

---

ComponentOne

# PropertyGrid for Silverlight

Copyright © 1987-2009 ComponentOne LLC. All rights reserved.

*Corporate Headquarters*  
**ComponentOne LLC**  
201 South Highland Avenue  
3<sup>rd</sup> Floor  
Pittsburgh, PA 15206 • USA

**Internet:** [info@ComponentOne.com](mailto:info@ComponentOne.com)  
**Web site:** <http://www.componentone.com>

**Sales**

E-mail: [sales@componentone.com](mailto:sales@componentone.com)  
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

**Trademarks**

**ComponentOne PropertyGrid for Silverlight** and the **ComponentOne PropertyGrid for Silverlight** logo are trademarks, and ComponentOne is a registered trademark of ComponentOne LLC. All other trademarks used herein are the properties of their respective owners.

**Warranty**

ComponentOne warrants that the original CD (or diskettes) are free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective CD (or disk) to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for a defective CD (or disk) by sending it and a check for \$25 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original CD (or disks) set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. We are not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

**Copying and Distribution**

While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only.

---

# Table of Contents

<b>ComponentOne PropertyGrid for Silverlight.....</b>	<b>1</b>
<b>Introduction to the C1PropertyGrid Control.....</b>	<b>1</b>
<b>Customizing the C1PropertyGrid .....</b>	<b>3</b>
Customizing the Control Layout .....	3
Customizing Display Names .....	4
Categorizing Properties .....	4
Displaying Methods and Properties .....	6
Customizing the Editors.....	7
Creating Custom Editors.....	7



# ComponentOne PropertyGrid for Silverlight

**ComponentOne PropertyGrid™ for Silverlight** is a Silverlight version of the popular **PropertyGrid** control that ships as part of the .NET WinForms platform. Using **ComponentOne PropertyGrid™ for Silverlight**, users can browse and edit properties on any .NET object.

The **C1PropertyGrid** control is part of the **C1.Silverlight.Extended** assembly.

## Introduction to the C1PropertyGrid Control

Like the original **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.

For example, assuming you have a simple **Customer** class defined as follows:

```
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:

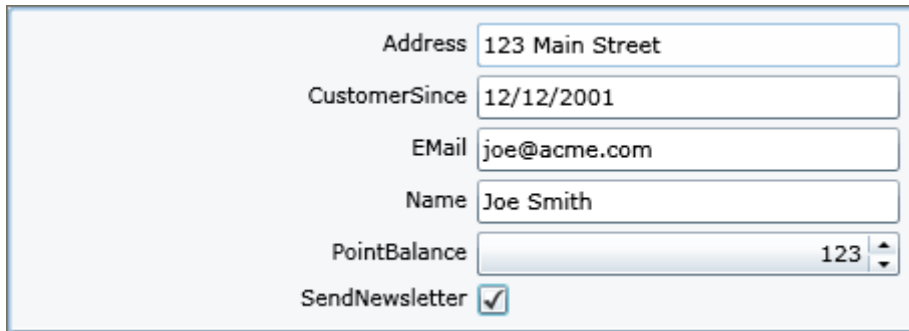
```
public Page()
{
    InitializeComponent();

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

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

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

And the resulting application would look like this:



The screenshot shows a C1PropertyGrid control with the following properties and values:

Property Name	Value
Address	123 Main Street
CustomerSince	12/12/2001
EMail	joe@acme.com
Name	Joe Smith
PointBalance	123
SendNewsletter	<input checked="" type="checkbox"/>

This simple UI allows users to edit all the properties in our **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.

Note that the **C1PropertyGrid** only displays properties of value type and strings. It does not display properties that contain objects with other properties.

Notice that properties are shown in alphabetical order by default. You can change this by setting the **SortProperties** property to **False**.

# Customizing the C1PropertyGrid

Although it can be used by setting a single property, the **C1PropertyGrid** provides extensive customization support.

## 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:

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

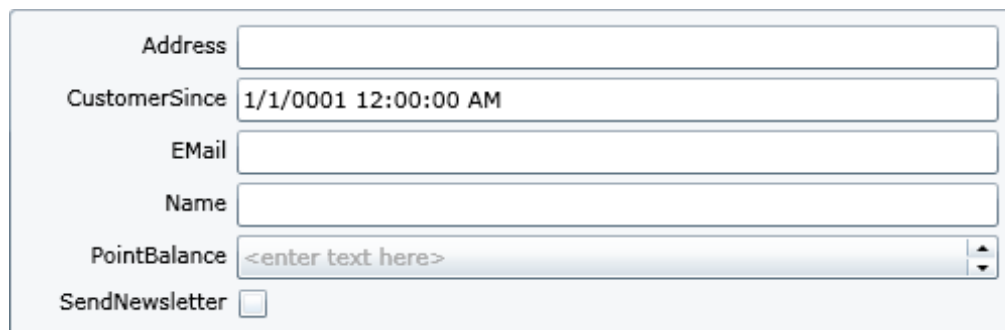
    // Create object to browse
    var customer = new Customer();

    // 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	<input type="text"/>
CustomerSince	1/1/0001 12:00:00 AM
EMail	<input type="text"/>
Name	<input type="text"/>
PointBalance	<enter text here>
SendNewsletter	<input type="checkbox"/>

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.

For example, consider this updated version of our **Customer** class:

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

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

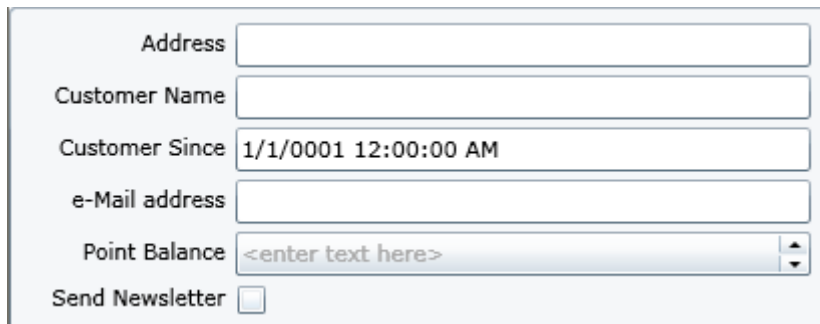
    public string Address { get; set; }

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

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

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

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



The screenshot shows a window titled 'C1PropertyGrid' containing several input fields and a checkbox. The fields are labeled as follows: 'Address' (text box), 'Customer Name' (text box), 'Customer Since' (text box containing '1/1/0001 12:00:00 AM'), 'e-Mail address' (text box), 'Point Balance' (spin box containing '<enter text here>'), and 'Send Newsletter' (checkbox).

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.

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

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

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

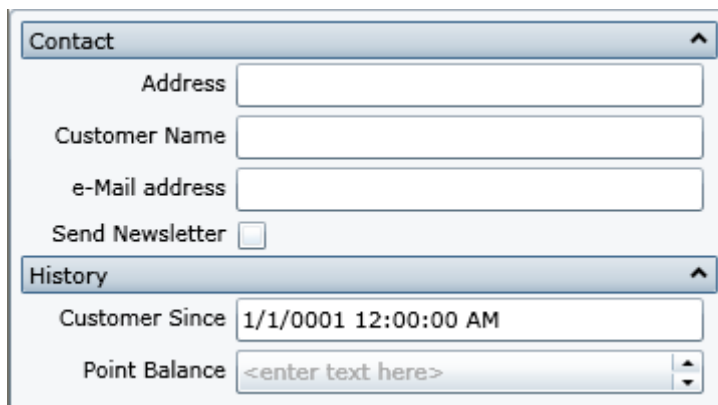
    [Category("Contact")]
    public string Address { get; set; }

    [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:



The screenshot shows a property grid with two expandable categories: "Contact" and "History".

- Contact Category:**
  - Address: Text input field
  - Customer Name: Text input field
  - e-Mail address: Text input field
  - Send Newsletter: Check box (unchecked)
- History Category:**
  - Customer Since: Text input field containing "1/1/0001 12:00:00 AM"
  - Point Balance: Text input field containing "<enter text here>" with a spinner control on the right.

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 may prevent the **C1PropertyGrid** from grouping properties by setting the **CategorizedView** property to **False**.

## 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.
AutoGenerateProperties	If the <b>PropertyAttributes</b> collection is empty and this property is set to <b>True</b> (the default), then the <b>C1PropertyGrid</b> automatically shows all public properties of the <b>SelectedObject</b> .
AutoGenerateMethods	If the <b>PropertyAttributes</b> collection is empty and this property is set to <b>True</b> , then the <b>C1PropertyGrid</b> automatically shows all public methods of the <b>SelectedObject</b> 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:

```
// 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 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:

```
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
3. 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 Silverlight**.