStyle	Gets or sets the font style. This is a dependency property.
LabelStyle	Gets or sets the Style applied to all the generated labels for the **SelectedObject**'s properties.
Style	Gets or sets the style used by this element when it is rendered. This is a dependency property.

PropertyGrid Template Parts

In Microsoft Expression Blend, you can view and edit template parts by creating a new template (for example, click the **C1PropertyGrid** control to select it and choose **Object | Edit Template | Edit a Copy**). Once you've created a new template, the parts of the template will appear in the **Parts** window:



Note that you may have to select the **ControlTemplate** for its parts to be visible in the **Parts** window. In the Parts window, you can double-click any element to create that part in the template.

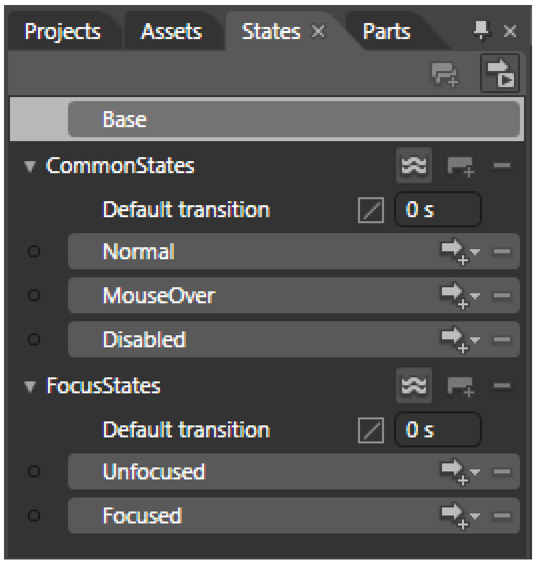
Template parts available in the **C1PropertyGrid** control include:

Name	**Type**	**Description**
Description	ContentControl	Represents a control with a single piece of content. Here, it represents the description area at the bottom of the control.
DescriptionArea	Border	Draws a border, background, or both around another element. Here the border surrounds the description area at the bottom of the control.

Methods	ItemsControl	Represents a control that can be used to present a collection of items. Here it represents the collection of methods.
Properties	ItemsControl	Represents a control that can be used to present a collection of items. Here it represents the collection of properties.

PropertyGrid Visual States

In Microsoft Expression Blend, you can add custom states and state groups to define a different appearance for each state of your user control – for example, the visual state of the control could change on mouse over. You can view and edit visual states by creating a new template and adding a new template part (page 18). Once you've done so the available visual states for that part will be visible in the **Visual States** window:



Common states include **Normal** for the normal appearance of the item, **MouseOver** for the item on mouse over, and **Disabled** for when the item is not enabled. Focus states include **Unfocused** for when the item is not in focus and **Focused** when the item is in focus.

PropertyGrid for WPF Samples

Please be advised that this ComponentOne software tool is accompanied by various sample projects and/or demos, which may make use of other ComponentOne development tools included with the ComponentOne Studios. Samples can be accessed from the **ComponentOne Studio for WPF ControlExplorer**. To view samples, on your desktop, click the **Start** button and then click **All Programs | ComponentOne | Studio for WPF | Samples | WPF ControlExplorer**.

C# Samples

The following C# sample is included:

Sample	Description
ControlExplorer	The **PropertyGrid** page in the **ControlExplorer** sample demonstrates how to add content to and customize the C1PropertyGrid control.

PropertyGrid for WPF Task-Based Help

The following task-based help topics assume that you are familiar with Visual Studio and Expression Blend and know how to use the C1PropertyGrid control in general. If you are unfamiliar with the **ComponentOne PropertyGrid for WPF** product, please see the PropertyGrid for WPF Quick Start (page 3) first.

Each topic in this section provides a solution for specific tasks using the **ComponentOne PropertyGrid for WPF** product. Most task-based help topics also assume that you have created a new WPF project and added a C1PropertyGrid control to the project.

Binding C1PropertyGrid to a Class

PropertyGrid for WPF allows you to easily bind the control to a class. At run time items in the class can be browsed and edited using the C1PropertyGrid control. For example, add a simple **Customer** class to your project defined as follows:

- Visual Basic

```
Private _Name As String
    Public Property Name() As String
        Get
            Return _Name
        End Get
        Set(ByVal value As String)
            _Name = value
        End Set
    End Property
Private _EMail As String
    Public Property EMail() As String
        Get
            Return _EMail
        End Get
        Set(ByVal value As String)
            _EMail = value
        End Set
    End Property
Private _Address As String
```

```
    Public Property Address() As String
        Get
            Return _Address
        End Get
        Set(ByVal value As String)
            _Address = value
        End Set
    End Property
Private _CustomerSince As DateTime
    Public Property CustomerSince() As DateTime
        Get
            Return _CustomerSince
        End Get
        Set(ByVal value As DateTime)
            _CustomerSince = value
        End Set
    End Property
Private _SendNewsletter As Boolean
    Public Property SendNewsletter() As Boolean
        Get
            Return _SendNewsletter
        End Get
        Set(ByVal value As Boolean)
            _SendNewsletter = value
        End Set
    End Property
Private _PointBalance As System.Nullable(Of Integer)
    Public Property PointBalance() As System.Nullable(Of Integer)
        Get
            Return _PointBalance
        End Get
        Set(ByVal value As System.Nullable(Of Integer))
            _PointBalance = value
        End Set
    End Property
End Class
```

- C#

```
public class Customer
{
  public string Name { get; set; }
  public string EMail { get; set; }
  public string Address { get; set; }
  public DateTime CustomerSince { get; set; }
  public bool SendNewsletter { get; set; }
  public int? PointBalance { get; set; }
}
```

You could build a user interface to display and edit customers using the following code:

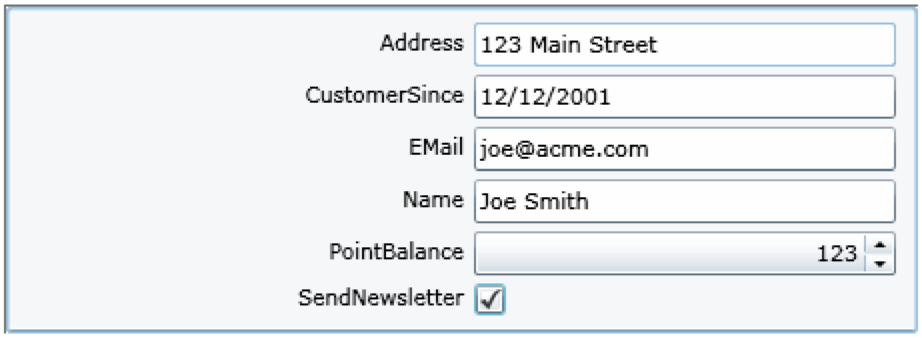
- Visual Basic

```
Public Sub New()
    InitializeComponent()

    ' Create object to browse
    Dim customer = New Customer()

    ' Create C1PropertyGrid
    Dim pg = New C1PropertyGrid()
    LayoutRoot.Children.Add(pg)

    ' Show customer properties
    pg.SelectedObject = customer
End Sub
```

- C#

```
public Page()
{
  InitializeComponent();

  // Create object to browse
  var customer = new Customer();

  // Create C1PropertyGrid
  var pg = new C1PropertyGrid();
  LayoutRoot.Children.Add(pg);

  // Show customer properties
  pg.SelectedObject = customer;
}
```

Run the application and observe that the resulting application would look similar to the following:



Address: 123 Main Street
CustomerSince: 12/12/2001
EMail: joe@acme.com
Name: Joe Smith
PointBalance: 123
SendNewsletter: ✓

This simple UI allows users to edit all the properties in the **Customer** objects. It was built automatically based on the object's properties and will be automatically updated if you add or modify the properties in the **Customer** class.

Notice that properties are shown in alphabetical order by default. You can change this by setting the PropertySort property; for more information see Sorting Members in C1PropertyGrid (page 11).

Customizing the Control Layout

The first aspect of the control that you may want to customize is the layout. The control presents two columns, one with labels and one with editors. The columns have the same size by default, but you can change that by changing the value of the LabelWidth and EditorWidth properties.

For example, you could make the label column narrower in the example above by adding one line of code:

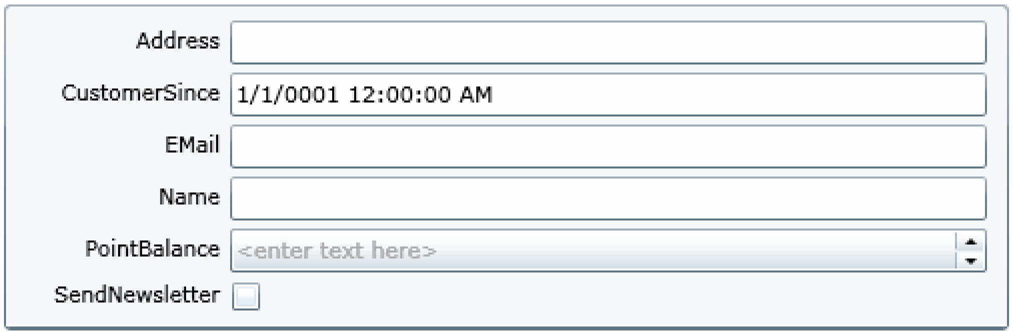
- Visual Basic

```
Public Sub New()
    InitializeComponent()

    ' Create object to browse
    Dim customer = New Customer()
    ' Create C1PropertyGrid
    Dim pg = New C1PropertyGrid()
    LayoutRoot.Children.Add(pg)

          ' Customize the PropertyGrid layout
          pg.LabelWidth = 100


    ' Show customer properties
    pg.SelectedObject = customer
End Sub
```

- C#

```
public Page()
{
  InitializeComponent();

  // Create object to browse
  var customer = new Customer();

  // Create C1PropertyGrid
  var pg = new C1PropertyGrid();
  LayoutRoot.Children.Add(pg);

        // Customize the PropertyGrid layout
        pg.LabelWidth = 100;

  // Show customer properties
  pg.SelectedObject = customer;
}
```

The result would be as shown below:



Address	
CustomerSince	1/1/0001 12:00:00 AM
EMail	
Name	
PointBalance	<enter text here>
SendNewsletter	☐

As you can see, the label column is now narrower and more room is left for the editor part. If you resize the form, you will notice that the width of the label column remains constant.

Customizing Display Names

By default, the labels shown next to each property display the property name. This works fine in many cases, but you may want to customize the display to provide more descriptive names. The easiest way to achieve this is to decorate the properties on the object with custom attributes and by setting the **Name** property in the **Display** attribute (note that the Display attribute is defined in the System.ComponentModel.DataAnnotations namespace, in the System.ComponentModel,DataAnnotations assembly).

For example, you could define the **Display** attribute in the class itself and set the value for the **Name** property as in the following code:

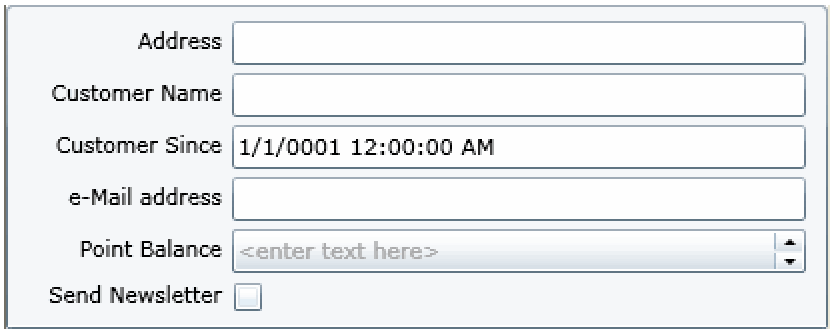
- Visual Basic

```
Public Class Customer
Private _Name As String
    <Display(Name:="Customer Name")> _
    Public Property Name() As String
        Get
            Return _Name
        End Get
        Set(ByVal value As String)
            _Name = value
        End Set
    End Property

Private _EMail As String
    <Display(Name:="e-Mail address")> _
    Public Property EMail() As String
        Get
            Return _EMail
        End Get
        Set(ByVal value As String)
            _EMail = value
        End Set
    End Property

Private _Address As String
    Public Property Address() As String
        Get
            Return _Address
        End Get
        Set(ByVal value As String)
            _Address = value
        End Set
    End Property

Private _CustomerSince As DateTime
    <Display(Name:="Customer Since")> _
    Public Property CustomerSince() As DateTime
        Get
            Return _CustomerSince
        End Get
        Set(ByVal value As DateTime)
            _CustomerSince = value
        End Set
    End Property

Private _SendNewsletter As Boolean
    <Display(Name:="Send Newsletter")> _
    Public Property SendNewsletter() As Boolean
        Get
            Return _SendNewsletter
        End Get
        Set(ByVal value As Boolean)
            _SendNewsletter = value
        End Set
    End Property

Private _PointBalance As System.Nullable(Of Integer)
    <Display(Name:="Point Balance")> _
    Public Property PointBalance() As System.Nullable(Of Integer)
        Get
            Return _PointBalance
        End Get
        Set(ByVal value As System.Nullable(Of Integer))
            _PointBalance = value
        End Set
    End Property
End Class
```

- C#

```
public class Customer
{
  [Display(Name = "Customer Name")]
  public string Name { get; set; }

  [Display(Name = "e-Mail address")]
  public string EMail { get; set; }

  public string Address { get; set; }

  [Display(Name = "Customer Since")]
  public DateTime CustomerSince { get; set; }

  [Display(Name ="Send Newsletter")]
  public bool SendNewsletter { get; set; }

  [Display(Name ="Point Balance")]
  public int? PointBalance { get; set; }
}
```

The **C1PropertyGrid** uses this additional information and displays the customer as shown below:



Address	
Customer Name	
Customer Since	1/1/0001 12:00:00 AM
e-Mail address	
Point Balance	<enter text here>
Send Newsletter	☐

This method requires that you have access to the class being displayed in the C1PropertyGrid. If you want to change the display strings but cannot modify the class being shown, then you would have to use the PropertyAttributes property to provide explicit information about each property you want to show on the C1PropertyGrid.

Categorizing Properties

You can group properties by category by adding a **Category** attribute to each property on the object being browsed (note that the **Category** attribute is defined in the System.ComponentModel namespace, in the System.Windows assembly). By default the members in the list will be listed alphabetically and uncategorized. By adding categories you can better organize members by listing related members together.

Continuing with our example, here's a revised version of the **Customer** class which includes categories:

- Visual Basic

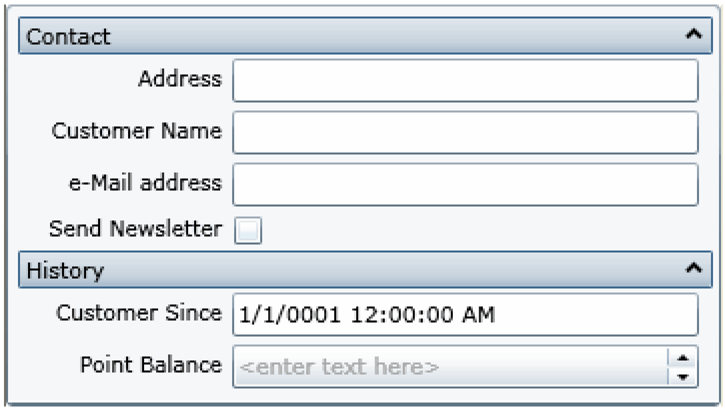
```
Public Class Customer
Private _Name As String
    <Category("Contact")> _
    <DisplayName("Customer Name")> _
    Public Property Name() As String
        Get
            Return _Name
        End Get
        Set(ByVal value As String)
            _Name = value
        End Set
    End Property

Private _EMail As String
    <Category("Contact")> _
    <DisplayName("e-Mail address")> _
    Public Property EMail() As String
        Get
            Return _EMail
        End Get
        Set(ByVal value As String)
            _EMail = value
        End Set
    End Property

Private _Address As String
    <Category("Contact")> _
    Public Property Address() As String
        Get
            Return _Address
        End Get
        Set(ByVal value As String)
            _Address = value
        End Set
    End Property

Private _CustomerSince As DateTime
    <Category("History")> _
    <DisplayName("Customer Since")> _
    Public Property CustomerSince() As DateTime
        Get
            Return _CustomerSince
        End Get
        Set(ByVal value As DateTime)
            _CustomerSince = value
        End Set
    End Property
```

```
Private _SendNewsletter As Boolean
    <Category("Contact")> _
    <DisplayName("Send Newsletter")> _
    Public Property SendNewsletter() As Boolean
        Get
            Return _SendNewsletter
        End Get
        Set(ByVal value As Boolean)
            _SendNewsletter = value
        End Set
    End Property

Private _PointBalance As System.Nullable(Of Integer)
    <Category("History")> _
    <DisplayName("Point Balance")> _
    Public Property PointBalance() As System.Nullable(Of Integer)
        Get
            Return _PointBalance
        End Get
        Set(ByVal value As System.Nullable(Of Integer))
            _PointBalance = value
        End Set
    End Property
End Class
```

- C#

```
public class Customer
{
  [Category("Contact")]
  [DisplayName("Customer Name")]
  public string Name { get; set; }

  [Category("Contact")]
  [DisplayName("e-Mail address")]
  public string EMail { get; set; }

  [Category("Contact")]
  public string Address { get; set; }

  [Category("History")]
  [DisplayName("Customer Since")]
  public DateTime CustomerSince { get; set; }

  [Category("Contact")]
  [DisplayName("Send Newsletter")]
  public bool SendNewsletter { get; set; }

  [Category("History")]
  [DisplayName("Point Balance")]
  public int? PointBalance { get; set; }
}
```

And here is the result of this change:



Notice how properties are neatly grouped by category. Each group can be expanded and collapsed, making it easier for the user to find specific properties.

You can also use the DefaultCategoryName property to set the name of a default category which will contain all the properties that have no other category defined.

Displaying Methods and Properties

When the value of the **SelectedObject** property changes, the **C1PropertyGrid** updates itself based on the settings of three properties:

Property	Setting
PropertyAttributes	If this property is set to a non-empty collection, then the collection is used to generate the UI. Only the properties or methods included in the collection are displayed as long as the AutoGenerateProperties property is set to **False**. Otherwise all the properties will be shown and this collection will be used to override the default display of the properties or methods.
AutoGenerateProperties	If the **PropertyAttributes** collection is empty and this property is set to **True** (the default), then the C1PropertyGrid automatically shows all public properties of the **SelectedObject**.
AutoGenerateMethods	If the **PropertyAttributes** collection is empty and this property is set to **True**, then the **C1PropertyGrid** automatically shows all public methods of the **SelectedObject** that take no parameters. Methods are shown as buttons which can be clicked to invoke the method.

For example, the following code would cause the **C1PropertyGrid** control to show one group with two entries in it for customer name and e-mail address:

- Visual Basic

```
' Create C1PropertyGrid
Dim pg = New C1PropertyGrid()
LayoutRoot.Children.Add(pg)

' Customize the C1PropertyGrid layout
pg.LabelWidth = 100

' Show customer properties
pg.SelectedObject = customer

' Customize what is shown to the user
pg.AutoGenerateProperties = False
pg.PropertyAttributes.Add(New PropertyAttribute())
pg.PropertyAttributes.Add(New PropertyAttribute())
```

- C#

```
// Create C1PropertyGrid
var pg = new C1PropertyGrid();
LayoutRoot.Children.Add(pg);

// Customize the C1PropertyGrid layout
pg.LabelWidth = 100;

// Show customer properties
pg.SelectedObject = customer;

// Customize what is shown to the user
pg.AutoGenerateProperties = false;
pg.PropertyAttributes.Add(new PropertyAttribute()
  {
    MemberName = "Name",
    DisplayName = "Customer Name",
    Category = "Contact"
  }
);
pg.PropertyAttributes.Add(new PropertyAttribute()
  {
    MemberName = "EMail",
    DisplayName = "e-Mail Address",
    Category = "Contact"
  }
);
```

You can use this method to customize the display of objects that you have no access to. For example, if your **Customer** class were defined in a third-party assembly, you would not be able to add attributes to its members but would still be able to customize how the object is shown in the **C1PropertyGrid**.

Note that the **MemberName** must match the exact name of a property or method; otherwise the entry will be ignored.

Customizing the Editors

The **C1PropertyGrid** control has a set of built-in editors which support all common data types: **string**, **numeric**, **bool**, **Enum**, **Color**, **Brush**, **Image**, and so on.

The most suitable editor is automatically selected depending on the type of the property. When no suitable editor is found for the current property type, the **string** editor is used by default.

The list of editors is exposed by the **AvailableEditors** property of the **C1PropertyGrid** class. You can add your own custom editors to this list. The next section, Creating Custom Editors (page 31), explains how you can implement your own custom editors.

For each property, the **C1PropertyGrid** control checks the list of available editors and selects the first one that supports the current property type. This allows you to specify your own editor for all properties of a given type. Simply add your editor to the start of the **AvailableEditors** list and it will be used for all suitable properties. For example:

- Visual Basic

```
Dim pg = New C1PropertyGrid()
pg.AvailableEditors.Insert(0, New DateTimeEditor())
```

- C#

```
var pg = new C1PropertyGrid();
pg.AvailableEditors.Insert(0, new DateTimeEditor());
```

If you want to use a custom editor only for specific properties and not for all properties of a certain type you can either add an "Editor" attribute to the object being edited or use the **PropertyAttributes** property described above.

Creating Custom Editors

If the built-in editors do not fit your needs, you can easily create your own editors and use them with the **C1PropertyGrid**.

The following simple steps are required:

1. Create a class that implements the **ITypeEditorControl** interface.
2. Add an instance of this class to the **AvailableEditors** collection on the **C1PropertyGrid** control.

   OR

   Specify this class as the editor for a specific property by adding an **EditorAttribute** to the property definition.

The **ITypeEditorControl** interface contains the following members:

- **bool Supports(PropertyAttribute Property)**
  This method is used by the **C1PropertyGrid** to determine whether the editor supports the type of a given property.
- **void Attach(PropertyAttribute property)**
  This method is called when initializing the editor with the property it will manage. This typically consists of initializing the editor content based on the current property value.
- **void Detach(PropertyAttribute property)**
  This method is called when the editor instance is being released.
- **ITypeEditorControl Create()**
  This method is called when the **C1PropertyGrid** needs a new instance of the editor.
- **event PropertyChangedEventHandler ValueChanged**
  This event fires when the value of the property changes. This is used by editors that perform validation.

For a complete implementation of a custom editor, please refer to the **ControlExplorer** samples installed with **ComponentOne Studio for WPF**.