Behind the Scenes: Implementing Filtering for Windows Forms Data Binding

Cheryl Simmons  
Microsoft Corporation  
April 2008

Applies to:

* Microsoft Visual Studio 2005
* Microsoft .NET Framework 2.0
* Microsoft Windows Forms

Summary: Learn how to extend the generic **BindingList** and implement the filtering functionality of the **IBindingListView**.

**Contents**

* Introduction
* Filtering Overview
* Determining the Filtering Expression Format
* Creating a Generic Filtered Class
* Disabling Advanced Sorting
* Implementing the SupportsFiltering Property
* Implementing the Filter Property
* Implementing the RemoveFilter Method
* Using the Filtered List
* Next Steps
* Conclusion

# Introduction

When you bind Windows Forms controls to an ADO.NET data source, specifically with a **BindingSource** as an intermediary, the searching, sorting, and filtering capabilities built in to a **DataView** are automatically exposed by the **BindingSource**. However, when you want to bind to a list of business objects, you have to implement these features yourself. A good solution is a generic implementation that can be reused with various business objects. This article describes how to extend the generic **BindingList** and implement the filtering functionality of the **IBindingListView**. This implementation enables you to store custom business objects in a list that you can search, sort, and filter.

# Filtering Overview

In the summer of 2006, I published an article that demonstrated how to extend the generic **BindingList** and implement basic searching and sorting functionality. To read this article, see [Behind the Scenes: Improvements to Windows Forms Data Binding in the .NET Framework 2.0, Part 2](http://go.microsoft.com/fwlink/?LinkID=69697). I use the example presented in that article as the starting point for this article, and I add filtering capabilities. The final result is an extension of the generic binding list that you can search, sort, and filter. The example provided in this article correctly interacts with the **BindingSource** and bound controls to offer filtering functionality. In addition, I’ve designed this example to interoperate with the drop-down filtering implemented in [Building a Drop-Down Filter List for a DataGridView Column Header Cell](http://go.microsoft.com/fwlink/?LinkId=105803). The filtering implementation, like the sorting implementation, relies on the **IComparable** interface to do the heavy lifting.

Filtering can be implemented in various ways. In this article I walk you through the following steps:

* Determining the filtering expression format to use.
* Creating a generic filtered class.
* Disabling the advanced sorting capabilities of IBindingListView.
* Implementing the filtering capabilities of IBindingListView.
* Using and testing the filtered list populated with business objects and bound to a DataGridView control.

# Determining the Filtering Expression Format

Before I can implement filtering, I have to determine the format of filtering expressions that my application will accept. A filtering expression can follow any format that you want. However, to maintain consistency with ADO.NET, the filter expressions used for this article follows a subset of the guidelines established by the **DataColumn.Expression** property. These are the same guidelines used by the **DataView.RowFilter** property for single-column filtering. The **RowFilter** property expects a column name followed by an operator and a value to filter on. Since I am working with business objects, the filter I demonstrate expects a property name instead of a column name. To interoperate with the **DataGridView** drop-down auto-filtering, I accept optional brackets around the property name, which allow symbols to be used in the filtering expression. In addition, the filtering is limited to the equal to, less than, and greater than operators (=, <, >). The filter value should be in single quotation marks if it is a string or date value. The following are some examples of filtering expressions that my code handles.

* LastName = 'Smith'
* Salary > 50000
* Birthday < ‘1/1/1968’

# Creating a Generic Filtered Class

To add filtering, I start by creating a generic class named **FilteredBindingList**. This class extends the generic binding list and implements the **IBindingListView** interface. The code I use for searching and sorting is very similar to the code I presented in [Behind the Scenes: Improvements to Windows Forms Data Binding in the .NET Framework 2.0, Part 2](http://go.microsoft.com/fwlink/?LinkID=69697). In the next few sections I discuss code I added to implement the filtering portion of the **IBindingListView**. The following is the class declaration. I extend the generic **BindingList** class and implement the **IBindingListView** interface.

# Disabling Advanced Sorting

The **IBindingListView** interface extends the **IBindingList** interface by offering filtering and advanced (multi-column) sorting capability. Since this example of **IBindingListView** implements the filtering portion only, I disable the advanced sorting capability. I set the **SupportsAdvancedSorting** property to always return false to notify bound controls that the data source does not perform advanced sorting. I set the **SortDescriptions** property to return **null**. I set the **ApplySort** overload that takes a collection of sort descriptions to throw a **NotSupportedException** to indicate that multiple-column sorting cannot be performed. The following code shows how I disable the advanced sorting capability of **IBindingListView**.

# Implementing the SupportsFiltering Property

The first step to provide filtering is to implement the **SupportsFiltering** property. The **SupportsFiltering** property notifies bound controls and components that filtering is supported. The following code implements **SupportsFiltering** and always returns **true**.

public bool SupportsFiltering

{

get { return true; }

}

# Implementing the Filter Property

The next requirement is to implement the **Filter** property. The **Filter** property is the guts of filtering for the data source. When the filter property is set, it is applied to the data source and the data source returns the items that fit the filter criteria.

First, I declare a private string to hold the filter, and then expose it publicly, as shown in the following code.

private string filterValue = null;

public string Filter

{

...

}

The get implementation for the Filter property is simple; just return filterValue.

get

{

return filterValue;

}

Implementing the set operation for the **Filter** property is more difficult and requires some preliminary steps. As mentioned previously, when the filter value is set, the filter is applied to the list and the list returns the items that meet the filter criteria. In addition, the original list must be cached because the filter can be removed, in which case, the list should revert to its original contents. Therefore, the unfiltered list is stored in a read-only property named **OriginalList**. I chose a public read-only property to allow for the code to be unit-tested. The following code shows the declaration for the unfiltered list.

private List<T> originalListValue = new List<T>();

public List<T> OriginalList

{

get

{ return originalListValue; }

}

Since I want to enable the user to add and remove items from the list, I override the **OnListChanged** method and check for items added and removed from the list. To keep my unfiltered list up-to-date, I add or remove these items from the unfiltered list as necessary.

To eliminate problems with the bound controls, I check to see whether the list is filtered, and turn off the ability to add items to the list if a filter is applied. Since the ability to filter can be overridden on the client-side, I check to see whether a filter is applied when an item is added. Once the item is added, I reapply the filter to the list, which includes the newly added item.

protected override void OnListChanged(ListChangedEventArgs e)

{

// If the list is reset, check for a filter. If a filter

// is applied, don't allow items to be added to the list.

if (e.ListChangedType == ListChangedType.Reset)

{

if (Filter == null || Filter == "")

AllowNew = true;

else

AllowNew = false;

}

// Add the new item to the original list.

if (e.ListChangedType == ListChangedType.ItemAdded)

{

OriginalList.Add(this[e.NewIndex]);

if (!String.IsNullOrEmpty(Filter))

{

string cachedFilter = this.Filter;

this.Filter = "";

this.Filter = cachedFilter;

}

}

// Remove the new item from the original list.

if (e.ListChangedType == ListChangedType.ItemDeleted)

OriginalList.RemoveAt(e.NewIndex);

base.OnListChanged(e);

}

## Determining if the Filter is Valid

The first step in setting the **Filter** property is to check whether the filter value has changed. If the filter value is unchanged, the set operation returns. Next, I check to see whether the filter matches the expected format, and throw an exception if it does not. For readability purposes, I’ve created a method named **BuildRegExForFilterFormat** to build up the regular expression that checks the filter format.

// Build a regular expression to determine if

// filter is in correct format.

public static string BuildRegExForFilterFormat()

{

StringBuilder regex = new StringBuilder();

// Look for optional literal brackets,

// followed by word characters or space.

regex.Append(@"\[?[\w\s]+\]?\s?");

// Add the operators: > < or =.

regex.Append(@"[><=]");

//Add optional space followed by optional quote and

// any character followed by the optional quote.

regex.Append(@"\s?'?.+'?");

return regex.ToString();

}

At this point in the set operation, I temporarily disable change notifications for the list. It is important that I do this step because next I manipulate the contents of the list. In the context of an application, bound controls should not be notified of these internal changes. To disable change notifications, I set the **RaiseListChangedEvents** property to **false**. This property is set back to **true** when the set call is finished.

Now, I split the filter string at “AND” which indicates a multi-column filter. This allows me to loop through each filter part, parse the filter string, and apply it to the data source. Using this approach, filters are applied incrementally and in the same order that they are found in the filter string. Next, I set the filter value and set the **RaiseListChangedEvents** property back to **true**. Finally, I call **OnListChanged** with parameters to signal bound controls to refresh. In the following code, the filter parsing and application is accomplished by two separate methods; **ParseFilter** and **ApplyFilter**. These methods, and their supporting methods, are examined in detail, in the next section.

set

{

if (filterValue == value) return;

// If the value is not null or empty, but doesn't

// match expected format, throw an exception.

if (!string.IsNullOrEmpty(value) &&

!Regex.IsMatch(value,

BuildRegExForFilterFormat(), RegexOptions.Singleline))

throw new ArgumentException("Filter is not in " +

"the format: propName[<>=]'value'.");

//Turn off list-changed events.

RaiseListChangedEvents = false;

// If the value is null or empty, reset list.

if (string.IsNullOrEmpty(value))

ResetList();

else

{

int count = 0;

string[] matches = value.Split(new string[] { " AND " },

StringSplitOptions.RemoveEmptyEntries);

while (count < matches.Length)

{

string filterPart = matches[count].ToString();

// Check to see if the filter was set previously.

// Also, check if current filter is a subset of

// the previous filter.

if (!String.IsNullOrEmpty(filterValue)

&& !value.Contains(filterValue))

ResetList();

// Parse and apply the filter.

SingleFilterInfo filterInfo = ParseFilter(filterPart);

ApplyFilter(filterInfo);

count++;

}

}

// Set the filter value and turn on list changed events.

filterValue = value;

RaiseListChangedEvents = true;

OnListChanged(new ListChangedEventArgs(ListChangedType.Reset, -1));

}

The following is the **ResetList** method. I copy the items from the original list back to my list and reapply the sort, if necessary. For sort implementation details, see the complete code example.

private void ResetList()

{

this.ClearItems();

foreach (T t in originalListValue)

this.Items.Add(t);

if (IsSortedCore)

ApplySortCore(SortPropertyCore, SortDirectionCore);

}

## Parsing the Filter

Parsing the filter can be very labor-intensive, depending on the sophistication of the filtering that is implemented. You must determine the filter operator and the filter property, and then store these values. I chose to store them in a structure named **SingleFilterInfo**. The following code shows you the structure.

public struct SingleFilterInfo

{

internal string PropName;

internal PropertyDescriptor PropDesc;

internal Object CompareValue;

internal FilterOperator OperatorValue;

}

The comparison value is declared as an Object because I do not know this type until I determine the property to filter by. The filter operator is declared as an **FilterOperator** enumeration value. The following enumeration shows the filter operators I accept.

// Enum to hold filter operators. The chars

// are converted to their integer values.

public enum FilterOperator

{

EqualTo = '=',

LessThan = '<',

GreaterThan = '>',

None = ' '

}

With this structure and enumeration in place, I can perform the actual filter parsing. First I declare a **SingleFilterInfo** object and use the **DetermineFilterOperator** method to determine the operator and store this value. Next, I split the filter string and examine the first value. The code retrieves and uses a **TypeConverter** to convert the filter value to the property type. The code throws exceptions if the specified property does not exist or the type conversion cannot be performed.

internal SingleFilterInfo ParseFilter(string filterPart)

{

SingleFilterInfo filterInfo = new SingleFilterInfo();

filterInfo.OperatorValue = DetermineFilterOperator(filterPart);

string[] filterStringParts =

filterPart.Split(new char[] {

(char)filterInfo.OperatorValue });

filterInfo.PropName =

filterStringParts[0].Replace("[", "").

Replace("]", "").Replace(" AND ", "").Trim();

// Get the property descriptor for the filter property name.

PropertyDescriptor filterPropDesc =

TypeDescriptor.GetProperties(typeof(T))[filterInfo.PropName];

// Convert the filter compare value to the property type.

if (filterPropDesc == null)

throw new InvalidOperationException("Specified property to " +

"filter " + filterInfo.PropName +

" on does not exist on type: " + typeof(T).Name);

filterInfo.PropDesc = filterPropDesc;

string comparePartNoQuotes = StripOffQuotes(filterStringParts[1]);

try

{

TypeConverter converter =

TypeDescriptor.GetConverter(filterPropDesc.PropertyType);

filterInfo.CompareValue =

converter.ConvertFromString(comparePartNoQuotes);

}

catch (NotSupportedException)

{

throw new InvalidOperationException("Specified filter" +

"value " + comparePartNoQuotes + " can not be converted" +

"from string. Implement a type converter for " +

filterPropDesc.PropertyType.ToString());

}

return filterInfo;

}

The following are the **DetermineFilterOperator** and **StripOffQuotes** methods that are used by **ParseFilter**.

internal FilterOperator DetermineFilterOperator(string filterPart)

{

// Determine the filter's operator.

if (Regex.IsMatch(filterPart, "[^>^<]="))

return FilterOperator.EqualTo;

else if (Regex.IsMatch(filterPart, "<[^>^=]"))

return FilterOperator.LessThan;

else if (Regex.IsMatch(filterPart, "[^<]>[^=]"))

return FilterOperator.GreaterThan;

else

return FilterOperator.None;

}

internal static string StripOffQuotes(string filterPart)

{

// Strip off quotes in compare value if they are present.

if (Regex.IsMatch(filterPart, "'.+'"))

{

int quote = filterPart.IndexOf('\'');

filterPart = filterPart.Remove(quote, 1);

quote = filterPart.LastIndexOf('\'');

filterPart = filterPart.Remove(quote, 1);

filterPart = filterPart.Trim();

}

return filterPart;

}

## Applying the Filter

Next, I apply the filter by passing the **SingleFilterInfo** object I built in **ParseFilter** to the **ApplyFilter** method. First, I check to see whether the specified property type implements the **IComparable** interface. The filter mechanism relies on the **Compare** method of this interface and the code throws an exception if the property does not implement this interface. Depending on the operator, I use the **Compare** method to populate the results list. Finally, I clear the original list and copy the results list to this list.

internal void ApplyFilter(SingleFilterInfo filterParts)

{

List<T> results;

// Check to see if the property type we are filtering by implements

// the IComparable interface.

Type interfaceType =

TypeDescriptor.GetProperties(typeof(T))[filterParts.PropName]

.PropertyType.GetInterface("IComparable");

if (interfaceType == null)

throw new InvalidOperationException("Filtered property" +

" must implement IComparable.");

results = new List<T>();

// Check each value and add to the results list.

foreach (T item in this)

{

if (filterParts.PropDesc.GetValue(item) != null)

{

IComparable compareValue =

filterParts.PropDesc.GetValue(item) as IComparable;

int result =

compareValue.CompareTo(filterParts.CompareValue);

if (filterParts.OperatorValue ==

FilterOperator.EqualTo && result == 0)

results.Add(item);

if (filterParts.OperatorValue ==

FilterOperator.GreaterThan && result > 0)

results.Add(item);

if (filterParts.OperatorValue ==

FilterOperator.LessThan && result < 0)

results.Add(item);

}

}

this.ClearItems();

foreach (T itemFound in results)

this.Add(itemFound);

}

# Implementing the RemoveFilter Method

The final step is to implement the **RemoveFilter** method. This implementation just checks whether the **Filter** property is equal to **null**. If **Filter** is not **null**, **Filter** is set to **null**. Setting **Filter** to **null** calls the set implementation of **Filter**, which clears the filtered items and copies the original list contents back into the list.

public void RemoveFilter()

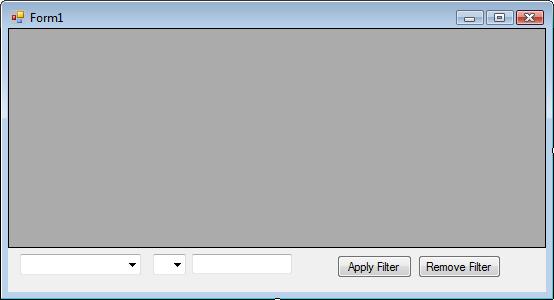
{

if (Filter != null) Filter = null;

}

# Using the Filtered List

To use the filtered list, I created a new Windows Forms application. In my example, the filtered list and application are located in the same namespace and assembly. Depending on how you have configured your application, you may have to add an assembly reference and a using statement for the namespace that contains the **FilteredListView** class. Using the Windows Forms Designer, I added a **BindingSource**, two **Button** controls, two **ComboBox** controls, a **DataGridView** and a **TextBox** to the form. I left the names set to the default; *dataGridView1*, *bindingSource1*, and so on. I set the **Dock** property of the **DataGridView** to **Top** and I set the **Text** property of *button1* and *button2* to “Apply Filter” and “Remove Filter”, respectively. When finished, the form resembles the following illustration.



## Adding a Business Object

The following code is a simple **Employee** object I use to demonstrate the filtered list. If you want to use this business object, copy it to the form, after the form class. Otherwise, you will have to reference the assembly that contains your business object.

public class Employee

{

private string lastNameValue;

private string firstNameValue;

private int salaryValue;

private DateTime startDateValue;

public Employee() {

LastName = "Last Name";

FirstName = "First Name";

Salary = 40000;

StartDate = DateTime.Today;

}

public Employee(string lastName, string firstName,

int salary, DateTime startDate)

{

LastName = lastName;

FirstName = firstName;

Salary = salary;

StartDate = startDate;

}

public string LastName

{

get { return lastNameValue; }

set { lastNameValue = value; }

}

public string FirstName

{

get { return firstNameValue; }

set { firstNameValue = value; }

}

public int Salary

{

get { return salaryValue; }

set { salaryValue = value; }

}

public DateTime StartDate

{

get { return startDateValue; }

set { startDateValue = value; }

}

## Establishing the Data Source

Now that I have a business object, the next step is to set up the data source. The following code is a list declaration and the form’s constructor. This constructor populates the list, and sets the data source for the binding source and other controls. If I want to use a different business object, I simply replace the code that adds **Employee** objects to the list with code that populates the list with a different business object.

FilteredBindingList<Employee> businessObjects;

public Form1()

{

InitializeComponent();

businessObjects = new FilteredBindingList<Employee>();

businessObjects.Add(new Employee("Haas", "Jonathan", 35000,

new DateTime(2005, 12, 12)));

businessObjects.Add(new Employee("Adams", "Ellen", 55000,

new DateTime(2004, 1, 11)));

businessObjects.Add(new Employee("Hanif", "Kerim", 45000,

new DateTime(2003, 12, 4)));

businessObjects.Add(new Employee("Akers", "Kim", 35600,

new DateTime(2002, 12, 12)));

businessObjects.Add(new Employee("Harris", "Phyllis", 60000,

new DateTime(2004, 10, 10)));

businessObjects.Add(new Employee("Andersen", "Elizabeth", 65000,

new DateTime(2006, 4, 4)));

businessObjects.Add(new Employee("Alberts", "Amy", 35000,

new DateTime(2007, 2, 1)));

businessObjects.Add(new Employee("Hamlin", "Jay", 38000,

new DateTime(2004, 8, 8)));

businessObjects.Add(new Employee("Hee", "Gordon", 50000,

new DateTime(2004, 10, 12)));

businessObjects.Add(new Employee("Penor", "Lori", 40000,

new DateTime(2006, 12, 20)));

businessObjects.Add(new Employee("Pfeiffer", "Michael", 49000,

new DateTime(2002, 1, 29)));

businessObjects.Add(new Employee("Perry", "Brian", 54000,

new DateTime(2005, 9, 25)));

businessObjects.Add(new Employee("Philips", "Carol", 65000,

new DateTime(2005, 8, 14)));

businessObjects.Add(new Employee("Pica", "Guido", 52000,

new DateTime(2006, 6, 23)));

businessObjects.Add(new Employee("Jean", "Virginie", 38000,

new DateTime(2002, 5, 2)));

businessObjects.Add(new Employee("Reiter", "Tsvi", 39000,

new DateTime(2003, 4, 12)));

bindingSource1.DataSource = businessObjects;

dataGridView1.DataSource = bindingSource1;

PropertyDescriptorCollection objectProps =

TypeDescriptor.GetProperties(businessObjects[0]);

BindingSource bindingSource2 = new BindingSource();

foreach (PropertyDescriptor prop in objectProps)

{

if (prop.PropertyType.GetInterface("IComparable", true)

!= null)

bindingSource2.Add(prop.Name);

}

comboBox1.DataSource = bindingSource2;

comboBox2.Items.AddRange(new string[] { "<", "=", ">" });

comboBox2.SelectedIndex = 0;

}

## Setting the Filter and Removing It

The final step is to handle the **Click** events for the Apply Filter and Remove Filter button controls. To set the filter, I built a filter string out of the values selected for the combo boxes and the text value entered for the text box. Setting the **Filter** property on the binding source sets the filter on the underlying list. To remove the filter, I simply call the **RemoveFilter** method exposed by the **BindingSource**. This method in turn, calls the **RemoveFilter** method on the underlying list.

private void button1\_Click(object sender, EventArgs e)

{

try

{

bindingSource1.Filter = GetFilterString();

}

catch (ArgumentException ex)

{

MessageBox.Show(ex.Message +

" Try another filter expression.");

}

}

public string GetFilterString()

{

StringBuilder query = new StringBuilder();

if (comboBox1.SelectedItem != null)

query.Append((string)comboBox1.SelectedItem);

else

query.Append(comboBox1.Text);

if (comboBox2.SelectedItem != null)

query.Append((string)comboBox2.SelectedItem);

else

query.Append(comboBox2.Text);

query.Append("'" + textBox1.Text + "'");

return query.ToString();

}

private void button2\_Click(object sender, EventArgs e)

{

bindingSource1.RemoveFilter();

}

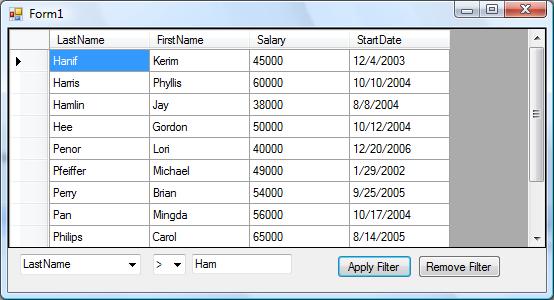
I added the following two lines to the end of the form’s constructor to associate the events with their event-handling methods.

this.button1.Click += new System.EventHandler(this.button1\_Click);

this.button2.Click += new System.EventHandler(this.button2\_Click);

## Running the Application

Now I compile and run the application. When the application is running, I can select values in the combo boxes, enter a filter value in the text box, and click the Apply Filter button to see the **DataGridView** refresh with the filtered contents. In addition, I can sort the grid contents before or after I apply a filter. The following illustration shows a list sorted by last name and a filter applied. The filter is *LastName > Ham*.



# Next Steps

I’ve shown you a simple application that uses my filtering implementation, but there are various ways that the application could be enhanced. For example, I may want to add controls that interact with the search functionality.

The filtering algorithm that is used in this sample depends on a type implementing the **IComparable** interface. If I sort or filter by a property that does not implement the **IComparable** interface, an exception will be thrown. This logic works well if I am filtering custom business objects whose properties are mainly simple types. To take the functionality in this sample a step further, I could consider implementing the **IComparable** interface and a **TypeConverter** for any properties of my custom business object that are complex types.

Also, as stated earlier, this code was designed to work with the **DataGridView** auto filter feature shown in [Building a Drop-Down Filter List for a DataGridView Column Header Cell](http://go.microsoft.com/fwlink/?LinkId=105803). Using this code, I have a data source that is capable of interacting with this popular Excel-like feature. You can see this interaction by downloading the sample solution, which includes the filtered binding list and accompanying unit tests, the employee object, the sample form shown in this article as well as a Windows Form that uses the auto-filtering **DataGridView**.

Alternatively, if my application does not require a generic filtering solution and I am using the .NET Framework 3.5, I can use LINQ on the client-side to perform complex filtering on a list of custom objects.

# Conclusion

In this article, I showed you one way to implement filtering. I created a filtered class that extended the generic binding list and implemented the **IBindingListView** interface. Next, I implemented the **SupportsFiltering**, **Filter**, and **RemoveFilter** members. To use the filter, I set the **Filter** property on the binding source. To test the filter, I created a simple Windows Forms application. I also suggested ways that the filtering implementation and the corresponding application could be improved. Now, you can use this filtering implementation and unit tests to start building your own custom filtering solution.