ComponentOne

Binding Expressions for Silverlight

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)

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

Table of Contents

ComponentOne Binding Expressions for Silverlight

Never write another binding converter again with **ComponentOne Binding Expressions™ for Silverlight**.

Save development time and write cleaner XAML by taking advantage of inline expression syntax. Concatenate strings, compute formulas, and even apply if/else logic directly in your binding statements without the need for complicated, code-behind converters.

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

Installing Studio for Silverlight

The following sections provide helpful information on installing **ComponentOne Studio for Silverlight**.

Studio for Silverlight Setup Files

The **ComponentOne Studio for Silverlight** installation program will create the following directory: **C:\Program Files\ComponentOne\Studio for Silverlight**. This directory contains the following subdirectories:

Bin	Contains copies of ComponentOne binaries (DLLs, EXEs, design-time assemblies).
Help	Contains documentation for all Studio components and other useful resources including XAML files.

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 Vista/Windows 7 machines:

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

Windows Vista and Windows 7 path: C:\Users\<username>\Documents\ComponentOne Samples\Studio for Silverlight

Esri Maps

Esri® files are installed with **ComponentOne Studio for Silverlight**, **ComponentOne Studio for WPF**, and **ComponentOne Studio for Windows Phone** by default to the following folders:

32-bit machine : C:\Program Files\ESRI SDKs\<Platform>\<version number>

64-bit machine: C:\Program Files (x86)\ESRI SDKs\<Platform>\<version number>

Files are provided for multiple languages, including: English, German (de), Spanish (es), French (fr), Italian (it), Japanese (ja), Portuguese (pt-BR), Russian (ru) and Chinese (zh-CN).

See Using Maps Powered by Esri (page 2) or visit the Esri website at http://www.esri.com for additional information.

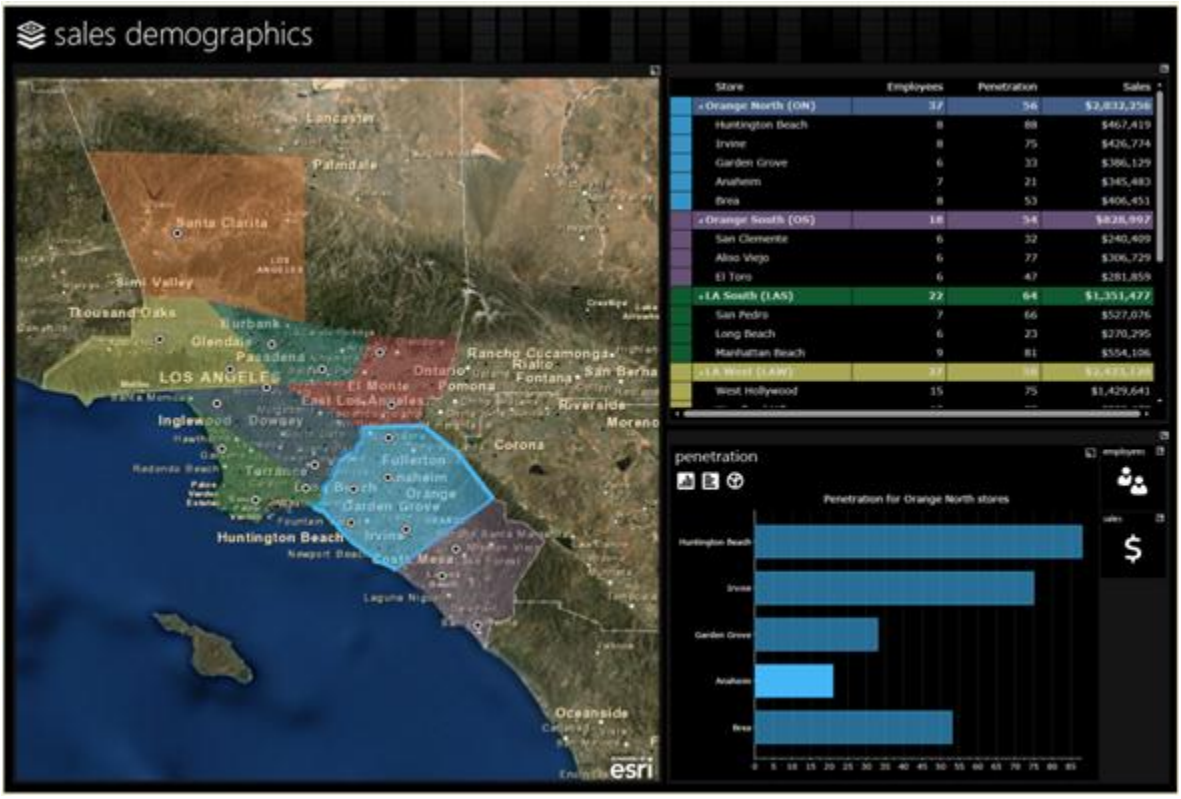
Using Maps Powered by Esri

Easily transform GIS data into business intelligence with controls for Silverlight, WPF, and Windows Phone powered by Esri® software.

By using the ComponentOne award-winning UI controls, you'll have the tools you need to seamlessly create rich, map-enabled user interfaces.

Benefits of Maps powered by Esri:

- Esri knows maps: Esri is the leading online map and GIS provider.
- Maps are technical: Using maps within your application is a very technical thing, so you don't want to take your chance using anyone but the best.
- Company of choice: Esri is the company of choice of many top companies and government agencies.
- Fulfill any developers' mapping needs: Esri mapping tools are flexible and will fill the needs of any mapping solution.



Esri Map Example

There are no additional charges for using the Esri maps included with ComponentOne products. Simply create a free online account at http://www.arcgisonline.com to start taking advantage of the Esri map controls. Esri licensing terms can be found in our Licensing Information and End User Licensing Agreement at http://www.componentone.com/SuperPages/Licensing/.

To learn more about Esri and Esri maps, please visit Esri at http://www.esri.com. There you will find detailed support, including documentation, forums, samples, and much more.

See the Studio for Silverlight Setup Files (page 1) topic for more information on the Esri files installed with this product.

System Requirements

System requirements for **ComponentOne Studio for Silverlight** include the following:

- Microsoft Silverlight 5.0 or later
- Microsoft Visual Studio 2010 or later

Installing Demonstration Versions

If you wish to try **ComponentOne Studio for Silverlight** 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 the registered version will stamp every application you compile so a ComponentOne banner will not appear when your users run the applications.

Uninstalling Studio for Silverlight

To uninstall **ComponentOne Studio for Silverlight**:

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

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

The **ComponentOne Studio for Silverlight** product is a commercial product. It is not shareware, freeware, or open source. If you use it in production applications, please purchase a copy from our Web site or from the software reseller of your choice.

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 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.

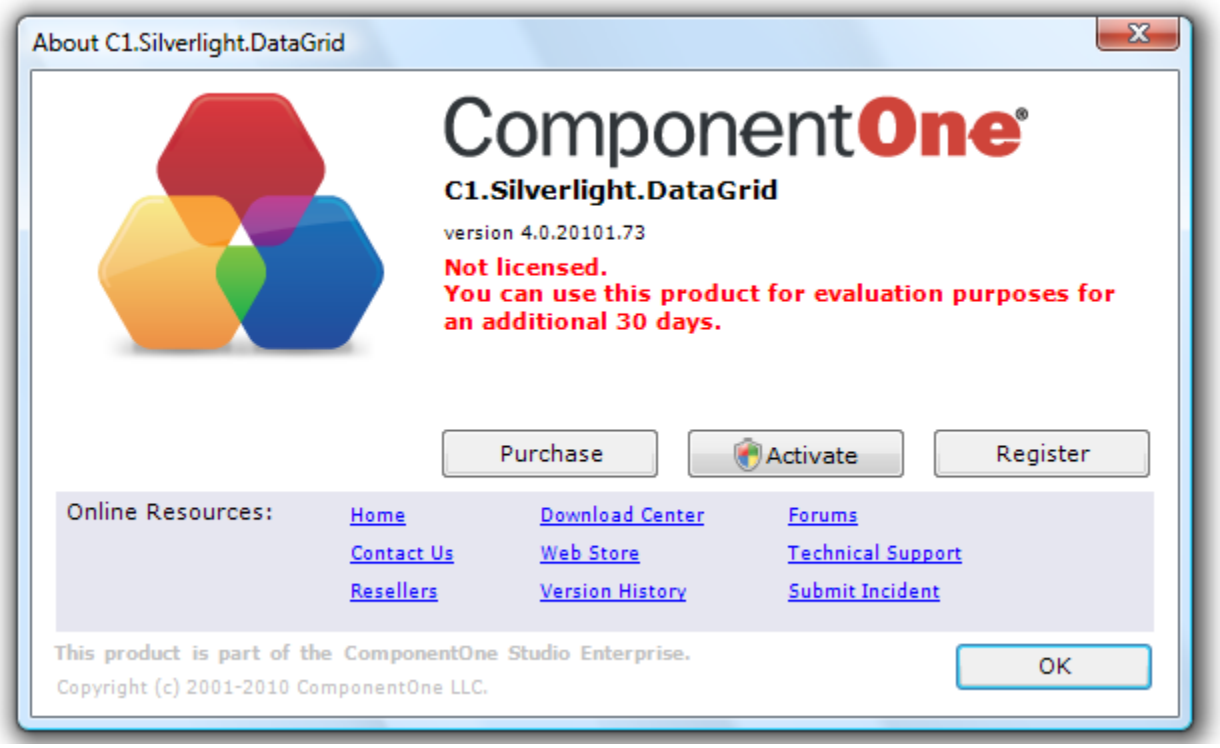
Without licensing, this type of distribution would not be practical for the vendor or convenient for the user. Vendors would either have to distribute evaluation software with limited functionality, or shift the burden of managing software licenses to customers, who could easily forget that the software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

Studio for Silverlight Licensing

Licensing for **ComponentOne Studio for Silverlight** is similar to licensing in other ComponentOne products but there are a few differences to note.

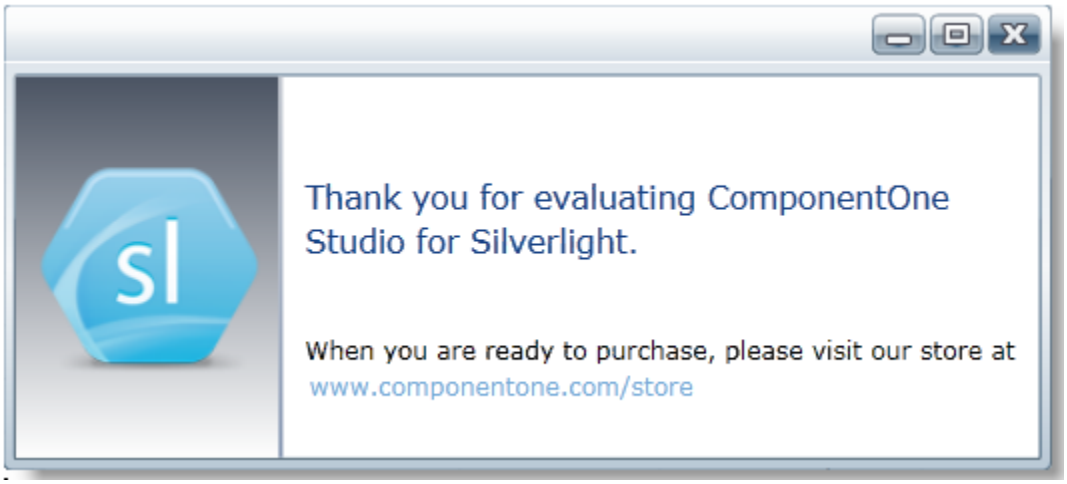
Initially licensing in handled similarly to other ComponentOne products. When a user decides to purchase a product, he receives an installation program and a Serial Number. During the installation process, the user is prompted for the serial number that is saved on the system.

In **ComponentOne Studio for Silverlight**, when a control is dropped on a form, a license nag dialog box appears one time. The nag screen appears similar to the following image:

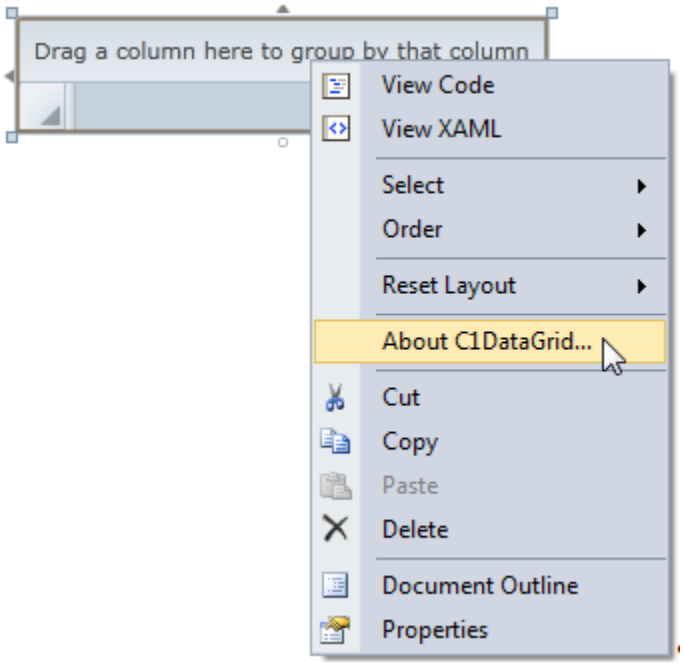


The **About** dialog box displays version information, online resources, and (if the control is unlicensed) buttons to purchase, activate, and register the product.

All ComponentOne products are designed to display licensing information at run time if the product is not licensed. None will throw licensing exceptions and prevent applications from running. Each time an unlicensed Silverlight application is run; end-users will see the following pop-up dialog box:



To stop this message from appearing, enter the product's serial number by clicking the **Activate** button on the **About** dialog box of any ComponentOne product, if available, or by rerunning the installation and entering the serial number in the licensing dialog box. To open the **About** dialog box, right-click the control and select the **About** option:



Note that when the user modifies any property of a ComponentOne Silverlight control in Visual Studio or Blend, the product will check if a valid license is present. If the product is not currently licensed, an attached property will be added to the control (the **C1NagScreen.Nag** property). Then, when the application executed, the product will check if that property is set, and show a nag screen if the **C1NagScreen.Nag** property is set to **True**. If the user has a valid license the property is not added or is just removed.

One important aspect of this of this process is that the user should manually remove all instances of **c1:C1NagScreen.Nag="true"** in the XAML markup in all files after registering the license (or re-open all the files

that include ComponentOne controls in any of the editors). This will ensure that the nag screen does not appear when the application is run.

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 Resources**
  ComponentOne provides customers with a comprehensive set of technical resources in the form of FAQs, samples and videos, Version Release History, 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.
- **Product Forums**
  ComponentOne's **product forums** are available for users to share information, tips, and techniques regarding ComponentOne products. ComponentOne developers will be available on the forums to share insider tips and technique and answer users' questions. 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 Studio for Silverlight 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.Silverlight.dll
- C1.Silverlight.Binding.5.dll
- C1.Silverlight.Binding.dll

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

About This Documentation

Acknowledgements

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.

Key Features

ComponentOne Binding Expressions for Silverlight provides the following unique key features:

- **Support for Operators and Functions**

  Perform simple operations (+, -, *, /) and functions directly in your binding statement. Supported functions include Sum, Average, and Concatenate. For a list of operators and functions, see C1Binding Expression Syntax Elements (page 13).

- **Use Quotes without Breaking your XAML**

  Use quotes in your binding expressions without breaking your XAML. C1Binding expressions support inline quotes in two forms: &**quot**; or use of the pipe (|) character.

- **Logical Expressions**

  Apply simple if/else logic directly in your binding statement. **C1Binding** supports an 'IF' function with the following syntax: if(condition, true value, false value).

- **Clean and Expressive XAML**

  By using C1Binding Expressions (page 9), your XAML will be much cleaner and more expressive. And it's 100% self-contained. You do not need any external converters referenced from elsewhere in your solution.

- **Calculation Engine**

  In addition to the C1Binding class, the **C1.Silverlight.Binding** and **C1.WPF.Binding** assemblies provide a C1CalcEngine class that is responsible for evaluating the expressions. The C1CalcEngine class can be used independently of the C1Binding, to implement calculated input boxes, spreadsheets, and many other applications. See the C1CalcEngine Class (page 12) for more information and some usage scenarios.

- **Silverlight 5 Support**

The **C1.Silverlight.Binding** classes require Silverlight 5. This is because they rely on MarkupExtensions, which are not available in earlier versions of Silverlight.

Introduction to ComponentOne Binding Expressions

WPF and Silverlight use Binding objects to connect data values to UI properties. For example, to display the name of a customer in a **TextBlock** element, you could use this XAML:

```
<TextBlock
  Text="{Binding FirstName}" />
```

In this example, the **Text** property of the **TextBlock** will be automatically synchronized with the content of the **FirstName** property of the data object currently assigned to the **DataContext** property of the **TextBox** element or any of its ancestors.

The **Binding** class is convenient but not very flexible. For example, if you wanted to bind the **TextBlock** above to the customer's full name, or to make the text red when the customer account is negative, you would need a **Converter**.

The **Converter** parameter of the **Binding** object allows you to add arbitrary logic to the binding, converting values when they are transferred between the source and the target.

To illustrate, and still using the example above, imagine we wanted the **TextBlock** to show the customer's full name, and to display the text in red when the customer's account is negative. To accomplish this, we would need two converters, implemented as follows:

```
/// <summary>
/// Converter for Foreground property: returns a red brush if the
/// customer amount is negative, black otherwise.
/// </summary>
public class ForegroundConverter : System.Windows.Data.IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
    CultureInfo culture)
  {
    var c = value as Customer;
    return c.Amount < 0
      ? new SolidColorBrush(Colors.Red)
      : new SolidColorBrush(Colors.Black);
  }
  public object ConvertBack(object value, Type targetType, object parameter,
    CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}

/// <summary>
/// Converter for customer's full name: returns a string containing
/// the customer's full name.
/// </summary>
public class FullNameConverter : System.Windows.Data.IValueConverter
{
```

```
  public object Convert(object value, Type targetType, object parameter,
    CultureInfo culture)
  {
    var c = value as Customer;
    return string.Format("{0} {1}", c.FirstName, c.LastName);
  }
  public object ConvertBack(object value, Type targetType, object parameter,
    CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

Once the converters have been defined, they can be used in XAML as follows:

```
<Grid>
  <Grid.Resources>
    <local:FullNameConverter x:Key="_cvtFullName" />
    <local:ForegroundConverter x:Key="_cvtForeground" />
  </Grid.Resources>

  <TextBlock
    Text="{Binding Converter={StaticResource _cvtFullName}}"
    Foreground="{Binding Converter={StaticResource _cvtForeground}}" />
</Grid>
```

This solution works fine, but there are a few shortcomings:

- You can't tell what the output will look like by looking at the XAML. The color converter could, for example, return any color, using any logic. The only way to figure this out would be to look at the converter implementation.
- If you wanted to set the **Opacity**, **Visibility**, or **FontWeight** properties of the **TextBlock** element, you would need to create more converter classes, and they would be tightly coupled with your XAML.
- The semantics of the converter class only stipulate that it should convert an object into another object. This provides a lot of flexibility, but it also makes converters relatively fragile.

The C1Binding Class

The C1Binding class addresses these issues by allowing you to use rich expressions in your bindings. Using C1Binding, the example above could be written as:

```
<TextBlock
  Text="{c1:C1Binding Expression='concatenate(FirstName, | |, LastName)' }"
  Foreground="{c1:C1Binding Expression='if(Amount &lt; 0, |red|, |black|)' }" />
```

Notice how much simpler and more direct this version is. There are no resources and no converters. The meaning of the bindings is clear from the XAML.

If the XAML author decided that the full name should be displayed as "Last, First" instead, he could make the change directly in the XAML. If he wanted negative amounts to be shown in bold, that would be easy too:

```
<TextBlock
  Text="{c1:C1Binding Expression='concatenate(FirstName, | |, LastName)' }"
  Foreground="{c1:C1Binding Expression='if(Amount &lt; 0, |red|, |black|)' }" />
  FontWeight="{c1:C1Binding Expression='if(Amount &lt; 0, |bold|, |normal|)' }" />
```

In the expressions above, the vertical bars represent quotes. They could also be written as ***"***, but the vertical bars are easier to type and read.

The "**Expression=**" term in the expressions above is optional in WPF. It is required in Silverlight 5, currently in beta. It may not be required by the time Silverlight 5 ships.

Using C1Binding Expression Syntax

C1Binding objects use the C1CalcEngine class to parse and evaluate expressions. The syntax supported is similar to the one used in Microsoft Excel®.

You can use all the usual logical operators (=, >, <,.<>, >=, <=), arithmetic (+, -, *, /, ^), and you can group expressions using parentheses. For example:

"1+2*3 < (1+2)*3" returns TRUE (7 < 9).

Any public properties in the binding source object can be used in expressions as well. For example:

"Price * 8.5%" returns the value of the **Price** property times 0.085.

Properties that return collections or dictionaries are also supported. For example, assuming the **DataContext** object has a **Children** property:

"Children(Children.Count-1).Name" returns the name of the last child.

Finally, the C1CalcEngine supports a subset of the functions available in Excel. See C1Binding Expression Syntax Elements (page 13) for a list of these functions.

C1Binding Usage Scenarios

Simple Arithmetic:
Suppose you want to display taxes due on a certain amount. Using traditional bindings, you would need a converter to calculate the tax amount. Using **C1Binding**, you can calculate the amount using a simple expression:

```
<TextBlock
  Text="{c1:C1Binding Expression='Amount * 8.5%' }"
```

Combining Values:
Suppose you want to extend the scenario above and display the total amount in addition to the tax amount. You could do that using several **TextBlock** elements, but it would be more efficient to use a single **TextBlock** and a C1Binding based on the CONCATENATE function:

```
<TextBlock
  Text="{c1:C1Binding Expression=
    'CONCATENATE(Amount, | Tax: |, Amount * 8.5%' }"
```

Formatting Values:
C1Binding and regular **Binding** objects have a **StringFormat** property that you can use to control the format of bound values. If your binding expression requires formatting several parts of the output, you can use the TEXT function as follows:

```
<TextBlock
  Text="{c1:C1Binding Expression=
```

```
    'CONCATENATE(TEXT(Amount, |c|), | Tax: |, TEXT(Amount * 8.5%, |c|))' }"
```

Conditional Formatting:
Suppose you want to create a user interface where amounts greater than a certain value are displayed in bold and red. You can accomplish this using the IF function, which performs conditional logic:

```
<TextBlock
  Text="{c1:C1Binding Expression='Amount', StringFormat='c'}"
  Foreground="{c1:C1Binding Expression='if(Amount &gt; 1000, |red|, |black|)' }" />
  FontWeight="{c1:C1Binding Expression='if(Amount &gt; 1000, |bold|, |normal|)' }" />
```

C1Binding Limitations

C1Binding objects can only be used in one-way binding scenarios.

The reason for this is simple: regular bindings are based on simple properties, which can be evaluated or assigned values. C1Binding, however, is based on expressions, which can be evaluated but not assigned to. For example, you can assign a new value to the "Amount" property, but not to the "Credit – Debit" expression.

In addition to this, C1Binding objects have the same limitations as regular **Binding** objects:

- They can only be assigned to dependency properties, which only exist in dependency objects (such as **UIElement** or **FrameworkElement** objects).
- If the binding is to be automatically updated when the property changes, then the binding source object must implement the **INotifyPropertyChanged** interface.
- If the types of the source and target properties are not compatible, then the target property must have the appropriate converters (e.g., most types can be converted from string values, including **Color**, **FontWeight**, and so on).

The C1CalcEngine Class

The C1CalcEngine class is responsible for parsing and evaluating the expressions used in **C1Binding** objects, but it is a public class and can also be used independently of C1Binding.

The C1CalcEngine class has two main methods: **Parse** and **Evaluate**:

1. **Parse** parses strings into **Expression** objects.
2. **Evaluate** parses strings and evaluates the resulting expression.

C1CalcEngine Usage Scenarios

Calculated TextBox:
Suppose you want to add a calculated **TextBox** to your application. Users should be able to type expressions into the **TextBox**. The expressions should be evaluated when the control loses focus or when the user presses Enter. You could do this with the following code:

```
public MainWindow()
{
  InitializeComponent();

  var tb = new TextBox();
  tb.LostFocus += (s,e) => { Evaluate(s); };
  tb.KeyDown += (s, e) => { Evaluate(s); };
}
void Evaluate(object sender)
{
  var tb = sender as TextBox;
  var ce = new C1.WPF.Binding.C1CalcEngine();
  try
  {
    var value = ce.Evaluate(tb.Text);
    tb.Text = value.ToString();
  }
  catch (Exception x)
  {
    MessageBox.Show("Error in Expression: " + x.Message);
  }
}
```

Turning Grids into Spreadsheets:
Another typical use for the C1CalcEngine class is the implementation of calculated cells in grid controls and data objects. We have used it to implement several samples that add spreadsheet functionality to grid controls. The code is too long to include here, but you can see one of the samples in action here: http://demo.componentone.com/Silverlight/ExcelBook/).

C1Binding Expression Syntax Elements

The C1Binding class uses the C1CalcEngine class to parse and evaluate expressions. The syntax elements described below apply to both classes.

Case

All expressions are case-insensitive.

Operators

Expressions may contain the following operators:

Operator Type	Operator Symbols
Comparison	< > = <= >=
Addition/Subtraction	+ -
Multiplication/Division	* /
Power	^
Grouping	( ) , .

The operators follow the usual precedence rules, so "1 + 2 * 3" equals 7. Expressions may be grouped using parentheses, so "(1 + 2) * 3" equals 9.

Functions

Expressions may include calls to the following functions:

Logical Functions

Name	Description	Syntax
AND	Returns TRUE if all of its arguments are TRUE	=AND(logical1[, logical2,…])
FALSE	Returns the logical value FALSE	=FALSE
IF	Specifies a logical test to perform	=IF(logical_test, value_if_true, value_if_false)
NOT	Reverses the logic of its argument	=NOT(logical)
OR	Returns TRUE if any argument is TRUE	=OR(logical1[, logical2,…])
TRUE	Returns the logical value TRUE	=TRUE

Mathematical Functions

Name	Description	Syntax
ABS	Returns the absolute value of a number	=ABS(number)
ACOS	Returns the arccosine of a number	=ACOS(number)
ASIN	Returns the arcsine of a number	=ASIN(number)
ATAN	Returns the arctangent of a number	=ATAN(number)
ATAN2	Returns the arctangent from x- and y-coordinates	=ATAN2(x_num, y_num)
CEILING	Rounds a number to the nearest integer or to the nearest multiple of significance	=CEILING(number)
COS	Returns the cosine of a number	=COS(number)
COSH	Returns the hyperbolic cosine of a number	=COSH(number)
EXP	Returns e raised to the power of a given number	=EXP(number)
FLOOR	Rounds a number down, toward zero	=FLOOR(number)
INT	Rounds a number down to the nearest integer	=INT(number)
LN	Returns the natural logarithm of a number	=LN(number)
LOG	Returns the logarithm of a number to a specified base	=LOG(number[, base])
LOG10	Returns the base-10 logarithm of a number	=LOG10(number)
PI	Returns the value of the PI constant	=PI()
POWER	Returns the result of a number raised to a power	=POWER(number, power)
RAND	Returns a random number between 0 and 1	=RAND()
RANDBETWEEN	Returns a random number between the numbers you specify	=RANDBETWEEN(bottom, top)
SIGN	Returns the sign of a number	=SIGN(number)
SIN	Returns the sine of the given angle	=SIN(number)
SINH	Returns the hyperbolic sine of a number	=SINH(number)
SQRT	Returns a positive square root	=SQRT(number)
SUM	Adds its arguments	=SUM(number1[, number2, …])
TAN	Returns the tangent of a number	=TAN(number)
TANH	Returns the hyperbolic tangent of a number	=TANH(number)
TRUNC	Truncates a number to an integer	=TRUNC(number)";

Statistical Functions

Name	Description	Syntax
AVERAGE	Returns the average of its arguments	
AVERAGEA	Returns the average of its arguments, including numbers, text, and logical values	=AVERAGE(number1 [, number2, …])
COUNT	Counts how many numbers are in the list of arguments	=AVERAGEA(number1 [, number2, …])
COUNTA	Counts how many values are in the list of arguments	=COUNT(number1 [, number2, …])
COUNTBLANK	Counts the number of blank cells within a range	=COUNTA(number1 [, number2, …])
COUNTIF	Counts the number of cells within a range that meet the given criteria	=COUNTIF(range, criteria)
MAX	Returns the maximum value in a list of arguments	=MAX(number1 [, number2, …])
MAXA	Returns the maximum value in a list of arguments, including numbers, text, and logical values	=MAXA(number1 [, number2, …])
MIN	Returns the minimum value in a list of arguments	=MIN(number1 [, number2, …])
MINA	Returns the smallest value in a list of arguments, including numbers, text, and logical values	=MINA(number1 [, number2, …])
STDEV	Estimates standard deviation based on a sample	=STDEV(number1 [, number2, …])
STDEVA	Estimates standard deviation based on a sample, including numbers, text, and logical values	=STDEVA(number1 [, number2, …])
STDEVP	Calculates standard deviation based on the entire population	=STDEVP(number1 [, number2, …])
STDEVPA	Calculates standard deviation based on the entire population, including numbers, text, and logical values	=STDEVPA(number1 [, number2, …])
VAR	Estimates variance based on a sample	=VAR(number1 [, number2, …])
VARA	Estimates variance based on a sample, including numbers, text, and logical values	=VARA(number1 [, number2, …])
VARP	Calculates variance based on the entire population	=VARP(number1 [, number2, …])
VARPA	Calculates variance based on the entire population, including numbers, text, and logical values	=VARPA(number1 [, number2, …])

Text Functions

Name	Description	Syntax
CODE	Returns a numeric code for the first character in a text string	=CODE(text)
CONCATENATE	Joins several text items into one text item	=CONCATENATE(text1 [, text2, …])
FIND	Finds one text value within another (case-sensitive)	=FIND(find_text, within_text [, start_num])
LEFT	Returns the leftmost characters from a text value	=LEFT(text[, num_chars])
LEN	Returns the number of characters in a text string	=LEN(text)
LOWER	Converts text to lowercase	=LOWER(text)
MID	Returns a specific number of characters from a text string starting at the position you specify	=MID(text, start_num, num_chars)
PROPER	Capitalizes the first letter in each word of a text value	=PROPER(text)
REPLACE	Replaces characters within text	=REPLACE(old_text, stat_num, num_chars, new_text)
REPT	Repeats text a given number of times	=REPT(trext, number_times)
RIGHT	Returns the rightmost characters from a text value	=RIGHT(text[, num_chars])
SEARCH	Finds one text value within another (not case-sensitive)	=SEARCH(find_text, within_text[, start_num])
SUBSTITUTE	Substitutes new text for old text in a text string	=SUBSTITUTE(text, old_text, new_text[, instance_num])
T	Converts its arguments to text	=T(value)
TEXT	Formats a number and converts it to text	=TEXT(value, format_text)
TRIM	Removes spaces from text	=TRIM(text)
UPPER	Converts text to uppercase	=UPPER(text)
VALUE	Converts a text argument to a number	=VALUE(text)

C1Binding Sample

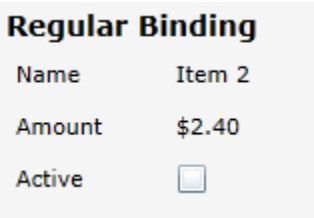
The **C1BindingDemo** sample installed with the product shows the differences between regular binding and C1Binding as well as template binding for each.

For example, here is the XAML used to perform regular binding:

```
<!-- regular binding -->
        <Border Grid.Row="1" Background="WhiteSmoke" Padding="8" >
            <StackPanel Orientation="Vertical" >
                <TextBlock Text="Regular Binding" Style="{StaticResource _styTitle}" />
```

```
                <StackPanel Orientation="Horizontal" Margin="5" >
                    <TextBlock Text="Name" Width="80" />
                    <TextBlock Text="{Binding Name}" />
                </StackPanel>
                <StackPanel Orientation="Horizontal" Margin="5" >
                    <TextBlock Text="Amount" Width="80" />
                    <TextBlock Text="{Binding Amount, StringFormat=c}" />
                </StackPanel>
                <StackPanel Orientation="Horizontal" Margin="5" >
                    <TextBlock Text="Active" Width="80" />
                    <CheckBox IsChecked="{Binding Active}" />
                </StackPanel>
            </StackPanel>
        </Border>
```

When the project runs, the regular binding looks like this:



We can compare this to the XAML used to perform **C1Binding**:

```
<!-- C1Binding -->
        <Border Grid.Row="2" Background="Yellow" Padding="8" >
            <StackPanel Orientation="Vertical" >
                <TextBlock Text="C1Binding" Style="{StaticResource _styTitle}" />
                <StackPanel Orientation="Horizontal" Margin="5" >
                    <TextBlock Text="Name" Width="80" />
                    <TextBlock
                        Text="{c1:C1Binding Expression=Name}"
                        FontWeight="{c1:C1Binding Expression='if(Active, |Bold|, |Normal|)'}"
                        Foreground="{c1:C1Binding Expression='if(Active, |Blue|, |Red|)'}"
                        />
                </StackPanel>
                <StackPanel Orientation="Horizontal" Margin="5" >
```

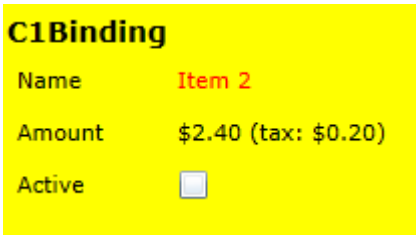
```
                    <TextBlock Text="Amount" Width="80" />
                    <TextBlock
                        Text="{c1:C1Binding Expression='concatenate(text(Amount, |c|), |
(tax: |, text(Amount * 8.5%, |c|), |)|)', StringFormat=c}"
                        FontWeight="{c1:C1Binding Expression='if(Active, |Bold|, |Normal|)'}"
                        />
                </StackPanel>
                <StackPanel Orientation="Horizontal" Margin="5" >
                    <TextBlock Text="Active" Width="80" />
                    <CheckBox IsChecked="{c1:C1Binding Expression=Active}" />
                </StackPanel>
            </StackPanel>
        </Border>
```

In this example, **C1Binding** expressions are used to make the binding a little more interesting. In this XAML, conditional formatting is used to display the **FontWeight** as **Bold** if the item is active or as N**ormal** if not. The **ForeGround** is **Blue** if the item is active or **Red** if not.

```
                    <TextBlock
                        Text="{c1:C1Binding Expression=Name}"
                        FontWeight="{c1:C1Binding Expression='if(Active, |Bold|, |Normal|)'}"
                        Foreground="{c1:C1Binding Expression='if(Active, |Blue|, |Red|)'}"
                        />
```

Then the amount is shown with the tax, which has been calculated using a multiplication operator, using the CONCATENATE function.

```
                    <TextBlock Text="Amount" Width="80" />
                    <TextBlock
                        Text="{c1:C1Binding Expression='concatenate(text(Amount, |c|), |
(tax: |, text(Amount * 8.5%, |c|), |)|)', StringFormat=c}"
                        FontWeight="{c1:C1Binding Expression='if(Active, |Bold|, |Normal|)'}"
                        />
```



The template binding is very similar, however, items are listed in a **<ListBox.ItemTemplate>**. The same C1Binding expressions are used as in the C1Binding XAML.

To get the regular binding to look and perform like the C1Binding, many **Converter** parameters would need to be used. C1Binding keeps the XAML clean and simple.