Overview

Data Loading using a DataSet

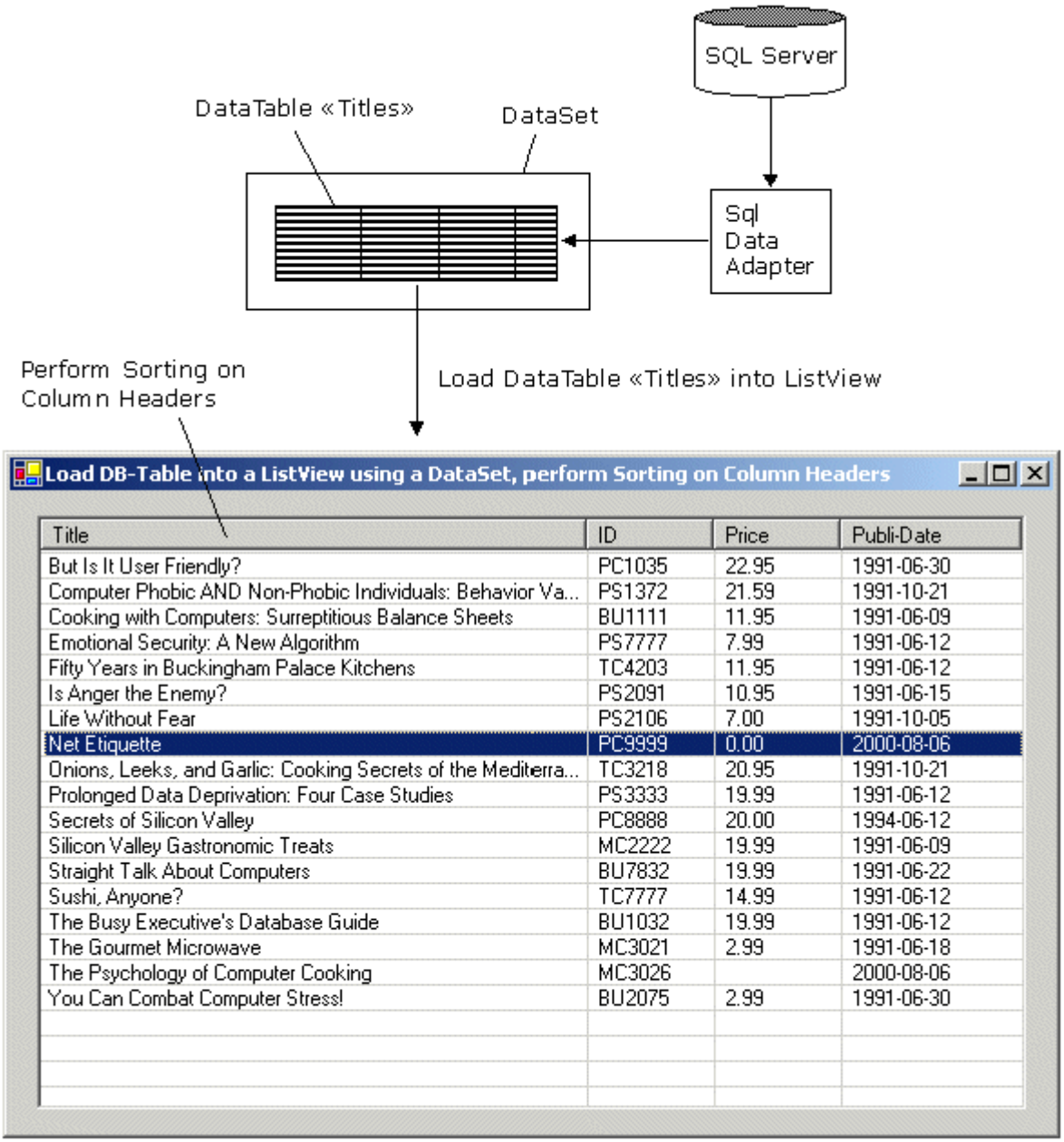
This article will show how to fill a **ListView** Control with the data loaded into a **DataSet**. You may use a DataSet bind it to a Grid Control to show the output of a query, but data binding of controls is not always the ideal method of accessing the data (You may encounter problems with the DataBinding). A DataSet maintains a copy of the entire resultset in the client systems memory in case you need to make changes to a row. Instead of using a bound grid and a DataSet, we can use the listview control with the view set to details mode and fill it with the data from a DataSet.

Sorting the ListView

When you are working with the ListView control, you may want to sort its contents based on a specific column. An example of this kind of functionality occurs in a Windows Explorer program when you view the contents of a folder on your hard disk. In Details view, Windows Explorer displays information about the files in that folder. For example, you see the file name, the file size, the file type, and the date that the file was modified. When you click one of the column headers, the list is sorted in ascending order based on that column. When you click the same column header again, the column is sorted in descending order.

The ListView Control

The example in this article defines a class that inherits from the **IComparer** interface. Additionally, this example uses the **Compare method** of the **CaseInsenstiveComparer** class to perform the actual comparison of the items. Note that this method of comparison is **not case sensitive** ("Apple" is considered to be the same as "apple"). Also, note that all of the columns in this example are **sorted in a "text" manner.**



Title	ID	Price	Publi-Date
But Is It User Friendly?	PC1035	22.95	1991-06-30
Computer Phobic AND Non-Phobic Individuals: Behavior Va...	PS1372	21.59	1991-10-21
Cooking with Computers: Surreptitious Balance Sheets	BU1111	11.95	1991-06-09
Emotional Security: A New Algorithm	PS7777	7.99	1991-06-12
Fifty Years in Buckingham Palace Kitchens	TC4203	11.95	1991-06-12
Is Anger the Enemy?	PS2091	10.95	1991-06-15
Life Without Fear	PS2106	7.00	1991-10-05
Net Etiquette	PC9999	0.00	2000-08-06
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterra...	TC3218	20.95	1991-10-21
Prolonged Data Deprivation: Four Case Studies	PS3333	19.99	1991-06-12
Secrets of Silicon Valley	PC8888	20.00	1994-06-12
Silicon Valley Gastronomic Treats	MC2222	19.99	1991-06-09
Straight Talk About Computers	BU7832	19.99	1991-06-22
Sushi, Anyone?	TC7777	14.99	1991-06-12
The Busy Executive's Database Guide	BU1032	19.99	1991-06-12
The Gourmet Microwave	MC3021	2.99	1991-06-18
The Psychology of Computer Cooking	MC3026		2000-08-06
You Can Combat Computer Stress!	BU2075	2.99	1991-06-30

Introduction

The ListView control is a great way to display file system information and data from an XML file or database. The ListView control is typically used to display a graphical icon that represents the item, as well as the item text. In addition, the ListView control can be used to display additional information about an item in a **subitem**. For example, if the ListView control is displaying a list of files, you can configure the ListView control to display details such as file size and attributes as subitems. To display subitem information in the ListView control, you must set the View property to **View.Details**. In addition, you must create **ColumnHeader** objects and assign them to the Columns property of the ListView control. Once these properties are set, items are displayed in a row and column format that is similar to a DataGrid control. The ability to display items in this way makes the ListView control a quick and easy solution for displaying data from any type of data source.

Sorting for the ListView control is provided by using the **Sorting property** of the ListView. This enables you define the type of sorting to apply to the items. This is a great feature if you want to sort only by items. If you want to sort by subitems, you must use the **custom sorting features of the ListView control**. This article will demonstrate how to perform custom sorting in the ListView control and how to handle special data-type conditions when sorting.

Custom Sorting Features of the ListView Control

The ListView control provides features that enable you to use sorting other than that provided by the Sorting property. When the ListView control sorts items using the Sorting property, it uses a class that implements the **System.Collections.IComparer** interface. This class provides the sorting features

used to sort each item. In order to sort by subitems, you must create your own class that implements the **IComparer** interface that in turn implements the sorting your ListView control needs. The class is defined with a constructor that specifies the column by which the ListView control is sorted. Once you have created this class, typically as a nested class of your form, you create an instance of this class and assign it to the **ListViewItemSorter** property of the ListView. This identifies the custom sorting class that the ListView control will use when the **Sort method** is called. The Sort method performs the actual sorting of the ListView items.

Initializing the Control

To begin, create an instance of a ListView control and add it to a form. After the control is on the form, add items to the ListView control using the Items property. You can add as many items as you want; just be sure that each **item's text is unique**. While you are creating the items, add **subitems** for each. The following table is an example of how this information might look in the ListView control.

Item	Subitem 1	Subitem 2
...	...	...5
...	...	...
...	...	...

```
// Initialize ListView
private void InitializeListView()
{
    // Set the view to show details.
    listView1.View = View.Details;

    // Allow the user to edit item text.
    listView1.LabelEdit = true;

    // Allow the user to rearrange columns.
    listView1.AllowColumnReorder = true;

    // Select the item and subitems when selection is made.
    listView1.FullRowSelect = true;

    // Display grid lines.
    listView1.GridLines = true;

    // Sort the items in the list in ascending order.
    listView1.Sorting = SortOrder.Ascending;

    // Attach Subitems to the ListView
    listView1.Columns.Add("Title", 300, HorizontalAlignment.Left);
    listView1.Columns.Add("ID", 70, HorizontalAlignment.Left);
    listView1.Columns.Add("Price", 70, HorizontalAlignment.Left);
    listView1.Columns.Add("Publi-Date", 100, HorizontalAlignment.Left);

    // The ListViewItemSorter property allows you to specify the
    // object that performs the sorting of items in the ListView.
    // You can use the ListViewItemSorter property in combination
    // with the Sort method to perform custom sorting.
    _lvwItemComparer = new ListViewItemComparer();
    this.listView1.ListViewItemSorter = _lvwItemComparer;
}
```

Loading the ListView with Data from a DataSet

In this example we use a DataSet to load the "Titles" DataTable, which was filled with the Database Table "Titles" in the Pub Database on SQL-Server 2000.

```
// Load Data from the DataSet into the ListView
private void LoadList()
{
    // Get the table from the data set
    DataTable dtable = _DataSet.Tables["Titles"];

    // Clear the ListView control
    listView1.Items.Clear();

    // Display items in the ListView control
```

```
        for (int i = 0; i < dtable.Rows.Count; i++)
        {
            DataRow drow = dtable.Rows[i];

            // Only row that have not been deleted
            if (drow.RowState != DataRowState.Deleted)
            {
                // Define the list items
                ListViewItem lvi = new ListViewItem(drow["title"].ToString());
                lvi.SubItems.Add (drow["title_id"].ToString());
                lvi.SubItems.Add (drow["price"].ToString());
                lvi.SubItems.Add (drow["pubdate"].ToString());

                // Add the list items to the ListView
                listView1.Items.Add(lvi);
            }
        }
    }
```

Handling the ColumnClick Event

In order to determine which set of subitems to sort by, you need to know when the user clicks a column heading for a subitem. To do this, you need to create an event-handling method for the **ColumnClick** event of the **ListView**. Place the event-handling method as a member of your form and ensure that it contains a signature similar to the one shown in the following code example.

```
// Perform Sorting on Column Headers
private void listView1_ColumnClick(
    object sender,
    System.Windows.Forms.ColumnClickEventArgs e)
{

    // Determine if clicked column is already the column that is being sorted.
    if (e.Column == _lvwItemComparer.SortColumn)
    {
        // Reverse the current sort direction for this column.
        if (_lvwItemComparer.Order == SortOrder.Ascending)
        {
            _lvwItemComparer.Order = SortOrder.Descending;
        }
        else
        {
            _lvwItemComparer.Order = SortOrder.Ascending;
        }
    }
    else
    {
        // Set the column number that is to be sorted; default to ascending.
        _lvwItemComparer.SortColumn = e.Column;
        _lvwItemComparer.Order = SortOrder.Ascending;
    }

    // Perform the sort with these new sort options.
    this.listView1.Sort();
}
```

Connect the event-handling method to the **ListView** control by adding code to the constructor of your form, as shown in the following example.

```
this.listView1.ColumnClick +=
  new System.Windows.Forms.ColumnClickEventHandler(
  this.listView1_ColumnClick);
```

Perform custom Sorting

Case Insenstive Sorting

The sorting is performed in a required method of the **IComparer** interface called **Compare**. This method takes two objects as parameters, which will contain the two items being compared. When the **Sort** method is called in the **ColumnClick** event-handling method of the ListView control, the Sort

method uses the **ListViewItemComparer** object that was defined and assigned to the **ListViewItemSorter property** and calls its Compare method.

In this example the ListViewItemComparer class uses the **Compare** method of the **CaseInsenstiveComparer** class to perform the actual comparison of the items. Note that this method of comparison is not case sensitive ("Apple" is considered to be the same as "apple"). Also, note that all of the columns in this example are sorted in a "text" manner.

The **Compare** method of the **CaseInsenstiveComparer** performs a case-insensitive comparison of two objects of the same type and returns a value indicating whether one is less than, equal to or greater than the other.

Add the following class definition to your Form class and ensure that it is nested properly inside your form.

```
// This class is an implementation of the 'IComparer' interface.
public class ListViewItemComparer : IComparer
{
    // Specifies the column to be sorted
    private int ColumnToSort;

    // Specifies the order in which to sort (i.e. 'Ascending').
    private SortOrder OrderOfSort;

    // Case insensitive comparer object
    private CaseInsensitiveComparer ObjectCompare;

    // Class constructor, initializes various elements
    public ListViewItemComparer()
    {
        // Initialize the column to '0'
        ColumnToSort = 0;

        // Initialize the sort order to 'none'
        OrderOfSort = SortOrder.None;

        // Initialize the CaseInsensitiveComparer object
        ObjectCompare = new CaseInsensitiveComparer();
    }

    // This method is inherited from the IComparer interface.
    // It compares the two objects passed using a case
    // insensitive comparison.
    //
    // x: First object to be compared
    // y: Second object to be compared
    //
    // The result of the comparison. "0" if equal,
    // negative if 'x' is less than 'y' and
    // positive if 'x' is greater than 'y'
    public int Compare(object x, object y)
    {
        int compareResult;
        ListViewItem listviewX, listviewY;

        // Cast the objects to be compared to ListViewItem objects
        listviewX = (ListViewItem)x;
        listviewY = (ListViewItem)y;

        // Case insensitive Compare
        compareResult = ObjectCompare.Compare (
            listviewX.SubItems[ColumnToSort].Text,
            listviewY.SubItems[ColumnToSort].Text
        );

        // Calculate correct return value based on object comparison
        if (OrderOfSort == SortOrder.Ascending)
        {
            // Ascending sort is selected, return normal result of compare operation
            return compareResult;
        }
```

```
        else if (OrderOfSort == SortOrder.Descending)
        {
            // Descending sort is selected, return negative result of compare operation
            return (-compareResult);
        }
        else
        {
            // Return '0' to indicate they are equal
            return 0;
        }
    }

    // Gets or sets the number of the column to which to
    // apply the sorting operation (Defaults to '0').
    public int SortColumn
    {
        set
        {
            ColumnToSort = value;
        }
        get
        {
            return ColumnToSort;
        }
    }

    // Gets or sets the order of sorting to apply
    // (for example, 'Ascending' or 'Descending').
    public SortOrder Order
    {
        set
        {
            OrderOfSort = value;
        }
        get
        {
            return OrderOfSort;
        }
    }
}
```

Simple String Sorting

Another approach is to use the **String.Compare** method.

When the ListViewItemComparer object is created, it is assigned the index of the column that was clicked. This column index is used to access subitems from the column that needs to be sorted. The subitems are then passed to the String.Compare method, which compares the items and returns one of three results. If the item in the x parameter is less than the item in the y parameter, a value less than zero is returned. If the items are identical, a zero is returned. Finally, if the item in the x parameter is greater than the item in the y parameter, a value greater than zero is returned.

```
public int Compare(object x, object y)
{
    int compareResult;
    ListViewItem listviewX, listviewY;

    // Cast the objects to be compared to ListViewItem objects
    listviewX = (ListViewItem)x;
    listviewY = (ListViewItem)y;

    // Simple String Compare
    compareResult = String.Compare (
        listviewX.SubItems[ColumnToSort].Text,
        listviewY.SubItems[ColumnToSort].Text
    );

    // Calculate correct return value based on object comparison
    if (OrderOfSort == SortOrder.Ascending)
    {
        // Ascending sort is selected, return normal result of compare operation
```

```
        return compareResult;
    }
    else if (OrderOfSort == SortOrder.Descending)
    {
        // Descending sort is selected, return negative result of compare operation
        return (-compareResult);
    }
    else
    {
        // Return '0' to indicate they are equal
        return 0;
    }
}
```

Sorting Dates

Data that is placed into the ListView control as an item is **displayed as text and stored as text**. This makes it easy to sort using the String.Compare method in an IComparer class. String.Compare sorts both alphabetical characters and numbers. However, certain data types do not sort correctly using String.Compare, such as date and time information. For this reason, the **System.DateTime** structure has a Compare method just as the String class does. This method can be used to perform the same type of sorting based on chronological order. In this section, you modify only the Compare method to allow for dates to be sorted properly.

```
public int Compare(object x, object y)
{
    int compareResult;
    ListViewItem listviewX, listviewY;

    // Cast the objects to be compared to ListViewItem objects
    listviewX = (ListViewItem)x;
    listviewY = (ListViewItem)y;

    // Determine whether the type being compared is a date type.
    try
    {
        // Parse the two objects passed as a parameter as a DateTime.
        System.DateTime firstDate =
            DateTime.Parse(listviewX.SubItems[ColumnToSort].Text);
        System.DateTime secondDate =
            DateTime.Parse(listviewY.SubItems[ColumnToSort].Text);

        // Compare the two dates.
        compareResult = DateTime.Compare(firstDate, secondDate);
    }

    // If neither compared object has a valid date format,
    // perform a Case Insensitive Sort
    catch
    {
        // Case Insensitive Compare
        compareResult = ObjectCompare.Compare (
            listviewX.SubItems[ColumnToSort].Text,
            listviewY.SubItems[ColumnToSort].Text
        );
    }

    // Calculate correct return value based on object comparison
    if (OrderOfSort == SortOrder.Ascending)
    {
        // Ascending sort is selected, return normal result of compare operation
        return compareResult;
    }
    else if (OrderOfSort == SortOrder.Descending)
    {
        // Descending sort is selected, return negative result of compare operation
        return (-compareResult);
    }
    else
    {
        // Return '0' to indicate they are equal
```

```
        return 0;
    }
}
```

The **Compare** method starts by casting the `x` and `y` parameters to **DateTime** objects. This extraction is performed in a try/catch block to catch exceptions that might occur by forcing the casting of the two items being compared into **DateTime** objects. If an exception does occur, it signals to the code that the type being converted is not a valid date or time and can be sorted by the **String.Compare** method. If the two types are dates, they are sorted using the **DateTime.Compare** method.

Conclusion

The **ListView** control can provide the ability to display data in a number of ways. It can be used to display single items as well as items that contain subitem information. Using the sorting features provided by the **ListView** control, you can also enable users to sort items in the **ListView** control based on those subitems, regardless of the type of data being presented. This ability to sort items and their subitems enables your application to behave in ways that are familiar to users of Microsoft® Windows® Explorer and other applications that provide a **ListView** display of data and the ability to sort its contents.