ComponentOne

ComboBox for WPF

Corporate Headquarters
ComponentOne LLC
201 South Highland Avenue
3rd 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)

Trademarks

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.

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

Table of Contents

ComponentOne ComboBox for WPF Overview

ComponentOne ComboBox™ for WPF is a full-featured combo box control that combines an editable text box with an auto-searchable drop-down list.

Getting Started

- Working with the C1ComboBox Control (page 23)

- Quick Start (page 17)

- Task-Based Help (page 31)

What's New in ComboBox for WPF

The **ComboBox for WPF** documentation was last updated on October 29, 2010 for its 2010 v3 release.

The following features have been added to **ComboBox for WPF**:

- **ComboBox for WPF** now features ComponentOne's ClearStyle technology, a new and simple approach to providing Silverlight and WPF control styling. For more information about ClearStyle technology, see the ComponentOne ClearStyle Technology (page 25) topic.
- You can now use a unified namespace, `xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"`, in your WPF projects. All ComponentOne WPF controls will be accessible through XAML using this namespace. Please note, however, that you must add a reference to the assembly of each control you wish to use.

Tip: A version history containing a list of new features, improvements, fixes, and changes for each product is available in HelpCentral at http://helpcentral.componentone.com/VersionHistory.aspx.

Installing ComboBox for WPF

The following sections provide helpful information on installing **ComponentOne ComboBox for WPF**.

ComboBox for WPF Setup Files

The installation program will create the directory **C:\Program Files\ComponentOne\Studio for WPF**, which contains the following subdirectories:

Bin	Contains copies of all ComponentOne binaries (DLLs, EXEs). For **Component ComboBox for WPF**, the following DLLs are installed: • C1.WPF.dll In addition, the following files from the Microsoft WPF Toolkit are also installed: • WPFToolkit.dll

- WPFToolkit.Design.dll
- WPFToolkit.VisualStudio.Design.dll

	For more information about the Microsoft WPF Toolkit, see CodePlex. The C1.WPF.dll and WPFToolkit.dll assemblies are required for deployment.
C1WPF\XAML	Contains the full XAML definitions of C1ComboBox styles and templates which can be used for creating your own custom styles and templates.

The **ComponentOne Studio for WPF Help Setup** program installs integrated Microsoft Help 2.0 and Microsoft Help Viewer help to the C:\Program Files\ComponentOne\Studio for WPF directory in the following folders:

H2Help	Contains Microsoft Help 2.0 integrated documentation for all Studio components.
HelpViewer	Contains Microsoft Help Viewer Visual Studio 2010 integrated documentation for all Studio components.

Samples

Samples for the product are installed in the **ComponentOne Samples** folder by default. The path of the **ComponentOne Samples** directory is slightly different on Windows XP and Windows 7/Vista machines:

Windows XP path: C:\Documents and Settings\<username>\My Documents\ComponentOne Samples

Windows 7/Vista path: C:\Users\<username>\Documents\ComponentOne Samples

The **ComponentOne Samples** folder contains the following subdirectories:

Common	Contains support and data files that are used by many of the demo programs.
Studio for WPF	Contains samples for **DateTimeEditors for WPF**.

You can access samples from the **ComponentOne Control Explorer**. To view samples, on your desktop, click the **Start** button and then click **ComponentOne | Studio for WPF | Samples | WPF ControlExplorer**.

System Requirements

System requirements include the following:

Operating Systems:	Microsoft Windows® XP with Service Pack 2 (SP2) Windows Vista™ Windows 2008 Server Windows 7
Environments:	.NET Framework 3.5 or later Visual Studio® 2005 extensions for .NET Framework 2.0 November 2006 CTP Visual Studio® 2008
Microsoft® Expression®	**ComboBox for WPF** includes design-time support for Expression Blend.

Blend Compatibility:

Note: The **C1.WPF.VisualStudio.Design.dll** assembly is required by Visual Studio 2008 and the **C1.WPF.Expression.Design.dll** assembly is required by Expression Blend. The **C1.WPF.Expression.Design.dll** and **C1.WPF.VisualStudio.Design.dll** assemblies installed with **ComboBox for WPF** should always be placed in the same folder as **C1.WPF.dll**; the DLLs should NOT be placed in the Global Assembly Cache (GAC).

Installing Demonstration Versions

If you wish to try **ComponentOne ComboBox for WPF** and do not have a serial number, follow the steps through the installation wizard and use the default serial number.

The only difference between unregistered (demonstration) and registered (purchased) versions of our products is that registered versions will stamp every application you compile so that a ComponentOne banner will not appear when your users run the applications.

Uninstalling ComboBox for WPF

To uninstall **ComponentOne Studio for WPF**:

1. Open the **Control Panel** and select **Add or Remove Programs** (**Programs and Features** in Vista/7).
2. Select **ComponentOne Studio for WPF** and click the **Remove** button.
3. Click **Yes** to remove the program.

To uninstall **ComponentOne Studio for WPF** integrated help:

1. Open the **Control Panel** and select **Add or Remove Programs** (**Programs and Features** in Windows 7/Vista).
2. Select **ComponentOne Studio for WPF Help** and click the **Remove** button.
3. Click **Yes** to remove the integrated help.

End-User License Agreement

All of the ComponentOne licensing information, including the ComponentOne end-user license agreements, frequently asked licensing questions, and the ComponentOne licensing model, is available online at http://www.componentone.com/SuperPages/Licensing/.

Licensing FAQs

This section describes the main technical aspects of licensing. It may help the user to understand and resolve licensing problems he may experience when using ComponentOne .NET and ASP.NET products.

What is Licensing?

Licensing is a mechanism used to protect intellectual property by ensuring that users are authorized to use software products.

Licensing is not only used to prevent illegal distribution of software products. Many software vendors, including ComponentOne, use licensing to allow potential users to test products before they decide to purchase them.

Without licensing, this type of distribution would not be practical for the vendor or convenient for the user. Vendors would either have to distribute evaluation software with limited functionality, or shift the burden of managing software licenses to customers, who could easily forget that the software being used is an evaluation version and has not been purchased.

How does Licensing Work?

ComponentOne uses a licensing model based on the standard set by Microsoft, which works with all types of components.

> **Note:** The **Compact Framework** components use a slightly different mechanism for run-time licensing than the other ComponentOne components due to platform differences.

When a user decides to purchase a product, he receives an installation program and a Serial Number. During the installation process, the user is prompted for the serial number that is saved on the system. (Users can also enter the serial number by clicking the **License** button on the **About Box** of any ComponentOne product, if available, or by rerunning the installation and entering the serial number in the licensing dialog box.)

When a licensed component is added to a form or Web page, Visual Studio obtains version and licensing information from the newly created component. When queried by Visual Studio, the component looks for licensing information stored in the system and generates a run-time license and version information, which Visual Studio saves in the following two files:

- An assembly resource file which contains the actual run-time license.
- A "licenses.licx" file that contains the licensed component strong name and version information.

These files are automatically added to the project.

In WinForms and ASP.NET 1.x applications, the run-time license is stored as an embedded resource in the assembly hosting the component or control by Visual Studio. In ASP.NET 2.x applications, the run-time license may also be stored as an embedded resource in the **App_Licenses.dll** assembly, which is used to store all run-time licenses for all components directly hosted by WebForms in the application. Thus, the **App_licenses.dll** must always be deployed with the application.

The **licenses.licx** file is a simple text file that contains strong names and version information for each of the licensed components used in the application. Whenever Visual Studio is called upon to rebuild the application resources, this file is read and used as a list of components to query for run-time licenses to be embedded in the appropriate assembly resource. Note that editing or adding an appropriate line to this file can force Visual Studio to add run-time licenses of other controls as well.

Note that the **licenses.licx** file is usually not shown in the Solution Explorer; it appears if you press the **Show All Files** button in the Solution Explorer's Toolbox or, from Visual Studio's main menu, select **Show All Files** on the **Project** menu.

Later, when the component is created at run time, it obtains the run-time license from the appropriate assembly resource that was created at design time and can decide whether to simply accept the run-time license, to throw an exception and fail altogether, or to display some information reminding the user that the software has not been licensed.

All ComponentOne products are designed to display licensing information if the product is not licensed. None will throw licensing exceptions and prevent applications from running.

Common Scenarios

The following topics describe some of the licensing scenarios you may encounter.

Creating components at design time

This is the most common scenario and also the simplest: the user adds one or more controls to the form, the licensing information is stored in the **licenses.licx** file, and the component works.

Note that the mechanism is exactly the same for Windows Forms and Web Forms (ASP.NET) projects.

Creating components at run time

This is also a fairly common scenario. You do not need an instance of the component on the form, but would like to create one or more instances at run time.

In this case, the project will not contain a **licenses.licx** file (or the file will not contain an appropriate run-time license for the component) and therefore licensing will fail.

To fix this problem, add an instance of the component to a form in the project. This will create the **licenses.licx** file and things will then work as expected. (The component can be removed from the form after the **licenses.licx** file has been created).

Adding an instance of the component to a form, then removing that component, is just a simple way of adding a line with the component strong name to the **licenses.licx** file. If desired, you can do this manually using notepad or Visual Studio itself by opening the file and adding the text. When Visual Studio recreates the application resources, the component will be queried and its run-time license added to the appropriate assembly resource.

Inheriting from licensed components

If a component that inherits from a licensed component is created, the licensing information to be stored in the form is still needed. This can be done in two ways:

- Add a **LicenseProvider** attribute to the component.

  This will mark the derived component class as licensed. When the component is added to a form, Visual Studio will create and manage the **licenses.licx** file and the base class will handle the licensing process as usual. No additional work is needed. For example:

  ```
  [LicenseProvider(typeof(LicenseProvider))]
  class MyGrid: C1.Win.C1FlexGrid.C1FlexGrid
  {
    // ...
  }
  ```

- Add an instance of the base component to the form.

  This will embed the licensing information into the **licenses.licx** file as in the previous scenario and the base component will find it and use it. As before, the extra instance can be deleted after the **licenses.licx** file has been created.

Please note that ComponentOne licensing will not accept a run-time license for a derived control if the run-time license is embedded in the same assembly as the derived class definition and the assembly is a DLL. This restriction is necessary to prevent a derived control class assembly from being used in other applications without a design-time license. If you create such an assembly, you will need to take one of the actions previously described create a component at run time.

Using licensed components in console applications

When building console applications, there are no forms to add components to and therefore Visual Studio won't create a **licenses.licx** file.

In these cases, create a temporary Windows Forms application and add all the desired licensed components to a form. Then close the Windows Forms application and copy the **licenses.licx** file into the console application project.

Make sure the **licenses.licx** file is configured as an embedded resource. To do this, right-click the **licenses.licx** file in the Solution Explorer window and select **Properties**. In the Properties window, set the **Build Action** property to **Embedded Resource**.

Using licensed components in Visual C++ applications

There is an issue in VC++ 2003 where the **licenses.licx** is ignored during the build process; therefore, the licensing information is not included in VC++ applications.

To fix this problem, extra steps must be taken to compile the licensing resources and link them to the project. Note the following:

1. Build the C++ project as usual. This should create an EXE file and also a licenses.licx file with licensing information in it.
2. Copy the **licenses.licx** file from the application directory to the target folder (**Debug** or **Release**).
3. Copy the **C1Lc.exe** utility and the licensed DLLs to the target folder. (Don't use the standard lc.exe, it has bugs.)
4. Use **C1Lc.exe** to compile the **licenses.licx** file. The command line should look like this:
   ```
   c1lc /target:MyApp.exe /complist:licenses.licx /i:C1.Win.C1FlexGrid.dll
   ```
5. Link the licenses into the project. To do this, go back to Visual Studio, right-click the project, select **Properties**, and go to the **Linker/Command Line** option. Enter the following:
   ```
   /ASSEMBLYRESOURCE:Debug\MyApp.exe.licenses
   ```
6. Rebuild the executable to include the licensing information in the application.

Using licensed components with automated testing products

Automated testing products that load assemblies dynamically may cause them to display license dialog boxes. This is the expected behavior since the test application typically does not contain the necessary licensing information and there is no easy way to add it.

This can be avoided by adding the string "C1CheckForDesignLicenseAtRuntime" to the **AssemblyConfiguration** attribute of the assembly that contains or derives from ComponentOne controls. This attribute value directs the ComponentOne controls to use design-time licenses at run time.

For example:

```
#if AUTOMATED_TESTING
   [AssemblyConfiguration("C1CheckForDesignLicenseAtRuntime")]
#endif
   public class MyDerivedControl : C1LicensedControl
   {
         // ...
   }
```

Note that the **AssemblyConfiguration** string may contain additional text before or after the given string, so the **AssemblyConfiguration** attribute can be used for other purposes as well. For example:

```
[AssemblyConfiguration("C1CheckForDesignLicenseAtRuntime,BetaVersion")]
```

THIS METHOD SHOULD ONLY BE USED UNDER THE SCENARIO DESCRIBED. It requires a design-time license to be installed on the testing machine. Distributing or installing the license on other computers is a violation of the EULA.

Troubleshooting

We try very hard to make the licensing mechanism as unobtrusive as possible, but problems may occur for a number of reasons.

Below is a description of the most common problems and their solutions.

I have a licensed version of a ComponentOne product but I still get the splash screen when I run my project.

If this happens, there may be a problem with the **licenses.licx** file in the project. It either doesn't exist, contains wrong information, or is not configured correctly.

First, try a full rebuild (**Rebuild All** from the Visual Studio **Build** menu). This will usually rebuild the correct licensing resources.

If that fails follow these steps:

1. Open the project and go to the Solution Explorer window.
2. Click the **Show All Files** button on the top of the window.
3. Find the **licenses.licx** file and open it. If prompted, continue to open the file.
4. Change the version number of each component to the appropriate value. If the component does not appear in the file, obtain the appropriate data from another **licenses.licx** file or follow the alternate procedure following.
5. Save the file, then close the **licenses.licx** tab.
6. Rebuild the project using the **Rebuild All** option (not just **Rebuild**).

Alternatively, follow these steps:

1. Open the project and go to the Solution Explorer window.
2. Click the **Show All Files** button on the top of the window.
3. Find the **licenses.licx** file and delete it.
4. Close the project and reopen it.
5. Open the main form and add an instance of each licensed control.
6. Check the Solution Explorer window, there should be a **licenses.licx** file there.
7. Rebuild the project using the **Rebuild All** option (not just **Rebuild**).

For ASP.NET 2.x applications, follow these steps:

1. Open the project and go to the Solution Explorer window.
2. Find the **licenses.licx** file and right-click it.
3. Select the **Rebuild Licenses** option (this will rebuild the **App_Licenses.licx** file).
4. Rebuild the project using the **Rebuild All** option (not just **Rebuild**).

I have a licensed version of a ComponentOne product on my Web server but the components still behave as unlicensed.

There is no need to install any licenses on machines used as servers and not used for development.

The components must be licensed on the development machine, therefore the licensing information will be saved into the executable (.exe or .dll) when the project is built. After that, the application can be deployed on any machine, including Web servers.

For ASP.NET 2.x applications, be sure that the App_Licenses.dll assembly created during development of the application is deployed to the bin application bin directory on the Web server.

If your ASP.NET application uses WinForms user controls with constituent licensed controls, the runtime license is embedded in the WinForms user control assembly. In this case, you must be sure to rebuild and update the user control whenever the licensed embedded controls are updated.

I downloaded a new build of a component that I have purchased, and now I'm getting the splash screen when I build my projects.

Make sure that the serial number is still valid. If you licensed the component over a year ago, your subscription may have expired. In this case, you have two options:

Option 1 – Renew your subscription to get a new serial number.

If you choose this option, you will receive a new serial number that you can use to license the new components (from the installation utility or directly from the **About Box**).

The new subscription will entitle you to a full year of upgrades and to download the latest maintenance builds directly from http://prerelease.componentone.com/.

Option 2 – Continue to use the components you have.

Subscriptions expire, products do not. You can continue to use the components you received or downloaded while your subscription was valid.

Technical Support

ComponentOne offers various support options. For a complete list and a description of each, visit the ComponentOne Web site at http://www.componentone.com/SuperProducts/SupportServices/.

Some methods for obtaining technical support include:

- **Online Support via HelpCentral**
  ComponentOne HelpCentral provides customers with a comprehensive set of technical resources in the form of FAQs, samples, Version Release History, Articles, searchable Knowledge Base, searchable Online Help and more. We recommend this as the first place to look for answers to your technical questions.

- **Online Support via our Incident Submission Form**
  This online support service provides you with direct access to our Technical Support staff via an online incident submission form. When you submit an incident, you'll immediately receive a response via e-mail confirming that you've successfully created an incident. This email will provide you with an Issue Reference ID and will provide you with a set of possible answers to your question from our Knowledgebase. You will receive a response from one of the ComponentOne staff members via e-mail in 2 business days or less.

- **Peer-to-Peer Product Forums and Newsgroups**
  ComponentOne peer-to-peer product forums and newsgroups are available to exchange information, tips, and techniques regarding ComponentOne products. ComponentOne sponsors these areas as a forum for users to share information. While ComponentOne does not provide direct support in the forums and newsgroups, we periodically monitor them to ensure accuracy of information and provide comments when appropriate. Please note that a ComponentOne User Account is required to participate in the ComponentOne Product Forums.

- **Installation Issues**
  Registered users can obtain help with problems installing ComponentOne products. Contact technical support by using the online incident submission form or by phone (412.681.4738). Please note that this does not include issues related to distributing a product to end-users in an application.

- **Documentation**
  Microsoft integrated ComponentOne documentation can be installed with each of our products, and documentation is also available online. If you have suggestions on how we can improve our documentation, please email the Documentation team. Please note that e-mail sent to the Documentation team is for documentation feedback only. Technical Support and Sales issues should be sent directly to their respective departments.

Note: You must create a ComponentOne Account and register your product with a valid serial number to obtain support using some of the above methods.

Redistributable Files

ComponentOne ComboBox for WPF is developed and published by ComponentOne LLC. You may use it to develop applications in conjunction with Microsoft Visual Studio or any other programming environment that enables the user to use and integrate the control(s). You may also distribute, free of royalties, the following Redistributable Files with any such application you develop to the extent that they are used separately on a single CPU on the client/workstation side of the network:

- C1.WPF.dll

In addition, the following file from the Microsoft WPF Toolkit is also installed and is redistributable:

- WPFToolkit.dll

Site licenses are available for groups of multiple developers. Please contact Sales@ComponentOne.com for details.

About this Documentation

You can create your applications using Microsoft Expression Blend or Visual Studio, but Blend is currently the only design-time environment that allows users to design XAML documents visually. In this documentation, we will use the **Design** workspace of Blend for most examples.

Acknowledgements

Microsoft, Windows, Windows Vista, Visual Studio, and Microsoft Expression are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

ComponentOne

If you have any suggestions or ideas for new features or controls, please call us or write:

Corporate Headquarters

ComponentOne LLC

201 South Highland Avenue

3rd Floor

Pittsburgh, PA 15206 • USA

412.681.4343

412.681.4384 (Fax)

http://www.componentone.com/

ComponentOne Doc-To-Help

This documentation was produced using ComponentOne Doc-To-Help® Enterprise.

XAML and XAML Namespaces

XAML is a declarative XML-based language that is used as a user interface markup language in Windows Presentation Foundation (WPF) and the .NET Framework 3.0. With XAML you can create a graphically rich customized user interface, perform data binding, and much more. For more information on XAML and the .NET Framework 3.0, please see http://www.microsoft.com.

XAML Namespaces

Namespaces organize the objects defined in an assembly. Assemblies can contain multiple namespaces, which can in turn contain other namespaces. Namespaces prevent ambiguity and simplify references when using large groups of objects such as class libraries.

When you create a Microsoft Expression Blend project, a XAML file is created for you and some initial namespaces are specified:

Namespace	**Description**
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"	This is the default Windows Presentation Foundation namespace.
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"	This is a XAML namespace that is mapped to the **x:** prefix. The **x:** prefix provides a quick, easy way to reference the namespace, which defines many commonly-used features necessary for WPF applications.

When you add a C1ComboBox control to the window in Microsoft Expression Blend or Visual Studio, **Blend** or **Visual Studio** automatically creates an XML namespace for the control. The namespace looks like the following:

```
xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
```

The namespace value is **c1**. This is a unified namespace; once this is in the project, all ComponentOne WPF controls found in your references will be accessible through XAMl (and Intellisense). Note that you still need to add references to the assemblies for each control you need to use.

The namespace value is **my** and the namespace is **C1.WPF.Extended**.

You can also choose to create your own custom name for the namespace. For example:

```
xmlns:MyMTB="clr-namespace:C1.WPF;assembly=C1.WPF
```

You can now use your custom namespace when assigning properties, methods, and events. For example, use the following XAML to add a border around the panel:

```
<MyMTB:C1ComboBox Name="c1ComboBox1" BorderThickness="10,10,10,10">
```

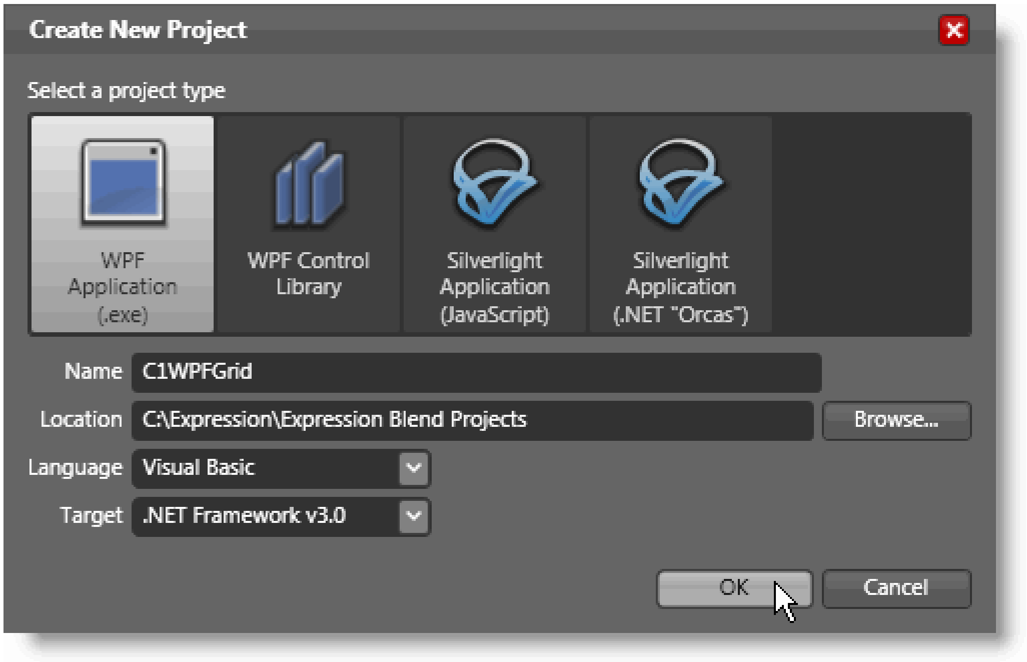
Creating a Microsoft Blend Project

To create a new Blend project, complete the following steps:

1. From the **File** menu, select **New Project** or click **New Project** in the Blend startup window.

   The **Create New Project** dialog box opens.

2. Make sure **WPF Application (.exe)** is selected and enter a name for the project in the Name text box. The **WPF Application (.exe)** creates a project for a Windows-based application that can be built and run while being designed.
3. Select the **Browse** button to specify a location for the project.
4. Select a language from the **Language** drop-down box and click **OK**.



Name C1WPFGrid
Location C:\Expression\Expression Blend Projects
Language Visual Basic
Target .NET Framework v3.0

A new Blend project with a XAML window is created.

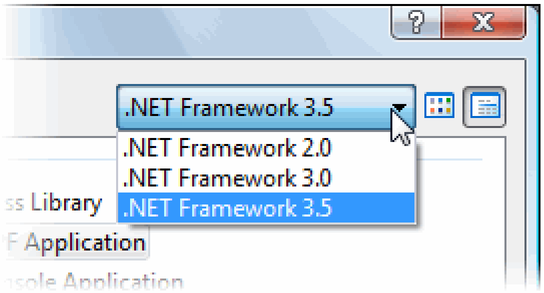
Creating a .NET Project in Visual Studio

To create a new .NET project in Visual Studio 2008, complete the following steps:

1. From the **File** menu in Microsoft Visual Studio 2008, select **New Project**.

   The **New Project** dialog box opens.

2. Choose the appropriate .NET Framework from the Framework drop-down box in the top-right of the dialog box.

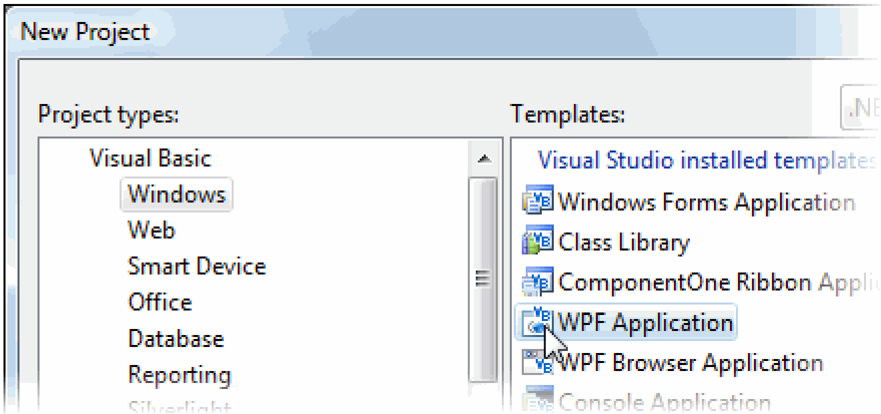


.NET Framework 3.5

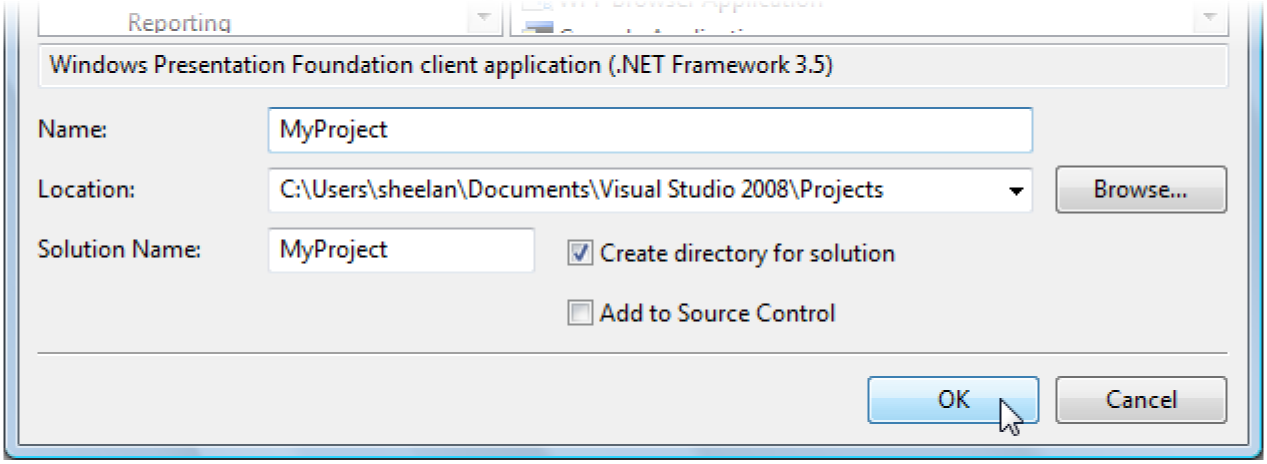
3. Under **Project types**, select either **Visual Basic** or **Visual C#**.

> **Note:** In Visual Studio 2005 select **NET Framework 3.0** under **Visual Basic** or **Visual C#** in the Project types menu.

4. Choose **WPF Application** from the list of **Templates** in the right pane.



5. Enter a name for your application in the **Name** field and click **OK**.



Windows Presentation Foundation client application (.NET Framework 3.5)

Name: MyProject

Location: C:\Users\sheelan\Documents\Visual Studio 2008\Projects

Solution Name: MyProject

☑ Create directory for solution

☐ Add to Source Control

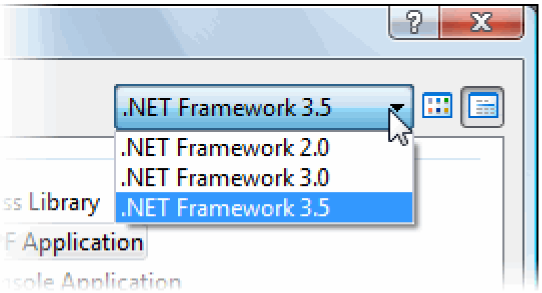
A new Microsoft Visual Studio .NET WPF project is created with a XAML file that will be used to define your user interface and commands in the application.

Note: You can create your WPF applications using Microsoft Expression Blend or Visual Studio, but Blend is currently the only design-time environment that allows users to design XAML documents visually. In this documentation, Blend will be used for most examples.

Creating an XAML Browser Application (XBAP) in Visual Studio

To create a new XAML Browser Application (XBAP) in Visual Studio 2008, complete the following steps:

1. From the **File** menu in Microsoft Visual Studio 2008, select **New Project**. The **New Project** dialog box opens.
2. Choose the appropriate .NET Framework from the Framework drop-down box in the top-right of the dialog box.



3. Under Project types, select either **Visual Basic** or **Visual C#**.
4. Choose **WPF Browser Application** from the list of **Templates** in the right pane.

> **Note:** If using Visual Studio 2005, you may need to select **XAML Browser Application (WPF)** after selecting **NET Framework 3.0** under **Visual Basic** or **Visual C#** in the left-side menu.

5. Enter a name for your application in the **Name** field and click **OK**.

   A new Microsoft Visual Studio .NET WPF Browser Application project is created with a XAML file that will be used to define your user interface and commands in the application.

Adding the ComboBox for WPF Components to a Blend Project

In order to use C1ComboBox or another **ComponentOne ComboBox for WPF** component in the Design workspace of Blend, you must first add a reference to the **C1.WPF.Extended** assembly and then add the component from Blend's **Asset Library**.

To add a reference to the assembly:

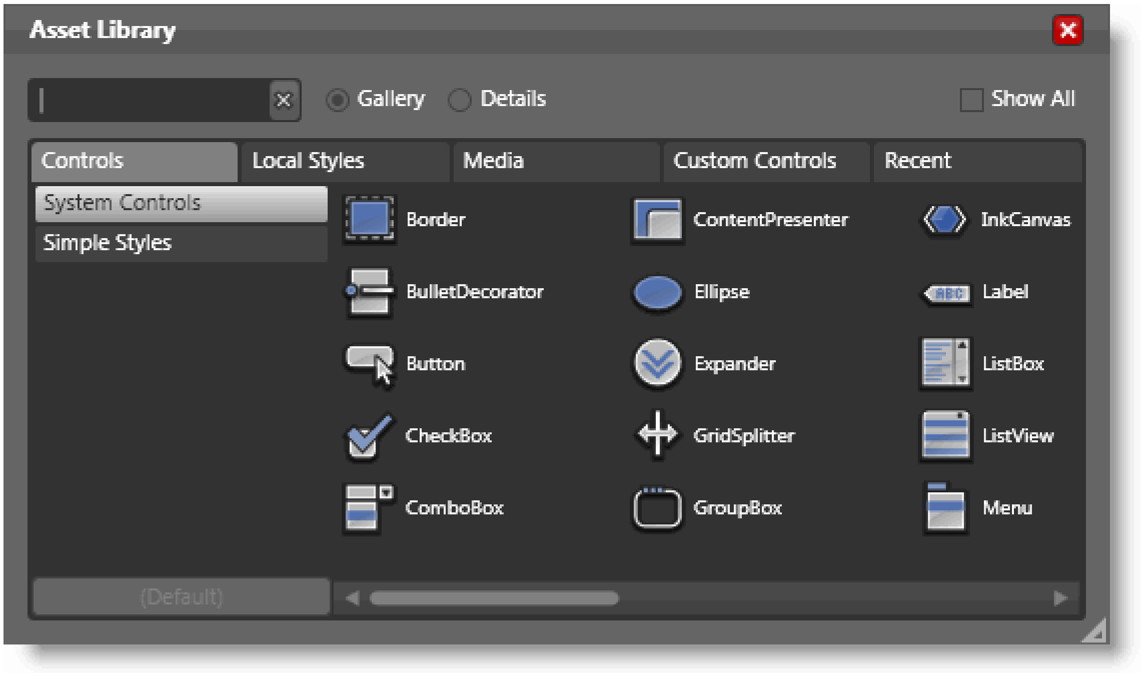
1. Select **Project | Add Reference**.
2. Browse to find the **C1.WPF.dll** assembly installed with **ComboBox for WPF**.

> **Note:** The **C1.WPF.dll** file is installed to **C:\Program Files\ComponentOne\Studio for WPF\bin** by default.

3. Select **C1.WPF.dll** and click **Open**. A reference is added to your project.

To add a component from the Asset Library:

1. Once you have added a reference to the **C1.WPF.Extended** assembly, click the **Asset Library** button » in the Blend Toolbox. The **Asset Library** appears:



2. Click the **Custom Controls** tab. All of the **ComboBox for WPF** main and auxiliary components are listed here.
3. Select **C1ComboBox**. The component will appear in the Toolbox above the **Asset Library** button.
4. Double-click the **C1ComboBox** component in the Toolbox to add it to **Window1.xaml**.

Adding the ComboBox for WPF Components to a Visual Studio Project

When you install **ComponentOne ComboBox for WPF** the C1ComboBox control should be added to your Visual Studio Toolbox. You can also manually add ComponentOne controls to the Toolbox.

ComponentOne ComboBox for WPF provides the following control:

- C1ComboBox

To use a **ComboBox for WPF** panel or control, add it to the window or add a reference to the **C1.WPF** assembly to your project.

Manually Adding ComboBox for WPF to the Toolbox

When you install **ComboBox for WPF**, the following **ComboBox for WPF** control and panel will appear in the Visual Studio Toolbox customization dialog box:

- C1ComboBox

To manually add the C1ComboBox control to the Visual Studio Toolbox, complete the following steps:

1. Open the Visual Studio IDE (Microsoft Development Environment). Make sure the Toolbox is visible (select **Toolbox** in the **View** menu, if necessary) and right-click the Toolbox to open its context menu.
2. To make **ComboBox for WPF** components appear on its own tab in the Toolbox, select **Add Tab** from the context menu and type in the tab name, **C1WPFComboBox**, for example.
3. Right-click the tab where the component is to appear and select **Choose Items** from the context menu.

   The **Choose Toolbox Items** dialog box opens.
4. In the dialog box, select the **WPF Components** tab.

5. Sort the list by Namespace (click the *Namespace* column header) and select the check boxes for components belonging to the **C1.WPF.Extended** namespace. Note that there may be more than one component for each namespace.

Adding ComboBox for WPF to the Window

To add **ComponentOne ComboBox for WPF** to a window or page, complete the following steps:

1. Add the C1ComboBox control to the Visual Studio Toolbox.
2. Double-click C1ComboBox or drag the control onto the window.

Adding a Reference to the Assembly

To add a reference to the **ComboBox for WPF** assembly, complete the following steps:

1. Select the **Add Reference** option from the **Project** menu of your project.
2. Select the **ComponentOne ComboBox for WPF** assembly from the list on the **.NET** tab or on the **Browse** tab, browse to find the **C1.WPF.dll** assembly and click **OK**.
3. Double-click the window caption area to open the code window. At the top of the file, add the following **Imports** statements (**using** in C#):

```
Imports C1.WPF
```

This makes the objects defined in the **ComboBox for WPF** assembly visible to the project.

Key Features

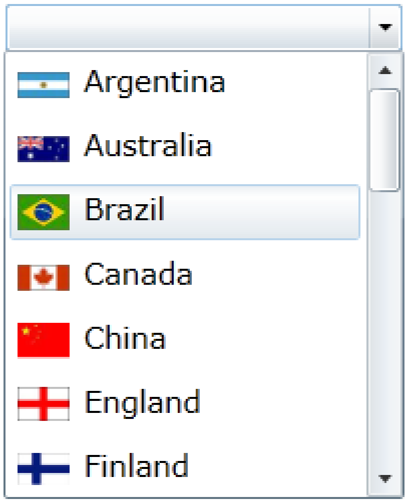
ComponentOne ComboBox for WPF allows you to create customized, rich applications. Make the most of **ComboBox for WPF** by taking advantage of the following key features:

- **Auto-Searchable Drop-Down List**

  Locate items quickly by typing the first few characters. ComboBox will automatically search the list and select the items for you as you type.

- **Populate the Drop-down List with Data Templates**

  ComboBox fully supports data templates, making it easy to add any visual elements to the list items. This includes text, images, and any other controls. The control uses element virtualization, so it always loads quickly, even when populated with hundreds of items.



- **Time-tested, Familiar Object Model**

  ComboBox has a rich object model based on the WPF ComboBox control. You can easily specify whether the end user is able to enter items that are not on the drop-down list, get or set the index of the selected item, the height of the drop-down list, and more.

ComboBox for WPF Quick Start

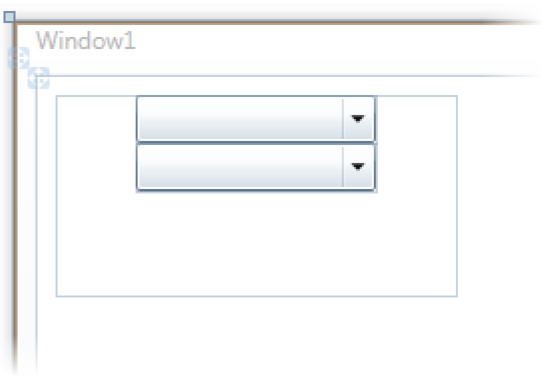
The following quick start guide is intended to get you up and running with **ComboBox for WPF**. In this quick start, you'll start in Visual Studio 2008 to create a new project with two C1ComboBox controls. The first control will be populated with a list of three items that, when clicked, will determine the list that appears in the second combo box.

Step 1 of 4: Creating an Application with a C1ComboBox Control

In this step, you'll begin in Visual Studio 2008 to create a WPF application using **ComboBox for WPF**.

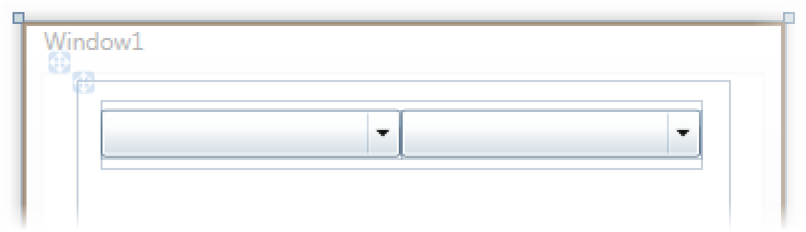
Complete the following steps:

1. In Visual Studio 2008, select **File | New | Project**.
2. In the **New Project** dialog box, select a language in the left pane, and in the templates list select **WPF Application**.
3. Enter a **Name** for your project and click **OK**.
4. Add two C1ComboBox controls to the project by completing the following steps:
   a. In the **Toolbox**, double-click the **StackPanel** icon to add it to the project.
   b. Select the **StackPanel** control.
   c. Double-click the C1ComboBox icon to add the control to the **StackPanel**.
   d. Repeat steps 4b and 4c to add another C1ComboBox to the **StackPanel**. The project resembles the following:



5. Set the **StackPanel** control's properties as follows:
   - Set the **Width** property to "300".
   - Set the **Height** property to "35".
   - Set the **Orientation** property to **Horizontal**.
6. Set **c1ComboBox1**'s properties as follows:
   - Set the **Width** property to "150".
   - Set the **Height** property to "35".
   - Set the **Name** property to "Category"
7. Set **c1ComboBox2**'s properties as follows:
   - Set the **Width** property to "150".
   - Set the **Height** property to "35".
   - Set the **Name** property to "Shows".

   The project resembles the following:



You have completed the first step of the quick start by creating a WPF project and adding two C1ComboBox controls to it. In the next step, you'll add items to the first C1ComboBox control.

Step 2 of 4: Adding Items to the First C1ComboBox Control

In the last step, you created a project and added two C1ComboBox controls to it. In this step, you will add three items to the first combo box.

Complete the following steps:

1. Select the first C1ComboBox, **Category**.
2. In the **Properties** window, click the **Items** ellipsis button to open the **Collection Editor: Items** dialog box.
3. Click **Add** three times to add three C1ComboBoxItems to the control. Three C1ComboBoxItems named **c1ComboBoxItem1**, **c1ComboBoxItem2**, and **c1ComboBoxItem3**, are added to the control.
4. Set **c1ComboBoxItem1**'s properties as follows:
    - Set the **Content** property to "Comedy".
    - Set the **Height** property to "25".
5. Set **c1ComboBoxItem2**'s properties as follows:
    - Set the **Content** property to "Drama".
    - Set the **Height** property to "25".
6. Set **C1ComboBoxItem3**'s properties as follows:
    - Set the **Content** property to "Science Fiction".
    - Set the **Height** property to "25".
7. Click **OK** to close the **Collection Editor: Items** dialog box.

In this step, you added items to the first combo box. In the next step, you will add code to the project that will populate the second combo box with items when a user selects an item in the first combo box.

Step 3 of 4: Adding Code to the Control

In the last step, you added items to the first combo box. In this step, you will add code to the project that will populate the second combo box according to the option the user selects in the first combo box.

1. Select the first C1ComboBox control ("Category").
2. In the **Properties** window, click the **Events** ⚡ button.
3. Double-click the inside the **SelectedIndexChanged** text box to add the **C1ComboBox1_SelectedIndexChanged** event handler.

    The **MainPage.xaml.cs** page opens.

4. Import the following namespace into your project:
   - Visual Basic

     ```
     Imports System.Collections.Generic
     ```

   - C#

     ```
     using System.Collections.Generic;
     ```

5. Add the following code to the **C1ComboBox1_SelectedIndexChanged** event handler:
   - Visual Basic

     ```
     'Create List for Comedy selection
     Dim dropDownList_Comedy As New List(Of String)()
     dropDownList_Comedy.Add("Absolutely Fabulous")
     dropDownList_Comedy.Add("The Colbert Report")
     dropDownList_Comedy.Add("The Daily Show")
     dropDownList_Comedy.Add("The Office")

     'Create List for Drama selection
     Dim dropDownList_Drama As New List(Of String)()
     dropDownList_Drama.Add("Breaking Bad")
     dropDownList_Drama.Add("Desperate Housewives")
     dropDownList_Drama.Add("Mad Men")
     dropDownList_Drama.Add("The Sopranos")

     'Create List for Science Fiction selection
     Dim dropDownList_SciFi As New List(Of String)()
     dropDownList_SciFi.Add("Battlestar Galactica")
     dropDownList_SciFi.Add("Caprica")
     dropDownList_SciFi.Add("Stargate")
     dropDownList_SciFi.Add("Star Trek")

     'Check for SelectedIndex value and assign appropriate list to 2nd combo
     box
     If Category.SelectedIndex = 0 Then
        Shows.ItemsSource = dropDownList_Comedy
     ElseIf Category.SelectedIndex = 1 Then
        Shows.ItemsSource = dropDownList_Drama
     ElseIf Category.SelectedIndex = 2 Then
        Shows.ItemsSource = dropDownList_SciFi
     End If
     ```

- C#

```
//Create List for Comedy selection
List<string> dropDownList_Comedy = new List<string>();
dropDownList_Comedy.Add("Absolutely Fabulous");
dropDownList_Comedy.Add("The Colbert Report");
dropDownList_Comedy.Add("The Daily Show");
dropDownList_Comedy.Add("The Office");

//Create List for Drama selection
List<string> dropDownList_Drama = new List<string>();
dropDownList_Drama.Add("Breaking Bad");
dropDownList_Drama.Add("Desperate Housewives");
dropDownList_Drama.Add("Mad Men");
dropDownList_Drama.Add("The Sopranos");

//Create List for Science Fiction selection
List<string> dropDownList_SciFi = new List<string>();
dropDownList_SciFi.Add("Battlestar Galactica");
dropDownList_SciFi.Add("Caprica");
dropDownList_SciFi.Add("Stargate");
dropDownList_SciFi.Add("Star Trek");

//Check for SelectedIndex value and assign appropriate list to 2nd combo box
if (Category.SelectedIndex == 0)
{
     Shows.ItemsSource = dropDownList_Comedy;
}
else if (Category.SelectedIndex == 1)
{
     Shows.ItemsSource = dropDownList_Drama;
}
else if (Category.SelectedIndex ==2)
{
     Shows.ItemsSource = dropDownList_SciFi;
}
```

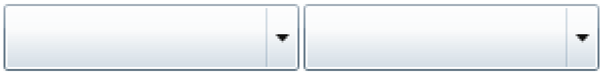
In the next step, you will run the project and observe the results of this quick start.

Step 4 of 4: Running the Project

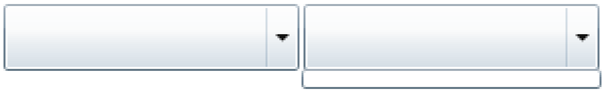
In the previous three steps, you created a WPF project with two combo boxes, added items to the first combo box, and wrote code that will populate the second combo box with items once an item is selected in the first combo box. In this step, you will run the project and observe the results of this quick start.

Complete the following steps:

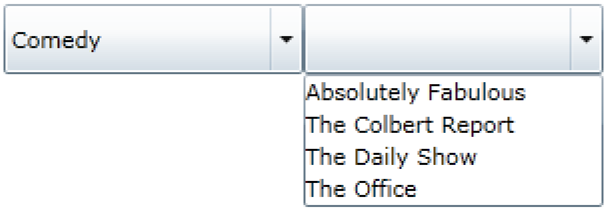
1. Press F5 to run the project. The project loads with two blank combo boxes:



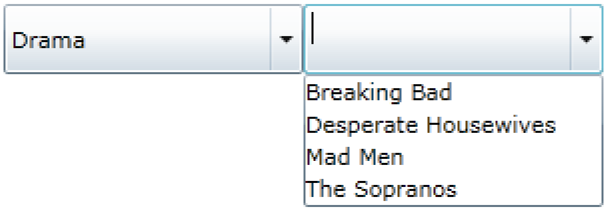
2. Click the second combo box's drop-down arrow and observe that the drop-down list is empty:



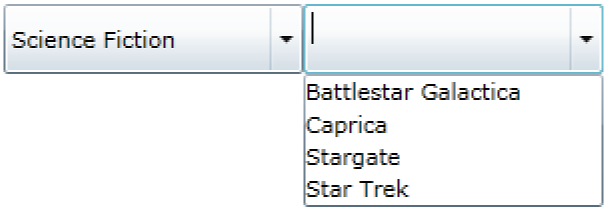
3. Click the first combo box's drop-down arrow and select **Comedy**.
4. Click the second combo box's drop-down arrow and observe that the drop-down list features the following items:



5. Click the first combo box's drop-down arrow and select **Drama**.
6. Click the second combo box's drop-down arrow and observe that the drop-down list features the following items:



7. Click the first combo box's drop-down arrow and select **Science Fiction**.
8. Click the second combo box's drop-down arrow and observe that the drop-down list features the following items:



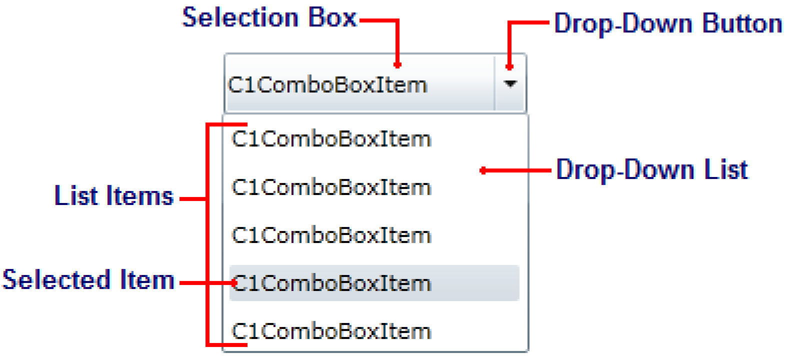
Congratulations! You have completed the **ComboBox for WPF** quick start.

Working with the C1ComboBox Control

This section provides an overview of C1ComboBox control basics. If you haven't used the control, we recommend starting with the ComboBox for WPF Quick Start (page 17) topic.

C1ComboBox Elements

The C1ComboBox control is a flexible control used to display data in a drop-down list. It is essentially the combination of two controls: a text box that allows users to enter a selection, and a list box that allows users to select from a series of list options. The following image diagrams the C1ComboBox control.



See below for a description of each C1ComboBox element.

- **Selection Box**

  The selection box serves two purposes: it allows users to enter the list item they're searching for directly into the text box, and it displays the currently selected item. The content of this box is equal to the content of the C1ComboBox control's selected index item.

- **Drop-Down Button**

  The drop-down button reveals the drop-down list when clicked.

- **Drop-Down List**

  The drop-down list consists of a series of list items (see below); it can contain as little or as many list items as you need. If the number of items exceeds the size of the drop-down list, a scrollbar will automatically appear.

- **List Items**

  Each list item in a drop-down list is represented by the C1ComboBoxItem class. List items can contain text, pictures, and even controls.

- **Selected Item**

  The selected item in a list can be fixed by the developer or chosen by a user at run-time. The value of a selected list item's IsSelected property is **True**.

C1ComboBox Features

The following topics detail a few of the C1ComboBox control's features. For more information on utilizing these features, see the ComboBox for WPF Task-Based Help (page 31) section.

Drop-Down List Direction

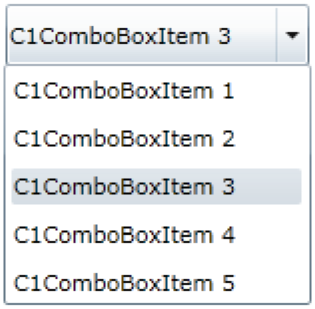
By default, when the user clicks the C1ComboBox control's drop-down arrow at run-time, the drop-down list will appear below the control; if that is not possible, it will appear above the control. You can, however, change the direction in which the drop-down list appears by setting the DropDownDirection property to one of the following four options:

Event	Description
BelowOrAbove (default)	Tries to open the drop-down list below the header. If it is not possible tries to open above it.
AboveOrBelow	Tries to open the drop-down list above the header. If it is not possible tries to open below it.
ForceBelow	Forces the drop-down list to open below the header.
ForceAbove	Forces the drop-down list to open above the header.

For instructions about how to change the drop-down direction, see Changing the Drop-Down List Direction (page 34).

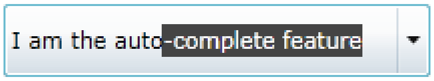
Item Selection

The SelectedIndex property determines which item is selected in a drop-down list. The SelectedIndex is based on a zero-based index, meaning that 0 represents the first C1ComboBoxItem, 1 represents the second C1ComboBoxItem, and so on. In the image below, the SelectedIndex is set to **2**, which selects the third C1ComboBoxItem.



AutoComplete

The C1ComboBox control features an auto-completion feature, which selects a list item based on user input. As the user types, the list item is loaded into the selection box, as seen in the following image:



I am the auto-complete feature

The user only has to press ENTER to select the list item suggested by the AutoComplete feature.

The AutoComplete feature can be disabled by setting the AutoComplete property to **False**. To learn how to disable the feature at design time, in XAML, and in code, see Disabling AutoComplete (page 35).

Drop-Down List Sizing

By default, the size of the drop-down list is determined by the width of the widest C1ComboBoxItem item and the collective height of all of the C1ComboBoxItem items, as the DropDownWidth and DropDownHeight properties are both set to **NaN**.

You can control the maximum width and maximum height of the drop-down list by setting the C1ComboBox control's MaxDropDownWidth and MaxDropDownHeight properties. Setting these properties ensures that the area of the drop-down list can never expand to a larger area than you've specified. If the width or height of the list exceeds the specified maximum height and width, scrollbars will automatically be added to the drop-down list.

For task-based help on drop-down list sizing, see Setting the Maximum Height and Maximum Width of the Drop-Down List (page 36).

ComboBox for WPF Layout and Appearance

The following topics detail how to customize the C1ComboBox 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 layout the control and to customize the control's actions.

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.

C1ComboBox and C1ComboBoxItem ClearStyle Properties

ComboBox for WPF supports ComponentOne's new ClearStyle technology that allows you to easily change control colors without having to change control templates. By just setting a few color properties you can quickly style the entire grid.

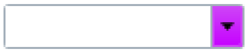
The following table outlines the brush properties of the **C1ComboBox** control:

Brush	**Description**
Background	Gets or sets the brush of the control's background.
ButtonBackground	Gets or sets the brush of the drop-down button's background.
ButtonForeground	Gets or sets the brush of the drop-down button's foreground.
FocusBrush	Gets or sets the brush for the control when it has focus.
MouseOverBrush	Gets or sets the brush for the control when it is moused over.
PressedBrush	Gets or sets the brush for the control when it is pressed.
SelectedBackground	Gets or sets the brush of the background for the selected C1ComboBoxItem.

The following table outlines the brush properties of the **C1ComboBoxItem** control:

Brush	**Description**
Background	Gets or sets the brush of the control's background.

You can completely change the appearance of the **C1ComboBox** and **C1ComboBoxItem** controls by setting a few properties, such as the **C1ComboBox** control's **ButtonBackground** property, which sets the background color for the control's drop-down arrow. For example, if you set the **C1ComboBox** control's **ButtonBackground** property to "#FFC500FF", each header in the **C1ComboBox** control would appear similar to the following:



It's that simple with ComponentOne's ClearStyle technology. For more information on ClearStyle, see the ComponentOne ClearStyle Technology (page 25) topic.

ComboBox for WPF Appearance Properties

ComponentOne ComboBox for WPF includes several properties that allow you to customize the appearance of the control. You can change the appearance of the text displayed in the control and customize graphic elements of the control. The following topics describe some of these appearance properties.

Text Properties

The following properties let you customize the appearance of text in the combo box control.

Property	Description
FontFamily	Gets or sets the font family of the control. This is a dependency property.
FontSize	Gets or sets the font size. This is a dependency property.
FontStretch	Gets or sets the degree to which a font is condensed or expanded on the screen. This is a dependency property.
FontStyle	Gets or sets the font style. This is a dependency property.
FontWeight	Gets or sets the weight or thickness of the specified font. This is a dependency property.
TextAlignment	Gets or sets how the text should be aligned in the drop-down list. This is a dependency property.

Content Positioning Properties

The following properties let you customize the position of header and content area content in the C1ComboBox control.

Property	Description
HorizontalContentAlignment	Gets or sets the horizontal alignment of the control's content. This is a dependency property.
VerticalContentAlignment	Gets or sets the vertical alignment of the control's content. This is a dependency property.

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.

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 **C1ComboBox** control.

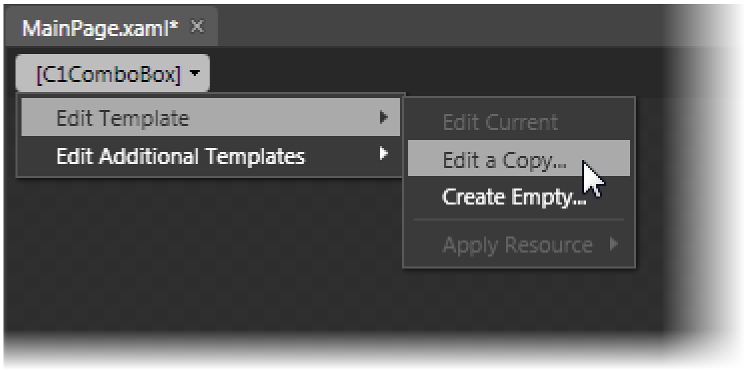
Property	Description
Height	Gets or sets the suggested height of the element. This is a dependency property.
MaxHeight	Gets or sets the maximum height constraint of the element. This is a dependency property.
MaxWidth	Gets or sets the maximum width constraint of the element. This is a dependency property.
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.
DropDownHeight	Gets or sets the height of the dropdown (set to Double.NaN to size automatically).
DropDownWidth	Gets or sets the width of the drop-down list (set to Double.NaN to size automatically).
MaxDropDownHeight	Gets or sets maximum height constraint of the drop-down box.
MaxDropDownWidth	Gets or sets maximum width constraint of the drop-down box.

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 ComboBox 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 C1ComboBox control and, in the menu, selecting **Edit Template**. Select **Edit a Copy** to create an editable copy of the current template or select **Create Empty** to create a new blank template.



If you want to edit the C1ComboBoxItem template, simply select the C1ComboBoxItem and, in the menu, select **Edit Template**. 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.

Additional ComboBox Templates

In addition to the default templates, the C1ComboBox control includes a few additional templates. These additional templates can also be accessed in Microsoft Expression Blend – in Blend select the C1ComboBox control and, in the menu, select **Edit Additional Templates**. Choose a template, and select **Create Empty.**

Item Templates

ComponentOne ComboBox for WPF's combo box control is an **ItemsControl**s that serves as a container for other elements. As such, the control includes templates to customize items places within the combo box. These templates include an **ItemTemplate**, an **ItemsPanel**, and an **ItemContainerStyle** template. You use the **ItemTemplate** to specify the visualization of the data objects, the **ItemsPanel** to define the panel that controls the layout of items, and the **ItemStyleContainer** to set the style of all container items.

Accessing Templates

You can access these templates in Microsoft Expression Blend by selecting the C1ComboBox control and, in the menu, selecting **Edit Additional Templates**. Choose **Edit Generated Items (ItemTemplate)**, **Edit Layout of Items (ItemsPanel)**, or **Edit Generated Item Container (ItemStyleContainer)** and select **Create Empty** to create a new blank template or **Edit a Copy**.

A dialog box will appear allowing you to name the template and determine where to define the template.

ComboBox for WPF Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio .NET and know how to use the C1ComboBox control in general. If you are unfamiliar with the **ComponentOne ComboBox for WPF** product, please see the **ComboBox for WPF** quick start first.

Each topic in this section provides a solution for specific tasks using the **ComponentOne ComboBox for WPF** product.

Each task-based help topic also assumes that you have created a new WPF project.

Working with ComboBox Items

The following topics illustrate several ways to add list items to the C1ComboBox control.

Adding ComboBox Items in the Designer

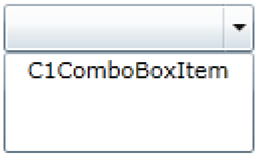
In this topic, you will learn how to add items to the C1ComboBox control in Expression Blend. This method is useful whenever you're creating a static combo box with just a few items.

Complete the following steps:

1. In the **Properties** window, click the **Items** ellipsis button ... to open the **Collection Editor: Items** dialog box.
2. Click **Add** to add a **C1ComboBoxItem** to the C1ComboBox control.

This Topic Illustrates the Following:

With the program running, click the drop-down arrow and observe that one item appears in the drop-down list as follows:



Adding ComboBox Items in XAML

In this topic, you will learn how to add items to the C1ComboBox control in XAML markup. This method is useful whenever you're creating a static combo box with just a few items.

Complete the following steps:

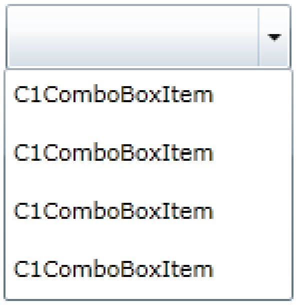
1. To add items to the C1ComboBox control, add the following XAML markup between the `<c1:C1ComboBox>` and `</c1:C1ComboBox>` tags:

```
<c1:C1ComboBoxItem Height="25" Content="C1ComboBoxItem"/>
<c1:C1ComboBoxItem Height="25" Content="C1ComboBoxItem"/>
<c1:C1ComboBoxItem Height="25" Content="C1ComboBoxItem"/>
<c1:C1ComboBoxItem Height="25" Content="C1ComboBoxItem"/>
```

2. Run the program.
3. Click the drop-down arrow and observe that four items appear in the drop-down list. The result resembles the following image:

✓ **This Topic Illustrates the Following:**

With the program running, click the drop-down arrow and observe that four items appear in the drop-down list. The result resembles the following image:



Adding ComboBox Items in Code

In this topic, you will learn how to add items to the C1ComboBox control in C# and Visual Basic code. This method is useful when you're creating a static combo box with just a few items.

Complete the following steps:

To disable AutoComplete, complete the following:

1. Open the **MainPage.xaml.cs** page.
2. Import the following namespace into your project:
   - Visual Basic

     ```
     Imports C1.WPF
     ```

   - C#

     ```
     Using C1.WPF;
     ```

3. Enter Code view and add the following code beneath the **InitializeComponent()** method:
   - Visual Basic

     ```
     C1ComboBox1.Items.Add(New C1ComboBoxItem() With {.Content =
     "C1ComboBoxItem1"})
     C1ComboBox1.Items.Add(New C1ComboBoxItem() With {.Content =
     "C1ComboBoxItem2"})
     C1ComboBox1.Items.Add(New C1ComboBoxItem() With {.Content =
     "C1ComboBoxItem3"})
     C1ComboBox1.Items.Add(New C1ComboBoxItem() With {.Content =
     "C1ComboBoxItem4"})
     ```

   - C#

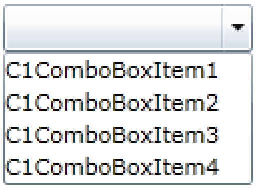
     ```
     c1ComboBox1.Items.Add(new C1ComboBoxItem() { Content =
     "C1ComboBoxItem1" });
     ```

```
c1ComboBox1.Items.Add(new C1ComboBoxItem() { Content =
"C1ComboBoxItem2" });
c1ComboBox1.Items.Add(new C1ComboBoxItem() { Content =
"C1ComboBoxItem3" });
c1ComboBox1.Items.Add(new C1ComboBoxItem() { Content =
"C1ComboBoxItem4" });
```

4. Run the program.

✓ **This Topic Illustrates the Following:**

With the project running, click the drop-down arrow and observe that four items appear in the drop-down list. The result resembles the following image:



C1ComboBoxItem1
C1ComboBoxItem2
C1ComboBoxItem3
C1ComboBoxItem4

Adding ComboBox Items from a Collection

In this topic, you will populate a combo box's drop-down list with a collection.

Complete the following steps:

1. Open the **MainPage.xaml.cs** page.
2. Import the following namespace into the project:
   - Visual Basic

     ```
     Imports System.Collections.Generic
     ```

   - C#

     `using System.Collections.Generic;`Create your list by adding the following code beneath the **InitializeComponent()** method:

   - Visual Basic

     ```
     Dim dropDownList As New List(Of String)()
     dropDownList.Add("C1ComboBoxItem1")
     dropDownList.Add("C1ComboBoxItem2")
     dropDownList.Add("C1ComboBoxItem3")
     dropDownList.Add("C1ComboBoxItem4")
     ```

   - C#

     ```
     List<string> dropDownList = new List<string>();
     dropDownList.Add("C1ComboBoxItem1");
     dropDownList.Add("C1ComboBoxItem2");
     dropDownList.Add("C1ComboBoxItem3");
     ```

```
dropDownList.Add("C1ComboBoxItem4");
```

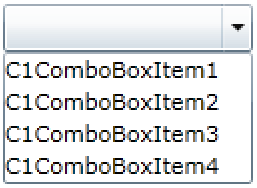
4. Add the list to the combo box by setting the **ItemsSource** property:
   - Visual Basic

     ```
     C1ComboBox1.ItemsSource = dropDownList
     ```
   - C#

     ```
     c1ComboBox1.ItemsSource = dropDownList;
     ```
5. Run the program.

✓ **This Topic Illustrates the Following:**

With the project running, click the drop-down arrow and observe that four items appear in the drop-down list. The result resembles the following image:



Changing the Drop-Down List Direction

By default, the drop-down list will attempt to open at the bottom of the control; if there is no room at the bottom to display the whole drop-down list, it will appear above the control. You can, however, specify where you would like the drop-down list to open.

In the Designer

Complete the following steps:

1. Click the C1ComboBox control once to select it.
2. In the **Properties** window, click the DropDownDirection drop-down arrow and select an option. For this example, select **ForceAbove**.
3. Run the program and click the drop-down arrow. Observe that the drop-down list appears above the control.

In XAML

Complete the following steps:

1. Add `DropDownDirection="ForceAbove"` to the `<c1:C1ComboBox>` tags so that the markup resembles the following:

   ```
   <c1:C1ComboBox Width="249" DropDownDirection="ForceAbove">
   ```

2. Run the program and click the drop-down arrow. Observe that the drop-down list appears above the control.

In Code

Complete the following steps:

1. Open the **MainPage.xaml.cs** page.
2. Add following code beneath the **InitializeComponent()** method:
    - Visual Basic

      ```
      C1ComboBox1.DropDownDirection = ForceAbove
      ```

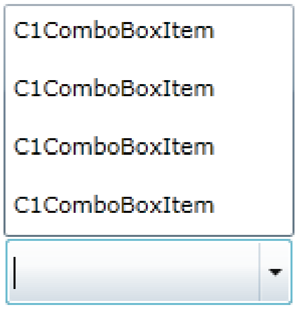
    - C#

      ```
      c1ComboBox1.DropDownDirection = ForceAbove;
      ```

3. Run the program and click the drop-down arrow. Observe that the drop-down list appears above the control.

This Topic Illustrates the Following:

In the following image, a combo box's drop-down list is forced to open above the control.



Disabling AutoComplete

By default, a user can type in the in the combo box's selection box to locate the item they want to select; you can disable this feature by setting the AutoComplete property to **False**.

In the Designer

Complete the following steps:

1. Click the C1ComboBox control once to select it.
2. In the **Properties** window, clear the AutoComplete check box.

In XAML

To disable AutoComplete, add `AutoComplete="False"` to the `<c1:C1ComboBox>` tag so that the markup resembles the following:

```
<c1:C1ComboBox HorizontalAlignment="Left" Width="249"
AutoComplete="False">
```

In Code

Complete the following steps:

1. Open the **MainPage.xaml.cs** page.
2. Add the following code beneath the **InitializeComponent()** method:
    - Visual Basic

      ```
      C1ComboBox1.AutoComplete = False
      ```

- C#

  ```
  c1ComboBox1.AutoComplete = false;
  ```

3. Run the program.

This Topic Illustrates the Following:

In this topic, you disabled the AutoComplete feature by setting the AutoComplete property to **False**. If you run the program and try to enter text, the control will not recommend a selection.

Setting the Maximum Height and Maximum Width of the Drop-Down List

You can specify the maximum height and maximum width of a combo box's drop-down list by setting its MaxDropDownHeight and MaxDropDownWidth properties. This topic assumes that the DropDownHeight and DropDownWidth properties are both set to **NaN**. For more information, see Drop-Down List Sizing (page 25).

In the Designer

Complete the following steps:

1. Click the C1ComboBox control once to select it.
2. In the **Properties** window, complete the following:
   - Set the MaxDropDownHeight to a value, such as "150".
   - Set the MaxDropDownWidth to a value, such as "350".
3. Run the program and click the combo box's drop-down arrow to see the result of your settings.

In XAML

Complete the following steps:

1. Add `MaxDropDownHeight="150"` and `MaxDropDownWidth="350"` to the `<c1:C1ComboBox>` tag so that the markup resembles the following:

   ```
   <c1:C1ComboBox HorizontalAlignment="Left" Width="249"
   MaxDropDownHeight="150" MaxDropDownWidth="350">
   ```

2. Run the program and click the combo box's drop-down arrow to see the result of your settings.

In Code

Complete the following steps:

1. Open the **MainPage.xaml.cs** page.
2. Add the following code beneath the **InitializeComponent()** method to set the DropDownHeight property :
   - Visual Basic

     ```
     C1ComboBox1.MaxDropDownHeight = 150
     ```

   - C#

     ```
     c1ComboBox1.MaxDropDownHeight = 150;
     ```

3. Add the following code beneath the **InitializeComponent()** method to set the DropDownWidth property :
   - Visual Basic

     ```
     C1ComboBox1.MaxDropDownWidth = 350
     ```

- C#

  ```
  c1ComboBox1.MaxDropDownWidth = 350;
  ```

4. Run the program and click the combo box's drop-down arrow to see the result of your settings.

✓ **This Topic Illustrates the Following:**

In this topic, you set the MaxDropDownWidth property to a value of 350 pixels and the MaxDropDownHeight property to a value of 150 pixels. With these settings, the width of the drop-down list will never be more than 350 pixels and the height will never be more than 150 pixels; however, the height and width can be *less* than 150 pixels by 350 pixels that if the items in the list aren't enough to fill that area.

Launching with the Drop-Down List Open

To launch the C1ComboBox with its drop-down list open, set the IsDropDownOpen property to **True**.

In the Designer

Complete the following steps:

1. Click the C1ComboBox control once to select it.
2. In the **Properties** window, select the IsDropDownOpen check box.
3. Run the program and observe that the drop-down list is open upon page load.

In XAML

Complete the following steps:

1. Add `IsDropDownOpen="True"` to the `<c1:C1ComboBox>` tag so that the markup resembles the following:

   ```
   <c1:C1ComboBox HorizontalAlignment="Left" Width="249"
   IsDropDownOpen="True">
   ```

2. Run the program and observe that the drop-down list is open upon page load.

In Code

Complete the following steps:

1. Open the **MainPage.xaml.cs** page.
2. Add the following code beneath the **InitializeComponent()** method:
   - Visual Basic

     ```
     C1ComboBox1.IsDropDownOpen = True
     ```

   - C#

     ```
     c1ComboBox1.IsDropDownOpen = true;
     ```

3. Run the program and observe that the drop-down list is open upon page load.

✓ **This Topic Illustrates the Following:**

In this topic, you set the IsDropDownOpen property to **True** so that the drop-down list would be open at run time. You can also use this property to open the drop-down list when a user mouses over the C1ComboBox control (see Opening the Drop-Down List on MouseOver (page 38)).

Opening the Drop-Down List on MouseOver

By default, the C1ComboBox control's drop-down list is only revealed when a user clicks the drop-down arrow. In this topic, you will write code that will cause the drop-down list to open whenever a user hovers over the control. This topic assumes that 1) you have already added a C1ComboBox control with at least one item to your project and 2) you are working in Expression Blend.

Complete the following:

1. Click the C1ComboBox control to select it.
2. In the **Properties** window, click the **Events** button to reveal the control's list of events.
3. Double-click inside of the **MouseEnter** text box. This will add the **C1ComboBox_MouseEnter** event handler to Code view.
4. Add the following code to the **C1ComboBox1_MouseEnter** event handler:
   - Visual Basic

     ```
     C1ComboBox1.IsDropDownOpen = True
     ```

   - C#

     ```
     c1ComboBox1.IsDropDownOpen = true;
     ```

5. Run the program.

This Topic Illustrates the Following:

With the program running, hover over the C1ComboBox control with your cursor. Observe that the drop-down list appears when you hover over the control. The drop-down list will stay open until you either select an item or click outside of the control.

Selecting an Item

You can select an item at run-time by setting the SelectedIndex property to the position of the item. This topic assumes that your project contains one C1ComboBox control with at least two C1ComboBoxItem items.

In the Designer

Complete the following steps:

1. Select the C1ComboBox control.
2. In the Properties window, set the SelectedIndex property to "1" so that the second C1ComboBoxItem will be selected.

In XAML

To set a selected item, add `SelectedIndex="0"` to the `<c1:C1ComboBoxItem>` tag so that the markup resembles the following:

```
<c1:C1ComboBoxItem Content="C1ComboBoxItem1" SelectedIndex="1">
```

In Code

Complete the following steps:

1. Open the **MainPage.xaml.cs** page.

2. Add the following code beneath the **InitializeComponent()** method:
   - Visual Basic

     ```
     C1ComboBoxItem1.SelectedIndex = 1
     ```

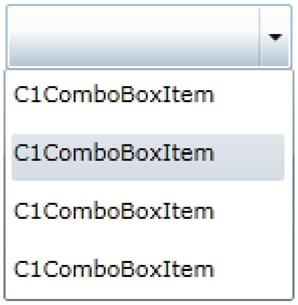
   - C#

     ```
     c1ComboBoxItem1.SelectedIndex = 1;
     ```

3. Run the program.

✓ **This Topic Illustrates the Following:**

When the drop-down list is revealed at run time, the second item will be selected, such as in the following image.


C1ComboBoxItem
C1ComboBoxItem
C1ComboBoxItem
C1ComboBoxItem