

A Practical Guide to .NET DataTables, DataSets and DataGrids - Part 1

<http://www.codeproject.com/Articles/6178/A-Practical-Guide-to-NET-DataTables-DataSets-and-D>

Contents

A Practical Guide to .NET DataTables, DataSets and DataGrids - Part 1.....	1
Table of Contents	4
1 Introduction	6
2 Overview	6
A Practical Guide to .NET DataTables, DataSets and DataGrids - Part 2.....	8
3 Tables	8
3.1 Table creation	9
3.2 Columns – Creating and Adding to Tables	10
3.3 Deleting/Removing Columns.....	13
3.4 Modifying Column Properties	13
3.5 Clearing Column Collection	13
3.6 Cloning a Table	14
3.7 Rows – creating and adding to a table.....	14
Method 1	15
Method 2	15
Method 3	15
Method 4	15
3.8 Modifying data within an existing table row	16
3.9 Fill Table using LoadDataRow() method.....	17
3.10 Retrieving Table Content.....	18
3.11 Row Versions and Accepting/Rejecting Changes	19
3.11.1 Methods and Enumerations.....	19
3.11.2 Sample 1 – Row States	21
3.11.3 Sample 2 – Initial Loading of Table.....	22

Current Version	22
Default Version	22
Original Version	23
Proposed Version	23
3.11.4 Sample 3 – DataRow AcceptChanges.....	23
3.11.5 Sample 4 – Table AcceptChanges.....	24
3.11.6 Sample 5 – DataRow BeginEdit	25
3.11.7 Sample 6 – DataRow CancelEdit	26
3.11.8 Sample 7 – DataRow BeginEdit – Example 2	27
3.11.9 Sample 8 – DataRow Change values – Example 2	28
3.11.10 Sample 9 – DataRow EndEdit – Modified Rows.....	29
3.11.11 Sample 10 – DataRow AcceptChanges of Modified Rows	30
3.11.12 Sample 11 – DataRow RejectChanges.....	31
3.11.13 Sample 12 – LoadDataRow without table having primary key	32
3.11.14 Sample 13 – LoadDataRow with table having primary key	34
3.11.15 Sample Code for Obtaining Version and State Information	35
3.12Handling DataTable Errors	36
3.13DataTable Events	36
A Practical Guide to .NET DataTables, DataSets and DataGrids - Part 3.....	38
4 Data Sets.....	38
4.1 DataSet Methods.....	38
4.2 DataSet Properties.....	38
4.3 Loading A DataSet.....	40
4.3.1 From a Table	40
4.3.2 From a Database	40
4.4 Linked Tables	42
4.5 Linked tables in a dataset.....	43
4.5.1 Filling	43
4.5.2 Removing	44
4.6 XML Export and Import DataSet Data	45
4.6.1 WriteXml	45

4.6.2 ReadXml	45
4.7 Handling DataSet Errors	45
4.8 Updating Database with DataSet/DataTable changes	46
A Practical Guide to .NET DataTables, DataSets and DataGrids - Part 4.....	47
5 Data Grid.....	47
5.1 Methods and Properties	47
5.2 Assigning Data Sources	48
5.3 Formatting.....	49
5.3.1 DataGridTableStyle - WinForms	49
5.3.2 Change only one table's DataGridTableStyles.....	50
5.3.3 Change a single column in a Table's DataGridTableStyle.....	51
5.3.4 Create a DataGridTableStyle for Tables Based upon its Data	52
5.3.5 Defining DataGridTableStyles through Visual Studio .NET IDE.....	55
5.4 Navigation	57
5.4.1 CurrencyManager	57
5.4.2 CurrentCell.....	57
5.4.3 Selecting Rows.....	57
5.4.4 Expand and Collapse linked tables	58
5.5 Copy DataGrid to the Clipboard.....	59
5.5.1 Copy selected table in DataGrid to clipboard	59
5.5.2 Copy all tables in DataGrid to clipboard	60
5.5.3 Format table data into a string.....	61
5.6 Exporting to a Tabbed delimited Text File	61
5.7 Cloning Table contained in a DataGrid	63
5.8 Comparison of WinForms & WebForms	64
5.8.1 WinForms	64
5.8.2 WebForms	65
6 References	72

Table of Contents

- 1 Introduction
- 2 Overview
- **3 Tables**
- 3.1 Table creation
- 3.2 Columns – Creating and Adding to Tables
- 3.3 Deleting/Removing Columns
- 3.4 Modifying Column Properties
- 3.5 Clearing Column Collection
- 3.6 Cloning a Table
- 3.7 Rows – creating and adding to a table
- 3.8 Modifying data within an existing table row
- 3.9 Fill Table using LoadDataRow() method
- 3.10 Retrieving Table Content
- 3.11 Row Versions and Accepting/Rejecting Changes
- 3.11.1 Methods and Enumerations
- 3.11.2 Sample 1 – Row States
- 3.11.3 Sample 2 – Initial Loading of Table
- 3.11.4 Sample 3 – DataRow AcceptChanges
- 3.11.5 Sample 4 – Table AcceptChanges
- 3.11.6 Sample 5 – DataRow BeginEdit
- 3.11.7 Sample 6 – DataRow CancelEdit
- 3.11.8 Sample 7 – DataRow BeginEdit – Example
- 3.11.9 Sample 8 – DataRow Change values – Example
- 3.11.10 Sample 9 – DataRow EndEdit – Modified Rows
- 3.11.11 Sample 10 – DataRow AcceptChanges of Modified Rows
- 3.11.12 Sample 11 – DataRow RejectChanges
- 3.11.13 Sample 12 – LoadDataRow without table having primary key
- 3.11.14 Sample 13 – LoadDataRow with table having primary key
- 3.11.15 Sample Code for Obtaining Version and State Information
- 3.12 Handling DataTable Errors
- 3.13 DataTable Events
- **4 Data Sets**
- 4.1 DataSet Methods
- 4.2 DataSet Properties
- 4.3 Loading A DataSet
- 4.3.1 From a Table
- 4.3.2 From a Database
- 4.3.2.1 Method 1 – sqlDataAdapter
- 4.3.2.2 Method 2 – sqlDataReader
- 4.4 Linked Tables
- 4.5 Linked tables in a dataset
- 4.5.1 Filling
- 4.5.2 Removing

- 4.6 XML Export and Import DataSet Data
- 4.6.1 WriteXml
- 4.6.2 ReadXml
- 4.7 Handling DataSet Errors
- 4.8 Updating Database with DataSet/DataTable changes
- **5 Data Grid**
- 5.1 Methods and Properties
- 5.2 Assigning Data Sources
- 5.3 Formatting
- 5.3.1 DataGridTableStyle - WinForms
- 5.3.2 Change only one table's DataGridTableStyles
- 5.3.3 Change a single column in a Table's DataGridTableStyle
- 5.3.3.1 Method 1 – for-loop through collection
- 5.3.3.2 Method 2 – Direct access using column string name
- 5.3.4 Create a DataGridTableStyle for Tables Based upon its Data
- 5.3.5 Defining DataGridTableStyles through Visual Studio .NET IDE
- 5.4 Navigation
- 5.4.1 CurrencyManager
- 5.4.2 CurrentCell
- 5.4.3 Selecting Rows
- 5.4.4 Expand and Collapse linked tables
- 5.5 Copy DataGrid to the Clipboard
- 5.5.1 Copy selected table in DataGrid to clipboard
- 5.5.2 Copy all tables in DataGrid to clipboard
- 5.5.3 Format table data into a string
- 5.6 Exporting to a Tabbed delimited Text File
- 5.7 Cloning Table contained in a DataGrid
- 5.8 Comparison of WinForms & WebForms
- 5.8.1 WinForms
- 5.8.2 WebForms
- 5.8.2.1 Edit a row
- 5.8.2.2 Cancel editing a row
- 5.8.2.3 Entering a new row
- 5.8.2.4 Updating a row
- 5.8.2.5 Setting Cell Style values
- 6 References

1 Introduction

The purpose of this document is to provide a practical guide to using Microsoft's .NET **DataTables**, **DataSets** and **DataGrid**. Most articles illustrate how to use the **DataGrid** when directly bound to tables within a database and even though this is an excellent way to use the **DataGrid**, it is also able to display and manage programmatically created and linked tables and datasets composed of these tables without being bound to a database. Microsoft's implementation has provided a rich syntax for populating and accessing rows and their cells within tables, for managing collections of tables, columns, rows and table styles and for managing inserts, updates, deletes and events. Microsoft's Visual Studio .NET development environment provides detailed explanations and code examples for the classes, which is excellent for obtaining a quick reference to a method or property, but not for understanding how they all fit together and are used in applications.

This article will present different ways to create and manage bound and unbound tables and datasets and to bind them to **DataGrids** for use by WebForms and WinForms. The different behaviors of the **DataGrid** depending upon whether it is in a WebForm or a WinForm will be presented. In addition, copying **DataGrid** content to the clipboard, importing and exporting in XML and printing will be presented. Techniques for linking **DataGrid** content to features within graphics objects to provide an interactive UI will be discussed in the last section.

The intent of this article is not to be a complete reference for all methods and members of the classes used for building Tables, Datasets and DataGrids, but to illustrate systematically how to build and manage them using their essential methods and properties. The article will also show equivalent ways for working with these entities to illustrate programming flexibility options. Once they are built the tables can be added to a database and/or the content easily extracted for updating existing database tables.

The structure and features of a table created using the **DataTable** class is at the heart of using the **DataGrid**; therefore, it will be presented first. Next the **DataSet** that manages collections of independent and linked tables will be presented followed by the **DataGrid** that displays and provides an interactive UI for its Tables and Datasets. All example code will be written in C# and the periodic table with its elements and isotopes will be used as a model and source of data.

2 Overview

Figure 1 is a summary view of the essential relationships between the **DataGrid**, **DataGridTableStyles**, **DataSets** and **DataTables** and their various methods and properties. This article will delve into the details of each of these components describing how to create them, to fill them with data and how they work together and thereby provide a clearer understanding of this relationship diagram.

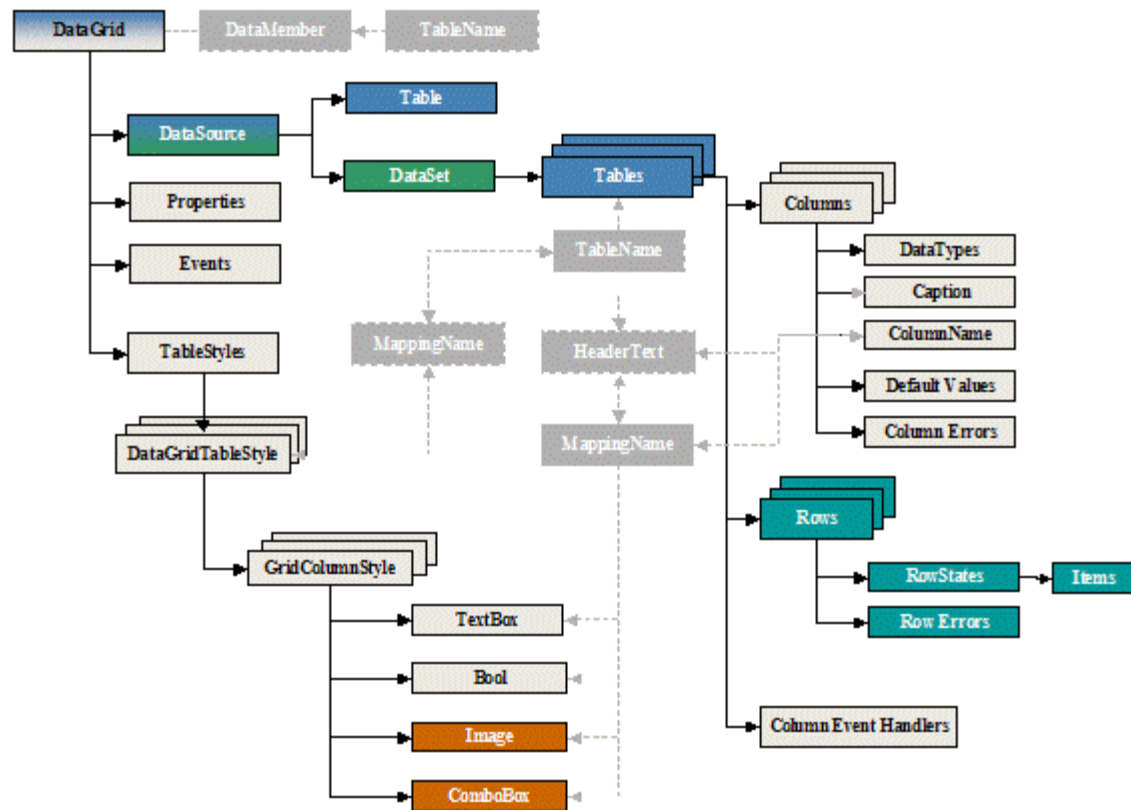


Figure 1 DataGrid, DataSet and DataTable relationship diagram

A Practical Guide to .NET DataTables, DataSets and DataGrids - Part 2

3 Tables

The architecture and capability of Tables should be understood since it carries over to understanding how a **DataSet** and **DataGrid** function. In the process of binding a DataGrid to a database the underlying code creates and associates collections of tables that are filled with data from the database. Also, a DataSet created from a database may contain tables with more information than needs to be displayed, columns may need to be added that are based upon complex formulas using information in other columns and data from multiple databases may need to be combined into a single tabular view. These operations are done by extracting information from these data sources and by filling a programmatically designed table that is unbound.

Fundamentally a table contains **Columns** and **Rows** collections, which means standard methods for accessing and manipulating collections can be used. The **Columns** collection contains, for each column, a name, a data type specification and maybe an assigned default value. Each table row in the Rows collection contains one cell for each column. The table class has an extensive set of methods for editing and managing versions of column and row data and for event notifications when changes occur. Figure 2 illustrates the overall architecture of a table.

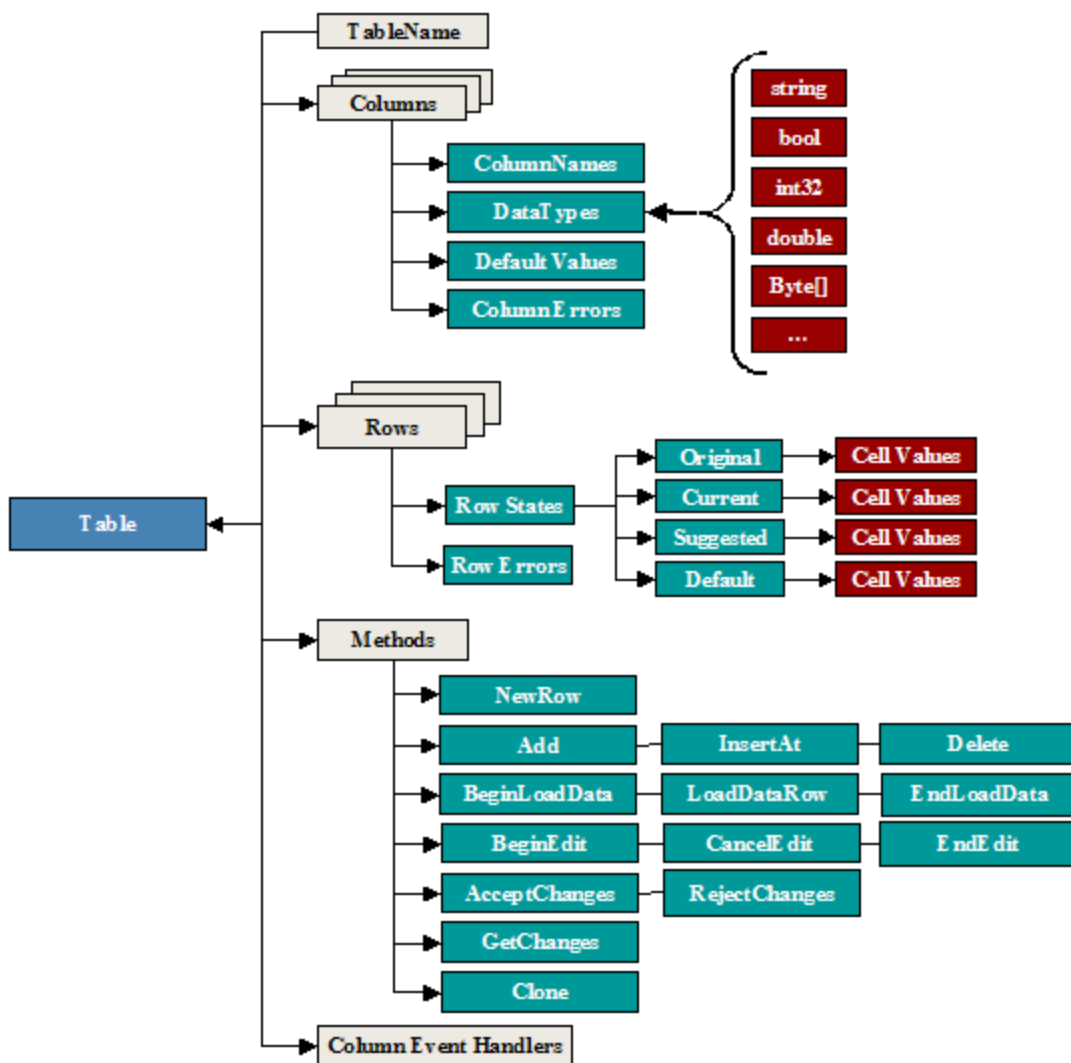


Figure 2 DataTable Decomposed

3.1 Table creation

A table memory object that will be able to contain/manage columns, rows and events can be easily created from the DataTable class as follows:

[Hide](#) [Copy Code](#)

```

// Create a table object by using the DataTable class:

DataTable dt = new DataTable();

// Name the table by assigning a data string containing
// the name to the table's
// TableName property:
  
```

```
dt.TableName = "Elements";  
  
// or use the DataTable(string TableName) constructor  
  
DataTable dt = new DataTable("Elements");
```

3.2 Columns – Creating and Adding to Tables

A table contains a collection of column definitions that will be used to define how each cell within a row can be referenced and the type of data content. The following scenario shows how to define a column and add it to a table's column collection.

- a. Define a Table as described in the Tables section.
- b. Create a column object to be added to the table by using the column class:

Hide Shrink ▲ Copy Code

```
DataColumn dc = new DataColumn();  
  
// Set the properties for the column:  
  
// string name for the column that is used as an index  
// for columns collection  
// and a cell within a row  
dc.ColumnName = "AtomicNbr";  
  
// string name that is used for a column label or header  
// for display purposes  
// if not set, then the default value is dc.ColumnName  
dc.Caption = "Atomic Number";  
  
// one of the standard system data types using the  
// GetType() method.  
dc.DataType = System.Type.GetType("System.Int32");  
  
// or one could use the typeof operator  
dc.DataType = typeof(System.Int32);  
  
// a default value that is assigned each time  
// a new row is created  
dc.DefaultValue = 0;  
  
// or use one of the other constructor's such as  
// DataColumn(string ColumnName, System.Type DataType)  
  
DataColumn dc = new DataColumn("AtomicNbr",  
    System.Type.GetType("System.Int32"));
```

- c. Add the new column to the table **Columns** collection. The order in which the columns are added determines their zero-based index.

```
dt.Columns.Add(dc);
```

d. Repeat b and c for each column to be added to the table **Columns** collection.

```
dc = new DataColumn("Element", System.Type.GetType("System.String"));
dc.DefaultValue = string.Empty;
dc.Caption = "Element";
dt.Columns.Add(dc);

dc = new DataColumn("Symbol", System.Type.GetType("System.String") );
dc.DefaultValue = string.Empty;
dc.Caption = "Symbol";
dt.Columns.Add(dc);

dc = new DataColumn("AtomicMass", System.Type.GetType("System.Decimal") );
dc.DefaultValue = 0.0;
dc.Caption = "Atomic Mass";
dt.Columns.Add(dc);
```

Examples of data types supported in the .NET environment.

Data Type	.NET System Types
Boolean	System.Boolean
Byte	System.Byte
Byte[] (Array)	System.Byte[]
Char (Character)	System.Char
DateTime	System.DateTime
Decimal	System.Decimal
Double	System.Double
Integer	System.Int16, System.Int32, System.Int64
Single	System.Single
String	System.String
Unsigned Integer	System.UInt16, System.UInt32, System.UInt64
TimeSpan	System.TimeSpan

At this point a table called "Elements" has been created with four columns "AtomicNbr", "Element", "Symbol" and "AtomicMass" with their respective data types and default values. The following three **DisplayColumnInfo()** method code examples show this by using different techniques for accessing members and displaying data from their collections. In the first example, a for-loop is used to illustrate accessing table column collections through an integer index while in the second example a **foreach** loop illustrates accessing the same collections using a column class type. The third example uses strings containing the column names as an index. These accessing data examples illustrate the natural syntax approaches for working with collections.

1. **for**-loop

[Hide](#) [Copy Code](#)

```
private void DisplayColumnInfo(DataTable dt)
{
    StringBuilder ColInfo = new StringBuilder();
    ColInfo.AppendFormat("Column\tName\tDataType\n");

    // note that the total number of columns in the
    // collection is contained in the 'Count' property
    for(int j=0; j<dt.Columns.Count; j++)
    {
        ColInfo.AppendFormat(" [{0}]\t{1}\t{2}\t {3}\n", j,
            dt.Columns[j].ColumnName,
            dt.Columns[j].Caption, dt.Columns[j].DataType.ToString());
    }

    MessageBox.Show(ColInfo.ToString() , "Column Name",
        MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
```

2. foreach loop

[Hide](#) [Copy Code](#)

```
private void DisplayColumnInfo(DataTable dt)
{
    StringBuilder ColInfo = new StringBuilder();
    ColInfo.AppendFormat("Column\tName\tDataType\n");
    int j = -1;

    foreach (DataColumn dc in dt.Columns)
    {
        ColInfo.AppendFormat(" [{0}]\t{1}\t{2}\t{3}\n", ++j, dc.ColumnName,
            dc.Caption, dc.DataType.ToString() );
    }

    MessageBox.Show(ColInfo.ToString(),
        "Column Name", MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
```

3. Using known column names as indexes – not column captions!

[Hide](#) [Copy Code](#)

```
private void DisplayColumnDataTypeInfo(DataTa}
```

3. Using known column names as indexes – not column captions!

[Hide](#) [Copy Code](#)

```
private void DisplayColumnDataTypeInfo(DataTable dt)
{
    StringBuilder ColInfo = new StringBuilder();
    ColInfo.AppendFormat("Column\tName\tDataType\n");
```

```

ColInfo.AppendFormat(" [{0}]\t{1}\t{2}\n",1, "AtomicNbr",
dt.Columns["AtomicNbr"].DataType.ToString());

ColInfo.AppendFormat(" [{0}]\t{1}\t{2}\n",1, "Element",
dt.Columns["Element"].DataType.ToString());

ColInfo.AppendFormat(" [{0}]\t{1}\t{2}\n",1, "Symbol",
dt.Columns["Symbol"].DataType.ToString());

ColInfo.AppendFormat(" [{0}]\t{1}\t{2}\n",1, "AtomicMass",
dt.Columns["AtomicMass"].DataType.ToString());

MessageBox.Show(ColInfo.ToString() , "Column Name",
MessageBoxButtons.OK,
MessageBoxIcon.Information);
}

```

3.3 Deleting/Removing Columns

Once a table has been defined columns can be deleted or removed as follows:

[Hide](#) [Copy Code](#)

```

// For example to delete a column "AtomicMass"
dt.Columns.Remove("AtomicMass");
// or using a zero-based index - "AtomicMass" is the
// 4th column with index 3
dt.Columns.RemoveAt(3);
// To make sure that a column can be removed,
// for example, first determine
// whether the column exists, belongs to the table,
// or is involved in a constraint
// or relation.
if (dt.Columns.Contains("AtomicMass"))
    if (dt.Columns.CanRemove(dt.Columns["AtomicMass"]))
    {
        dt.Columns.Remove("AtomicMass");
    }

```

3.4 Modifying Column Properties

Modifying a column property is simply accessing the property and setting its new value. For example:

[Hide](#) [Copy Code](#)

```

dt.Columns["AtomicNbr"].ColumnName = "AtomicNumber";
dt.Columns["AtomicMass"].DataType = typeof(System.float);

```

3.5 Clearing Column Collection

The entire table column collection can be cleared by simply using the `Clear()` method.

[Hide](#) [Copy Code](#)

```
dt.Columns.Clear();
```

3.6 Cloning a Table

Once a table has been defined it can be used to create an identical table with the same column collection or it can be used as a basis for a new table where columns will be deleted, added or modified. The original table's `Clone()` method is used to create the new table with the same structure including schemas and constraints; however, it does not copy the content contained in the rows.

[Hide](#) [Copy Code](#)

```
DataTable dt1 = dt.Clone();
```

Now dt1 can be changed, for example:

[Hide](#) [Copy Code](#)

```
// delete a column
dt1.Columns.Remove("AtomicMass");
// add a new column to the table
dc = new DataColumn("IsotopeNbr",
System.Type.GetType("System.Int32"));
dc.DefaultValue = 0;
dt1.Columns.Add(dc);
// modify the name and caption of an existing column
dt1.Columns["AtomicNbr"].ColumnName = "AtomicNumber";
dt1.Columns["AtomicNbr"].Caption = "Atomic Number";
```

3.7 Rows – creating and adding to a table.

This section will show how to add rows and assign values to rows in the table rows collection using four equivalent methods for accessing individual cells within a row. The choice of method really depends upon the type of task such as the source of the data being used to fill the rows or simply extracting data from the rows.

1. Define a Table with Columns as described in the Tables and Columns section
2. The following scenario is the fundamental procedure for creating a row, filling the cells in the row and then adding the row to the table. This section also illustrates equivalent ways to index a cell, which provides the developer with much flexibility.

[Hide](#) [Copy Code](#)

```
// First create a DataRow variable
DataRow dr;

// Next create a new row and assign it to the DataRow
// object dr using DataTable's
// NewRow() method.
dr = dt.NewRow();
// dr now contains a cell for each column defined in the
// Columns collection
```

Four equivalent methods used to assign values to individual cells within a row.

Method 1

[Hide](#) [Copy Code](#)

```
// fill each cell using a zero-based cell integer column indexes
dr[0] = 1;
dr[1] = "Hydrogen";
dr[2] = "H";
dr[3] = 1.0078;
```

Method 2

[Hide](#) [Copy Code](#)

```
// fill each cell using the column name as the string column index
dr["AtomicNbr"] = 1;
dr["Element"] = "Hydrogen";
dr["Symbol"] = "H";
dr["AtomicMass"] = 1.0078;
```

Method 3

[Hide](#) [Copy Code](#)

```
// fill each cell using DataColumn dc -- this is more applicable
// when using a DataColumn foreach loop
// e.g. foreach (DataColumn dc in dt.Columns) ...
DataColumn dc;
dc = dt.Columns["AtomicNbr"];
dr[dc] = 1;

dc = dt.Columns["Element"];
dr[dc] = "Hydrogen";

dc = dt.Columns["Symbol"];
dr[dc] = "H";

dc = dt.Columns["AtomicMass"];
dr[dc] = 1.0078;
```

Method 4

```
// fill each cell using DataColumn dc and its ColumnName property
// which is identical to Method 3 but is
// included here for completeness
// Again, more applicable when using a foreach Loop
DataColumn dc;
dc = dt.Columns["AtomicNbr"];
dr[dc.ColumnName] = 1;

dc = dt.Columns["Element"];
dr[dc.ColumnName] = "Hydrogen";

dc = dt.Columns["Symbol"];
dr[dc.ColumnName] = "H";

dc = dt.Columns["AtomicMass"];
dr[dc.ColumnName] = 1.0078;

// add the row to the table's row collection
dt.Rows.Add(dr);
```

3. This scenario can be easily extended to a more general procedure to add n rows to the table. For example suppose a two dimensional object array 'ElementData' with n rows and dt.Columns.Count columns contains data to be added to the table. It could be loaded as follows:

```
DataRow dr;

int j;
for (int i=0; i < n; i++)
{
    j = -1;
    dr = dt.NewRow();
    foreach (DataColumn dc in dt.Columns)
    {
        j++;
        // fill each cell, using the column dc as the index, from the
        // previously defined two dimensional object array à ElementData
        if (dc.DataType == typeof(System.String))
            dr[dc] = (System.String)ElementData[i][j];
        else
            if (dc.DataType == typeof(System.Int32))
                dr[dc] = (System.Int32)ElementData[i][j];
            else
                if (dc.DataType == typeof(System.Decimal))
                    dr[dc] = (System.Decimal)ElementData[i][j];
    }
    dt.Rows.Add(dr);
}
```

3.8 Modifying data within an existing table row

There are a number of ways to modify data in a table with the following illustrating the basic mechanism. Other techniques will be presented in the following sections.

[Hide](#) [Copy Code](#)

```
// First create a DataRow variable
DataRow dr;
// Next assign the row in the Rows collection
// to be modified to dr, for example select row
with index = 0
dr = dt.Rows[0];
// Select the
// column within the row to be modified by specifying
// a column index and assign
the new value
dr["AtomicNbr"] = 1.00781;
```

Equivalent alternative coding methods are as follows:

[Hide](#) [Copy Code](#)

```
dt.Rows[0]["AtomicNbr"] = 1.00781;
```

or

[Hide](#) [Copy Code](#)

```
dt.Rows[0][0] = 1.00781;
```

3.9 Fill Table using `LoadDataRow()` method

The `LoadElementDataRow()` code example method in this section illustrates loading data into a table using the DataTable's `LoadDataRow` method that takes an object containing data for each cell within a row. The `LoadDataRow` method is bracketed by `BeginLoadData()` and `EndLoadData()` methods that turn off and on event notifications and other properties related to linked tables. Using these methods can prevent unnecessary processing by event handlers that would otherwise be triggered that are discussed in the Event Handler section. Also, the `LoadDataRow` method will modify an existing row if primary keys match or add the row to the Rows collection. Refer to the section on Row Versions that discusses the different versions of rows managed by the table's class for sample code illustrating the different behaviors of the `LoadDataRow` method when a table has a primary key and when it does not.

[Hide](#) [Copy Code](#)

```
private DataRow LoadElementDataRow(DataTable dt,
    int AtomicNbr, string Element,
    string Symbol, double AtomicMass)
{
    // Turns off event notifications,
    // index maintenance, and constraints
    // while loading data
    dt.BeginLoadData();
```

```

// Add the row values to the rows collection and
// return the DataRow. If the second
// argument is set to true, then dt.AcceptChanges() is called
//otherwise new rows are
// marked as additions and changes to existing rows are marked as
//modifications.
DataRow dr = dt.LoadDataRow(new object[]
    {AtomicNbr, Element, Symbol, AtomicMass}
    , false);
// Turns on event notifications, index maintenance, and constraints
// that were turned off
// with the BeginLoadData() method
dt.EndLoadData();
return dr; // returns the DataRow filled
// with the new values
}

```

3.10 Retrieving Table Content

The `GetTableData()` example method retrieves the column labels and row data from an input table and formats them into a string that can be used for printing, copying to the clipboard and exporting to a tab delimited text file.

[Hide](#) [Copy Code](#)

```

private string GetTableData(DataTable dt)
{
    StringBuilder TableData = new StringBuilder();
    // retrieve header row column labels
    TableData.AppendFormat("Row");
    foreach (DataColumn dc in dt.Columns)
        TableData.AppendFormat("\t{0}", dc.ColumnName);
    TableData.AppendFormat("\n");

    // retrieve rows
    int j = -1;
    foreach (DataRow dr in dt.Rows)
    {
        TableData.AppendFormat("[{0}]", ++j);
        foreach (DataColumn dc in dt.Columns)
        {
            TableData.AppendFormat("\t{0}", dr[dc] );
        }
        TableData.AppendFormat("\n");
    }

    return TableData.ToString();
}

```

The output string for our element table with one row would look like the following when it is displayed in a grid format using an Excel spreadsheet or the DataGrid.

Row	AtomicNbr	Element	Symbol	AtomicMass
[0]	1	Hydrogen	H	1.0078

3.11 Row Versions and Accepting/Rejecting Changes

3.11.1 Methods and Enumerations

This is an important section to understand because the Table class maintains different states and versions of rows that can be used to provide rollback, undo and transaction logging capability. That is, this state and version information provides very powerful programmatic control over table data and UI strategies.

Before discussing row states and versions there are Table and Row methods that need to be defined:

Table Method	Description
AcceptChanges()	Accepts all row changes to the table. Changes can be accepted to individual rows when the DataRow AcceptChanges() method is called.
RejectChanges()	Rejects all row changes to the table that have taken place since the last call to the Table or DataRow AcceptChanges() .
GetChanges()	Returns a table containing all rows that have been modified. This is particular useful when building transaction logs to satisfy government and corporate regulations, such as CFR21-11. <i>Note: If the Table AcceptChanges is called prior to GetChanges then there will be no changes and the return value is null.</i>
Rows Method	Description
Add()	Adds a row to the rows collection
InsertAt()	Inserts a row at a specific position in the rows collection
RemoveAt()	Removes a row at an index from the rows collection
AcceptChanges()	Accepts all changes to the row including changes to the individual cells including adding and deleting the row to and from the table respectively.
RejectChanges()	Rejects changes to the row restoring the original values
BeginEdit()	Begins a row editing session
CancelEdit()	Cancels a row editing session and restores all previous values

EndEdit()	Ends a row editing session
------------------	----------------------------

There are four different versions of Row Collections that are automatically maintained by the Table's object that provides extensive programmatic control over edits, deletes and inserts.

DataRowVersion	Description
Current	This version contains the current set of all values contained in each table row. The current set and the default set are identical during modifying a row without calling BeginEdit() after AcceptChanges() is called.
Default	The Default rows collection contains all of the changes. Each time a new row is created the new row is initialized to the default column values. Each time a cell value within a row is modified, the modification will be reflected in this table.
Proposed	This version as its name implies contains only rows that have proposed changes where they are only present during a call to BeginEdit() . When EndEdit() is called the proposed changes are reflected in the Current DataRow , in the Original DataRow and proposed DataRow is deleted. When the CancelEdit is called, the proposed DataRow is deleted and the Default DataRow is changed back to the Current DataRow values.
Original	The Original Rows collection is updated each time AcceptChanges() is called. These values are used when RejectChanges() and CancelEdit() are called to return the values back to the state before any changes occurred since the last call to AcceptChanges() .
RowState	Description
Added	The row is marked as <i>Added</i> when a row is added or inserted to the Rows Collection and before an AcceptChanges() method is called.
Deleted	After AcceptChanges() is called the row is marked as <i>Deleted</i> when any of the following is performed: dr.Delete() dt.Rows.RemoveAt(index) dt.Rows.Remove(dr)
Detached	Before AcceptChanges() is called the row is marked as <i>Detached</i> when any of the following is performed: dr.Delete() dt.Rows.RemoveAt(index) dt.Rows.Remove(dr)

Modified	After <code>AcceptChanges()</code> is called any row cell value that is changed causes the row to be marked as <i>Modified</i> .
Unchanged	After <code>AcceptChanges()</code> is called the row or all rows are marked as <i>Unchanged</i> depending upon whether it is a <code>DataRow</code> <code>AcceptChanges()</code> call or Table <code>AcceptChanges()</code> call.

[Hide](#) [Copy Code](#)

```
DataTable dt = new DataTable("Elements");
DataColumn AtomicNbr = new DataColumn("AtomicNbr",
System.Type.GetType("System.Int32"));
AtomicNbr.DefaultValue=0;
dt.Columns.Add(AtomicNbr);
DataColumn Element = new DataColumn("Element",
System.Type.GetType("System.String"));
Element.DefaultValue= "Element";
dt.Columns.Add(Element);
DataRow dr;
```

3.11.2 Sample 1 – Row States

This section shows the different row states and the conditions for them.

[Hide](#) [Copy Code](#)

```
dr = dt.NewRow();
dr[Element]="Hydrogen";
dr[AtomicNbr]= 1;
// NewRow Before Add: RowState=Detached
dt.Rows.Add(dr);
// NewRow After Add: RowState=Added
dt.Rows[0].AcceptChanges();
//NewRow After AcceptChanges: RowState=Unchanged
dt.Rows.RemoveAt(0);
// note that the row is marked as Detached when
// RemoveAt() or Remove() is used
//NewRow After RemoveAt: RowState=Detached
// Add the row back, accept the changes and then delete the row
// note that the row state is now marked as deleted when Delete()
// if Delete() is called prior to the row being
// added then the row is marked as Detached.
dt.Rows.Add(dr);
dt.AcceptChanges();
dr.Delete();
// NewRow After Delete: <RowState=Deleted>
```

The following code examples will illustrate the above method functionality. After each section of code will be four tables, one each for each type of `DataRowVersion`, that will show whether the version contains a row and if so their respective values.

NOTE
Rows that do not exist (designated with a 'No' value under column 'Has Versions') for a particular <code>RowState</code> version are added or included for readability

and clarity. That is, if all the rows in a version table were listed, these would not be in the Table Rows collection.

The presentation schema is from code Sample 2 to Sample N where each successive code Sample uses the results from the previous Sample. All changes that occur to the version tables for Sample code section are designated in **bold red**.

3.11.3 Sample 2 – Initial Loading of Table

[Hide](#) [Copy Code](#)

```
dr = dt.NewRow();
dr[Element]="Hydrogen";
dr[AtomicNbr]= 1;
dt.Rows.Add(dr);
dr = dt.NewRow();
dr[Element]="Helium";
dr[AtomicNbr]= 2;
dt.Rows.Add(dr);
dr = dt.NewRow();
dr[Element]="Lithium";
dr[AtomicNbr]= 3;
dt.Rows.Add(dr);
dr = dt.NewRow();
// this row contains default values
dt.Rows.Add(dr);
```

Row 0 has only an Original Version and it is marked as Deleted. The Current and Default versions are identical with the four new rows being marked as Added. The Proposed version table does not contain any values.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	Yes	Added	1	Hydrogen
[2]	Yes	Added	2	Helium
[3]	Yes	Added	3	Lithium
[4]	Yes	Added	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	Yes	Added	1	Hydrogen

[2]	Yes	Added	2	Helium
[3]	Yes	Added	3	Lithium
[4]	Yes	Added	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Deleted	0	Element
[1]	No			
[2]	No			
[3]	No			
[4]	No			

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	No			
[4]	No			

3.11.4 Sample 3 – DataRow AcceptChanges

[Hide](#) [Copy Code](#)

```
dt.Rows[0].AcceptChanges();
dt.Rows[1].AcceptChanges();
```

After the `Rows[0].AcceptChanges()` was called, the row that was marked deleted, row 0 in the above version tables, has been deleted from all versions and all other rows have new indices. The next `AcceptChanges()` command references row 1 in the newly ordered Rows Collection. In this case, the Row State is marked as Unchanged in versions Current, Default and Original. In the Original version there is only one row and it corresponds to Row 1 that was accepted. The Proposed version table does not contain any values.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Added	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Added	3	Lithium
[3]	Yes	Added	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Added	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Added	3	Lithium
[3]	Yes	Added	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	Yes	Unchanged	2	Helium
[2]	No			
[3]	No			

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	No			

3.11.5 Sample 4 – Table AcceptChanges

[Hide](#) [Copy Code](#)

```
dt.AcceptChanges();
```

After the **Table.AcceptChanges()** is called, all remaining rows are marked Unchanged in the Current, Default and Original Version tables and the Original version table is identical to the Current and Default version tables. The Proposed version table does not contain any rows.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	No			

3.11.6 Sample 5 – DataRow BeginEdit

[Hide](#) [Copy Code](#)

```
dt.Rows[1].BeginEdit();
dt.Rows[1]["Element"] = "Helium";
dt.Rows[1]["AtomicNbr"] = 222;
```

The above code begins an editing session on row 1 and the new values are reflected in the Default version table and values now appear in the Proposed version table. All other table version entries remain unchanged.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	222	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	Yes	Unchanged	222	Helium
[2]	No			
[3]	No			

3.11.7 Sample 6 – DataRow CancelEdit

[Hide](#) [Copy Code](#)

```
dt.Rows[1].CancelEdit();
```

The **CancelEdit()** command returns the default values back to the Original state and clears out the Proposed values from row 1 in the Proposed version table.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen

[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	No			

3.11.8 Sample 7 – DataRow BeginEdit – Example 2

[Hide](#) [Copy Code](#)

```
dt.Rows[3].BeginEdit();
```

The **BeginEdit()** method initializes the Proposed row 3 with default values.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	Yes	Unchanged	0	Element

3.11.9 Sample 8 – DataRow Change values – Example 2

[Hide](#) [Copy Code](#)

```
dt.Rows[3]["Element"]="Carbon";  
dt.Rows[3]["AtomicNbr"]= 12;
```

The Default and Proposed row 3 values have been changed to reflect Carbon and 12. All other rows in all versions remain unchanged.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	0	Element

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	Yes	Unchanged	12	Carbon

3.11.10 Sample 9 – DataRow EndEdit – Modified Rows

[Hide](#) [Copy Code](#)

```
dt.Rows[3].EndEdit();
dt.Rows[0]["Element"] = "Oxygen";
dt.Rows[0]["AtomicNbr"] = 8;
// Add a new row.
dr = dt.NewRow();
dt.Rows.Add(dr);
```

After **EndEdit()** is called, the Current and Default versions for row 3 are updated and marked as Modified. The Original version for row 3 still retains the original values before the changes and is marked as Modified.

The next two lines assign "Oxygen" and its atomic number to row 0 and these values are reflected in the Current and Default versions, which are also marked as Modified. The Original version for row 0 remains unchanged.

The next two lines adds a new row to the Current and Default version tables initialized with default values and marked as Added. Note that the new row does not appear in the Original version table.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Modified	8	Oxygen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Modified	12	Carbon
[4]	Yes	Added	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Modified	8	Oxygen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Modified	12	Carbon
[4]	Yes	Added	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Modified	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Modified	0	Element
[4]	No			

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	No			
[4]	No			

3.11.11 Sample 10 – DataRow AcceptChanges of Modified Rows

[Hide](#) [Copy Code](#)

```
dt.Rows[3].AcceptChanges();
```

The DataRow **AcceptChanges()** for row 3 causes the corresponding row in the Current and Default version tables to be marked as Unchanged and the Original version table now contains the same row values.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Modified	8	Oxygen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium

[3]	Yes	Unchanged	12	Carbon
[4]	Yes	Added	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Modified	8	Oxygen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	Yes	Added	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Modified	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	No			

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	No			
[4]	No			

3.11.12 Sample 11 – DataRow RejectChanges

[Hide](#) [Copy Code](#)

```
dt.Rows[0].RejectChanges();
```

Calling the DataRow **RejectChanges()** method for row 0 causes the corresponding row values for the Current and Default tables to revert to the Original version values and three tables, Current, Default and Original, have the Row State for row 0 marked as unchanged.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
-----	--------------	-----------	-----------	---------

[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	Yes	Added	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	Yes	Added	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	No			

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	No			
[4]	No			

3.11.13 Sample 12 – LoadDataRow without table having primary key

[Hide](#) [Copy Code](#)

```
dt.BeginLoadData();
dt.LoadDataRow(new object[]{1,"Deuterium"}, false);
dt.EndLoadData();
```

If a table does not have a primary key then the LoadDataRow method will create a new row and fill it with values in the object array. Looking at rows 0 and 5 in the Current and Default version tables,

they both have the same atomic number, but different element names. Also, Row 5 is marked as being *Added*.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	Yes	Added	0	Element
[5]	Yes	Added	1	Deuterium

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	Yes	Added	0	Element
[5]	Yes	Added	1	Deuterium

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Unchanged	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	No			
[5]	No			

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	No			
[4]	No			
[5]	No			

3.11.14 Sample 13 – LoadDataRow with table having primary key

[Hide](#) [Copy Code](#)

```
// must delete the row with a duplicate
// AtomicNbr in order to create
// a primary key.
dt.Rows.RemoveAt(5);
dt.AcceptChanges();
// Create a primary key and load the new object array data.
dt.PrimaryKey = new DataColumn[] {dt.Columns["AtomicNbr"]};
dt.BeginLoadData();
dt.LoadDataRow(new object[]{1,"Deuterium"}, false);
dt.EndLoadData();
```

If a table has a primary key then the **LoadDataRow** method will modify the data in the row if the primary keys match or else it will append the row to the table. Looking at row 0 in the Current and Default version tables, the element name has been changed from Hydrogen to Deuterium. In the three version tables Current, Default and Original row 0 is now marked as being *Modified*.

Current Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Modified	1	Deuterium
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	Yes	Unchanged	0	Element

Default Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Modified	1	Deuterium
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	Yes	Unchanged	0	Element

Original Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	Yes	Modified	1	Hydrogen
[1]	Yes	Unchanged	2	Helium
[2]	Yes	Unchanged	3	Lithium
[3]	Yes	Unchanged	12	Carbon
[4]	Yes	Unchanged	0	Element

Proposed Version

Row	Has Versions	Row State	AtomicNbr	Element
[0]	No			
[1]	No			
[2]	No			
[3]	No			
[4]	No			

3.11.15 Sample Code for Obtaining Version and State Information

The above tables were generated using the following procedure.

Hide Shrink ▲ Copy Code

```
static void PrintRowVersions(DataTable dt)
{
    DataRowVersion[] rowVer = new DataRowVersion[4];
    rowVer[0] = DataRowVersion.Current;
    rowVer[1] = DataRowVersion.Default;
    rowVer[2] = DataRowVersion.Original;
    rowVer[3] = DataRowVersion.Proposed;
    StringBuilder TableData = new StringBuilder();
    for(int i=0; i<rowVer.Length; i++)
    {
        // Print the value of each column in each row.
        TableData.AppendFormat("{0} Version\n", rowVer[i].ToString());
        // retrieve header row column labels
        TableData.AppendFormat("Row\tHas Versions\tRow State");
        foreach (DataColumn dc in dt.Columns)
            TableData.AppendFormat("\t{0}", dc.ColumnName);
        TableData.AppendFormat("\n");
        int n=-1;
        foreach(DataRow row in dt.Rows )
        {
            n++;
            if (row.HasVersion(rowVer[i]) )
            {
                // Print the specified version of the row's value.
                TableData.AppendFormat("[{0}]\tYes\t{1}",
                    n.ToString(), row.RowState.ToString());
                foreach (DataColumn dc in dt.Columns)
                {
                    TableData.AppendFormat("\t{0}", row[dc,rowVer[i]]);
                }
                TableData.AppendFormat("\n");
            }
            else
            {
                TableData.AppendFormat("[{0}]\tNo\t", n.ToString());
                for(int j=0; j<dt.Columns.Count; j++)
                    TableData.AppendFormat("\t ");
                TableData.AppendFormat("\n");
            }
        }
    }
}
```

```

    TableData.AppendFormat("\n");
}
// output string data to a text file using a StreamWriter
// StreamWriter sw = new StreamWriter("c:\\RowVersions.txt");
sw.Write(TableData.ToString());
// sw.Close();
}

```

3.12 Handling DataTable Errors

A DataTable can be checked to determine if it contains any rows with errors by examining the Table's **HasErrors** property value. The following code illustrates how to isolate the rows and their columns with errors.

[Hide](#) [Copy Code](#)

```

if (dt.HasErrors)
{ // Errors have occurred in rows in table dt
    foreach (DataRow dr in dt.Rows)
    {
        if(dr.HasErrors)
        {
            // Row has errors
            // GetColumnsInError() returns an array of
            // DataColumnns that contain errors
            foreach(DataColumn dc in dr.GetColumnsInError())
            {
                // GetColumnError returns a description of the Column error
                MessageBox.Show(dr.GetColumnError(dc.Ordinal));
            }
        }
    }
}
}

```

3.13 DataTable Events

The following code provides an example of adding a DataTable column changed event handler and the code within the handler illustrates some techniques for processing the new column values.

[Hide](#) [Copy Code](#)

```

dt.ColumnChanged += new DataColumnChangeEventHandler
    (this.SampleForm_ColumnChanged);
private void SampleForm_ColumnChanged(object sender,
    System.Data.DataColumnChangeEventArgs e)
{
    if(e.Column.Ordinal <op> ...)
    {
        // could perform validation checks such as range values
        // or formatting or other types of
        //processing on the changed column.
    }
    if(e.Row.HasErrors)

```

```
{  
    // clear the error  
    e.Row.SetColumnError(e.Column, string.Empty);  
    // check to see if row has any more errors  
    DataColumn [] dcErrors = e.Row.GetColumnsInError();  
    // if there are no more errors then clear the row error flag.  
    if(dcErrors.Length == 0)  
        e.Row.ClearErrors();  
}  
}
```

A Practical Guide to .NET DataTables, DataSets and DataGrids - Part 3

4 Data Sets

4.1 DataSet Methods

DataSet Method	Description
<code>AcceptChanges()</code>	Accepts all changes to the DataSet
<code>Clear()</code>	Removes all rows from all tables in the DataSet – that is, removes all data.
<code>Clone()</code>	Creates a new DataSet with all tables having the same Table structure including any constraints and relationships. No Data is copied.
<code>Copy()</code>	Same as for the DataSet <code>Clone()</code> but it includes all Data.
<code>GetChanges()</code>	Creates a DataSet that contains all changes made to the dataset. If <code>AcceptChanges</code> was called then only changes made since the last call are returned.
<code>HasChanges()</code>	Returns true if any changes were made to the DataSet including adding tables and modifying rows.
<code>WriteXml()</code>	Outputs an XML file containing schema with all Tables, Data, Constraints and relationships.
<code>ReadXml()</code>	Inputs an XML file containing schema, Tables, Data, Constraints and relationships..

4.2 DataSet Properties

DataSet Property	Description	
CaseSensitive	If set to true then string compares in DataSet tables are case sensitive otherwise they are not.	
DataSetName	Name of the DataSet	
HasErrors	Returns true if there are any errors within any tables in the DataSet	
Relationss	Collection of Relations	
	Method/Property	Description
	Add()	

	AddRange()	If two or more relations are to be added to the collection they can be added using this method. They are appended to the existing collection in the order specified in the range.
	CanRemove	Returns true if the relation can be removed from the collectionn
	Clear()	Removes all relations from the relations collection
	Contains()	Returns a true if the collection contains the named relation.
	Count	Returns the number of relations in the collection
	IndexOf()	Returns the index of a relation in the collection equal to name.
	Remove()	Removes a relation by Name from the collection
	RemoveAt()	Removes a relation by index from the collection.
Tables	Collection of Tables	
	Method/Property Description	
	Add()	Adds a table to the collection
	AddRange()	If two or more tables are to be added to the collection they can be added using this method. They are appended to the existing collection in the order specified in the range.
	Clear()	Removes all tables from the collection
	Contains()	Returns a true if the collection contains a table with TableName equal to name.
	Count	Returns the number of tables in the collection
	IndexOf()	Returns the index of a table in the collection with a TableName equal to name.
	Remove()	Removes a table by TableName from the collection
	RemoveAt()	Removes a table by index from the collection.

4.3 Loading A DataSet

4.3.1 From a Table

Tables created and filled with data as discussed in the Tables section can be added to the Tables collection by using the Add() method or they can be added at once using the **AddRange()** method.

Example of Two equivalent methods used to add tables dtElements and dtIsotopes to the DataSet ElementDS using Add() and AddRange()

Method 1 – **Tables.Add()**

[Hide](#) [Copy Code](#)

```
// Add the Elements table to the DataSet
elementDS.Tables.Add(dtElements);
// Add the Isotopes table to the DataSet
elementDS.Tables.Add(dtIsotopes);
```

Method 2 – **Tables.AddRange()**

[Hide](#) [Copy Code](#)

```
ElementDS.Tables.AddRange(new DataTable()
{dtElements, dtIsotopes});
```

4.3.2 From a Database

A DataSet tables collection can also be filled with linked tables containing data directly from a database recordset, which is considered bound data.

4.3.2.1 Method 1 – **sqlDataAdapter**

The following code illustrates how to directly load or bind a database recordset using the sqlDataAdapter's **Fill()** method where the select query string was used to create the recordset. After the Fill() method is called, ds will contain a table in its collection with column headers that match the field names in the select query string and column datatypes will match those specified in the database table elements. Each row will contain data corresponding to each field.

[Hide](#) [Copy Code](#)

```
System.Data.SqlClient.SqlConnection sqlConnection1;
System.Data.SqlClient.SqlDataAdapter sqlDataAdapter1;
System.Data.SqlClient.SqlCommand sqlSelectCommand1;
sqlConnection1 = new System.Data.SqlClient.SqlConnection();
sqlDataAdapter1 = new System.Data.SqlClient.SqlDataAdapter();
```



```

sqlSelectCommand1 = new System.Data.SqlClient.SqlCommand();
sqlSelectCommand1.CommandText = "SELECT ElementsID, AtomicNbr,"
+ "Symbol, AtomicMass, Element FROM [Elements]";
sqlDataAdapter1.SelectCommand = sqlSelectCommand1;
//
// sqlConnection1
// <replace xxxxx, nnnnn and dbName with appropriate values>
//
sqlConnection1.ConnectionString = "workstation id=xxxxx;packet "
+ " size=4096;user id=nnnnn; pwd=yyyyy;data "
+ " source=xxxxx; persist"
+ " security info=False; initial catalog=dbName";
sqlSelectCommand1.Connection = sqlConnection1;
DataSet ds = new DataSet();
sqlDataAdapter1.Fill(ds);

```

4.3.2.2 Method 2 – sqlDataReader

This method is more complex than using the sqlDataAdapter's `Fill()` method, but it allows for preprocessing of data prior to populating table rows and the data is not bound directly to the database. In the example, instead of restricting the database record set to contain distinct Atomic Number rows through a SQL query, it is done programmatically for illustration purposes.

[Hide](#) [Copy Code](#)

```

string SQL= "SELECT ElementsID, AtomicNbr, " +
"Symbol, AtomicMass, Element"
+ " FROM [Elements] order by AtomicNbr ASC";
sqlConnection sqlConnection1=
new sqlConnection (
    "connection info to MSDE or SQL Server 2000+");
sqlCommand sqlCommand = new sqlCommand (SQL,sqlConnection1);
sqlConnection1.Open();
sqlDataReader elementReader = sqlCommand.ExecuteReader();
// Starting with the element table dt defined in
// the Tables section, an ElementID column
// is added that will be used as the primary key for the row.
DataColumn dc = new DataColumn("ElementsID",
    System.Type.GetType("System.Guid"));
dt.Columns.Add(dc);
// Make 'ElementsID' a primary key:

dt.PrimaryKey = new DataColumn[]{dt.Columns["ElementsID"]};
// Fill table dt with data from the database table Elements:

```

Note: the sqlDataReader class has a method that can be used to determine whether a null or non-existent value was returned for a particular cell. For example:

[Hide](#) [Copy Code](#)

```

If (!elementReader.IsDBNull(elementReader.GetOrdinal("AtomicNbr")))
{
    // fill cell with value
}

```

```

else
{
    // handle this condition
    // for example fill cell with a default value
}

```

This check has been omitted in the following code for clarity, but it is a good practice to use it.

Hide Shrink ▲ Copy Code

```

DataRow dr;
int PrevAtomicNbr = 0;
try
{
    while(myReader.Read())
    {
        if (PrevAtomicNbr != elementReader.GetInt32(
            elementReader.GetOrdinal("AtomicNbr")))
        {
            PrevAtomicNbr = elementReader.GetInt32(
                elementReader.GetOrdinal("AtomicNbr"));
            dr = dt.NewRow();
            dr["ElementsID"] =
                elementReader.GetGuid(elementReader.GetOrdinal("ElementsID"));
            dr["AtomicNbr"] = elementReader.GetInt32(
                elementReader.GetOrdinal("AtomicNbr"));
            dr["Symbol"] = elementReader.GetString(
                elementReader.GetOrdinal("Symbol"));
            dr["Element"] = elementReader.GetString(
                elementReader.GetOrdinal("Element"));
            dr["AtomicMass"] =
                elementReader.GetDecimal(elementReader.GetOrdinal("AtomicMass"));
            dt.Rows.Add(dr);
        }
    }
}
finally
{
    elementReader.Close();
    sqlConnection1.Close();
}
dt.AcceptChanges();
// Add table dt to a new dataset ds and its tables collection
DataSet ds = new DataSet();
ds.Tables.Add(dt);

```

4.4 Linked Tables

This example shows how to link two tables together through a primary key. In this example a Table with TableName of *Elements* is created with a Primary key of 'Atomic Number'. The second Table with TableName of *Isotopes* is linked through a relationship coupling its Atomic Number column to *Elements* primary key.

```
// create a new DataSet named Periodic that will
// hold two linked tables with
// TableNames of Elements and Isotopes respectively.
DataSet elementDS = new DataSet("Periodic");
// Create Elements Table
DataTable dtElements = new DataTable("Elements");
dtElements.Columns.Add("Atomic Number", typeof(int));
dtElements.Columns.Add("Element", typeof(string));
dtElements.Columns.Add("Symbol", typeof(string));
// Make 'Atomic Number' a primary key in Elements table
dtElements.PrimaryKey = new DataColumn[]
{dtElements.Columns["Atomic Number"]};
// Create Isotopes Table
DataTable dtIsotopes = new DataTable("Isotopes");
dtIsotopes.Columns.Add("Symbol", typeof(string));
dtIsotopes.Columns.Add("Atomic Number", typeof(int));
dtIsotopes.Columns.Add("Isotope Number", typeof(int));
dtIsotopes.Columns.Add("Percent Abundance",
typeof(System.Decimal));
dtIsotopes.Columns.Add("Atomic Mass", typeof(System.Decimal));
// Add tables to the DataSet
ElementDS.Tables.AddRange(new DataTable[]{dtElements, dtIsotopes});
```

Dataset.Relations.Add()

```
// Add a relationship for Tables Elements and Isotopes
// through the primary key 'Atomic Number' in DataSet
// Periodic and name the relationship 'Isotopes'. This name is
// used in the DataGrid to select table rows, in table Isotopes,
// that contain isotope values for the selected element.
elementDS.Relations.Add("Isotopes",
elementDS.Tables["Elements"].Columns["Atomic Number"],
elementDS.Tables["Isotopes"].Columns["Atomic Number"] );
```

4.5 Linked tables in a dataset

4.5.1 Filling

Assume that another DataSet ds exists that has a table with index 0 in its Tables collection that contains both Element and Isotope data that will be used to fill rows in the linked tables contained in the elementDS DataSet tables collection as defined in the previous section. Assume Table[0] in DataSet ds has the following columns:

AtomicNbr, Element, Symbol, IsotopeNbr, PctAbundance and AtomicMass

where rows are sorted by Atomic numbers ascending and then by IsotopeNbr numbers ascending.

```

// Assign the table containing both Elements and
// Isotopes to dt using index 0
DataTable dt = ds.Tables[0];
// Create two DataTable variables dtElements and dtIsotopes and
// assign tables contained in the DataSet elementDS Tables
// collection using TableNames as indexes.
DataTable dtElements = elementDS.Tables["Elements"];
DataTable dtIsotopes = elementDS.Tables["Isotopes"];
DataRow drElement;
DataRow drIsotope;
int prevAtomicNbr = 0;
foreach (DataRow dr in dt.Rows)
{
    if(prevAtomicNbr != (int)dr["AtomicNbr"])
    { //need only one row per AtomicNbr in Table["Elements"]
        // Fill an element row with data from dt.
        prevAtomicNbr = (int)dr["AtomicNbr"];
        drElement = dtElements.NewRow();
        drElement["Atomic Number"] = dr["AtomicNbr"];
        drElement["Element"] = dr["Element"];
        drElement["Symbol"] = dr["Symbol"];
        dtElements.Rows.Add(drElement);
    }
    // Fill an isotope row with data from dt.
    drIsotope = dtIsotopes.NewRow();
    drIsotope["Isotope Number"] = dr["IsotopeNbr"];
    drIsotope["Symbol"] = dr["Symbol"];
    drIsotope["Atomic Number"] = dr["AtomicNbr"];
    drIsotope["Percent Abundance"] = dr["PctAbundance"];
    drIsotope["Atomic Mass"] = dr["AtomicMass"];
    dtIsotopes.Rows.Add(drIsotope);
}

```

4.5.2 Removing

To remove all linked tables or a selected table that is linked from the **DataSet**, it is first necessary to remove all relations, then constraints and then the table otherwise relationship/constraint table errors are generated.

The following code example provides a generic routine for removing all linked tables in a dataset.

```

public void RemoveAllTables(DataSet ds)
{
    // need to do it in reverse order due to constraints
    ds.Relations.Clear();
    for (int i=ds.Tables.Count -1; i >=0; i--)
    {
        ds.Tables[i].Constraints.Clear();
        ds.Tables.RemoveAt(i);
    }
}

```

4.6 XML Export and Import DataSet Data

4.6.1 WriteXml

All tables with their schemas, relationships, constraints and data contained in a DataSet can be exported in XML by specifying a DataSet property Namespace and using the **WriteXml** method.

For example:

[Hide](#) [Copy Code](#)

```
ds.Namespace = "http://www.mydomain.com/xmlfiles"
ds.WriteXml(FileName, XmlWriteMode.WriteSchema);
```

4.6.2 ReadXml

All tables with their schemas, relationships, constraints and data contained in an XML file are imported into a DataSet by specifying a DataSet property Namespace and using the **ReadXml** method. Once in the **DataSet** it is just like any other dataset.

For example:

[Hide](#) [Copy Code](#)

```
DataSet ds = new DataSet();
ds.Namespace = "http://www.mydomain.com/xmlfiles";
ds.ReadXml(FileName, XmlReadMode.ReadSchema);
```

4.7 Handling DataSet Errors

Similar to the DataTable **HasErrors** property the DataSet **HasErrors** property returns true if any errors occurred in any of the tables being managed by the DataSet.

[Hide](#) [Copy Code](#)

```
if(ds.HasErrors)
{ // One or more of the tables in the DataSet has errors.
  MessageBox.Show("DataSet has Errors");
  // Insert code to resolve errors.
  // Refer to 'Handling DataTable Errors' section for example of
  // handling errors within tables.
}
```

4.8 Updating Database with DataSet/DataTable changes

The following code shows how to create a DataSet containing all of the changes that have occurred to tables within a DataSet. The new DataSet can be used for updating the database.

[Hide](#) [Copy Code](#)

```
// Create temporary DataSet variable.
DataSet dsChanges;
// GetChanges for modified rows only.
dsChanges = ds.GetChanges(DataRowState.Modified);
// Check the DataSet for errors.
if(!dsChanges.HasErrors)
{
    // No errors were found, update the DBMS with the SqlDataAdapter da
    // used to create the DataSet.
    da.RowUpdating += new SqlRowUpdatingEventHandler(OnRowUpdating);
    da.RowUpdated += new SqlRowUpdatedEventHandler(OnRowUpdated);
    int res = da.Update(dsChanges); // returns the number of rows updated
    da.RowUpdating -= new SqlRowUpdatingEventHandler(OnRowUpdating);
    da.RowUpdated -= new SqlRowUpdatedEventHandler(OnRowUpdated);
}
```

A Practical Guide to .NET DataTables, DataSets and DataGrids - Part 4

5 Data Grid

A DataGrid is used to display an independent Table or a collection of Tables contained in a DataSet. It also provides a UI for editing, deleting and inserting records along with a set of event notifications for programmatic response and for data change tracking. In addition it supports a collection of DataGrid Table Styles that provides custom presentation for each table in its DataSource. WebForms and WinForms can both display a DataGrid bound and unbound to a database; however, there are significant differences in usage.

This section will present how to work with a DataGrid and its many features and different use models.

5.1 Methods and Properties

Methods	Description
Collapse	Collapse a specified row or all rows.
Expand	Expand a specified row or all rows
IsExpanded	Returns true if a specified row is expanded otherwise false.
NavigateBack	Navigates back to the previously displayed table in the grid.
NavigateTo	Navigate to a table in the grid.
SetDataBinding	Binds a dataset to the DataSource and selects a table.
Select	This method selects row and highlights it. It is not necessarily the same as the current row.
UnSelect	This method unselects a row and turns of highlighting. It does not change the current row.
Properties	Description
AllowSorting	Set to True allows column sorting using the column headers.
AllowNavigation	Allows navigation within the data grid using for example the arrow or tab keys.
AlternatingBackColor	This sets a background color for alternating rows in the DataGrid making it easier to read across a row.
DataSource	This returns or sets the source of data for the DataGrid. It is either a DataSet or a DataTable.
DataMember	This sets the table to be displayed. When the DataSource is a DataSet the DataGrid display is in a hierarchical mode,

	setting this to one of the Tables in the DataSet automatically forces it to be displayed.
CurrentCell	Gets or Sets the currently active cell including the CurrentRowIndex
CurrentRowIndex	Gets or Sets the current row

5.2 Assigning Data Sources

The DataSource for a DataGrid is either a Table or a DataSet. For example, a previously created DataTable dt object is assigned to the DataGrid as using the DataGrid's DataSource member follows:

[Hide](#) [Copy Code](#)

```
dg.DataSource = dt; // DataSource contains only a
// DataTable object
```

A DataSet that contains multiple tables in its table's collection can be assigned using the DataGrid's DataSource member as follows::

[Hide](#) [Copy Code](#)

```
DataSet ds = new DataSet();
ds.Tables.Add(dt1);
ds.Tables.Add(dt2);
...
ds.Tables.Add(dtn);

dg.DataSource = ds; // DataSource contains a DataSet object
```

By default the first table in the collection is displayed in the DataGrid, but a particular table can be selected using the DataGrid's DataMember property as follows:

[Hide](#) [Copy Code](#)

```
dg.DataMember = dt2.TableName;
```

or equivalently:

[Hide](#) [Copy Code](#)

```
dg.DataMember = ds.Tables[1].TableName; // zero-based index
```

or a DataSet can be assigned and a Table selected at the same time using the **SetDataBinding()** method as follows:

[Hide](#) [Copy Code](#)

```
dg.SetDataBinding(ds, dt2.TableName);
```


Note : The only restriction for DataSets is that tables must have a primary key, if not then an error will occur at the time of assigning the DataSet to the DataSource.

5.3 Formatting

5.3.1 DataGridTableStyle - WinForms

A DataTable is used to hold a collection of Column definitions and a collection of rows containing data for each column. A DataSet contains a collection of Tables and a DataGrid provides the interactive UI for the presentation of a table from a DataTable or from one contained in a DataSet.

The rendering of each table being managed by the DataGrid is carried out through the collections of individual table styles (*DataGridTableStyle*) where each table style contains a collection of column styles. Refer to Figure 1 for a pictorial view of these relationships.

Microsoft's .NET IDE provides **DataGridTextBoxColumn** or **DataGridBoolColumn** objects that are as their name implies a TextBox and Boolean or CheckBox column respectively. The TextBox column is the default column type used when declaring a DataGrid.

Unique rendering for each column is provided through the column style properties and methods. In addition it is possible to modify the .NET provided column styles or define new column styles such as ComboBoxes and ImageControls that are inherited from the base classes.

Note: DataGridTableStyle is not available for WebForms instead one must use itemstyles – see WebForm example.

In the following example assume the following:

A DataTable dt is defined

- A DataSet ds is defined and contains dt at index equal 0 in its Tables collection
- A DataGrid dg is defined with DataSource containing ds

Initially the DataGrid *dg* does not have any TableStyles contained in the TableStyles collection, which can be seen through the *Count* property value:

Hide Copy Code

```
dg.TableStyles.Count;
```

Also, the **dg.Controls** collection only contains Vertical and Horizontal Scrollbar controls.

Hide Shrink ▲ Copy Code

```
// First, a DataGridTableStyle object is declared  
// that will hold a collection of
```

```

// GridColumnStyles.
System.Windows.Forms.DataGridTableStyle DGStyle =
    new DataGridTableStyle();
// In this example the .NET DataGridTextBoxColumn class is used.
DataGridTextBoxColumn textColumn;
// Loop through each Column in table dt to get a DataColumn
// object that will be used
// to define properties for its TextBoxColumn style.
foreach (DataColumn dc in dt.Columns)
{
    textColumn = new DataGridTextBoxColumn();
    // the MappingName must correspond to the Table Column Name
    // in order to establish the relationship between them
    textColumn.MappingName = dc.ColumnName;
    // the HeaderText value is displayed in Header for the column, here
    // the Caption value is used.
    textColumn.HeaderText = dc.Caption;
    // specify some other property values
    textColumn.Width = 200;
    textColumn.Alignment = System.Windows.Forms.HorizontalAlignment.Left ;
    textColumn.ReadOnly = true;
    // Add this column object with its specifications to the
    // GridColumnStyles collection
    DGStyle.GridColumnStyles.Add(textColumn);
}
// The name of the DataGridTableStyle must match that of the table
// Since the DataGrid can contain multiple tables,
// similar the TableStyles collection
// can contain multiple DataGridTableStyles, one for each table.
DGStyle.MappingName = ds.Tables[0].TableName;
DGStyle.AlternatingBackColor = Color.Gainsboro;
DGStyle.AllowSorting = false;
DGStyle.ReadOnly = true;
// The Clear() method is called to ensure that
// the previous style is removed.
dg.TableStyles.Clear();
// Add the new DataGridTableStyle collection to
// the TableStyles collection
dg.TableStyles.Add(DGStyle);

```

5.3.2 Change only one table's DataGridTableStyles

The following shows how to change only one Table's **DataGridTableStyles**. Assume the table's style is contained in the **TableStyles** collection at index equal 0

[Hide](#) [Copy Code](#)

```

// Clear only a single table of its GridColumnStyles
dgConversionTable.TableStyles[0].GridColumnStyles.Clear();
DataGridTextBoxColumn textColumn;
// Loop through each Column in table dt to get a
// DataColumn object that will be used
// to define properties for its TextBoxColumn style.
foreach (DataColumn dc in dt.Columns)
{
    textColumn = new DataGridTextBoxColumn();
    // the MappingName must correspond to the Table Column Name
    // in order to establish the relationship between them

```

```

textColumn.MappingName = dc.ColumnName;
// the HeaderText value is displayed in Header for the column, here
// the Caption value is used.
textColumn.HeaderText = dc.Caption;
// specify some other property values
textColumn.Width = 200;
textColumn.Alignment = System.Windows.Forms.HorizontalAlignment.Left ;
textColumn.ReadOnly = true;
// Add this column object with its specifications to
// the GridColumnStyles collection
// for TableStyles[0]
dgConversionTable.TableStyles[0].GridColumnStyles.Add(textColumn);
}

```

5.3.3 Change a single column in a Table's DataGridTableStyle

One or more columns in an existing Table's DataGridTableStyles collection can be changed using code to similar to the following. In this example, the first method uses the *HeaderText* to find the column that is to be changed in a table that is named "Demo". Once the desired object is obtained, the current HeaderText value for the TextBoxColumn "old header text" is changed to "new header text". Also, the Width of the column is changed to 150. In the second example the Column *MappingName* **must** be used instead of the *HeaderText* if they are different.

5.3.3.1 Method 1 – for-loop through collection

In this example

[Hide](#) [Copy Code](#)

```

GridColumnStylesCollection gcsColl =
    dgConversionTable.TableStyles["Demo"].GridColumnStyles;
for (int i=0; i< gcsColl.Count; i++)
{
    if(gcsColl[i].GetType() == typeof(DataGridTextBoxColumn))
    {
        DataGridTextBoxColumn textColumn = (DataGridTextBoxColumn)gcsColl[i];
        If (textColumn.HeaderText == "old header text")
        {
            textColumn.Width = 150;
            textColumn.HeaderText = "new header text";
            break;
        }
    }
}
}

```

5.3.3.2 Method 2 – Direct access using column string name

This method can be used if you only want to change a specific column. Be sure to use the column *MappingName* and not its *Caption* or *HeaderText* if they are different.

[Hide](#) [Copy Code](#)

```

GridColumnStylesCollection gcsColl =

```

```

        dgConversionTable.TableStyles["Demo"].GridColumnStyles;
if(gcsColl.Contains("mappingName"))
if(gcsColl["mappingName"].GetType() == typeof(DataGridTextBoxColumn))
{
    DataGridTextBoxColumn textColumn =
    (DataGridTextBoxColumn)gcsColl["mappingName"];
    textColumn.Width = 150;
    textColumn.HeaderText = "new header text";
}

```

5.3.4 Create a DataGridTableStyle for Tables Based upon its Data

Each Table within a DataSet can have its own presentation and property style. The **formatDG** method below provides an example of how to programmatically define a style for each table based upon the data type for each column. It enforces these business rules:

Width of a column is greater than whichever is larger - the width of the **ColumnName** text string or the maximum of the string length of the content of each cell in that column. The maximum width is set to the width of 100 characters. The letter 'W' is added to each string for padding since it is usually the widest character in the alphabet.

- Columns of DataType **System.String** or **System.Guid** have Column alignment = **HorizontalAlignment.Left** and **ReadOnly= false**
- Columns of DataType of **System.DateTime** or **typeof(int)** have Column Alignment = **HorizontalAlignment.Center** and **ReadOnly= false**
- Columns of DateType **System.Double**, **System.Float** or **System.Decimal** have Column Alignment = **HorizontalAlignment.Right** and **ReadOnly= false** and a format of "0.00"
- The **NullText** property is set to **string.empty**. That is whenever a field has a null value then no text is displayed. This is only true for text columns and not for Boolean columns.
- Style Column **MappingName** is the column's **ColumnName** that matches the name of the field in the corresponding database table while its **HeaderText** is set to the more readable string contained in the column's **Caption** property.
- The name of the **DataGridTableStyle** is the same name as for the table. That is, both the DataSet Table and **DataGridTableStyle** collections are in sync through the use of the table name, which means that when the **DataGrid** displays the Table it will use the corresponding TableStyle.

Hide Shrink ▲ Copy Code

```

public void formatDG(DataGrid dg)
{
    // Assume DataGrid DataSource is a DataSet that
    // contains at least one table.
    if (dg.DataSource.GetType() == typeof(DataSet))
    {
        DataSet ds = (DataSet)dg.DataSource;
        if (ds != null)
        {
            if(dg.TableStyles.Count > 0)
                dg.TableStyles.Clear();
            float prevSumWidths = 0.0f;
            int maxHeight = 0;

```

```

int calch = 0;
foreach (DataTable dt in ds.Tables)
{
    // DataTable dt = ds.Tables[tableName];
    DataGridTableStyle DGStyle = new DataGridTableStyle();
    DGStyle.MappingName = dt.TableName;
    DGStyle.AllowSorting = false;
    DGStyle.AlternatingBackColor = Color.Gainsboro;
    DataGridTextBoxColumn textColumn;
    int nbrColumns = dt.Columns.Count;
    DataRow dr;
    ArrayList cWidths = new ArrayList();
    Graphics g = dg.CreateGraphics();
    Font f = dg.Font;
    SizeF sf;
    // get widths for each column header
    foreach (DataColumn c in dt.Columns)
    {
        // "W" is used as padding
        sf = g.MeasureString("W" + c.Caption, f);
        cWidths.Add(sf.Width);
    }
    // Loop through each row comparing against initial column
    // width and then against each new maximum width based upon
    // data contained in each row for that column.
    string strTemp;
    for(int i=0; i < dt.Rows.Count; i++)
    {
        dr = dt.Rows[i];
        for (int j=0; j < nbrColumns; j++)
        {
            if(!Convert.IsDBNull( dr[j]))
            {
                strTemp = dr[j].ToString();
                // careful strings can get
                // very long, limit it to 100 characters
                if(dr[j].GetType() ==
                typeof(System.String) && strTemp.Length > 100)
                sf = g.MeasureString("W"+
                strTemp.Substring(0,100), f);
            }
            else
            {
                sf = g.MeasureString("W"+ strTemp, f);
            }
            if(sf.Width > (float)cWidths[j])
                cWidths[j] = sf.Width;
        }
    }
}
// set each column with its determined width
// set alignment for each column
// set format for decimal numbers
float sumWidths = 0.0f;
foreach (DataColumn c in dt.Columns)
{
    if(c.DataType == typeof(bool))
    {
        DataGridBoolColumn boolColumn =
        new DataGridBoolColumn();
        boolColumn.MappingName = c.ColumnName;
        boolColumn.HeaderText = "" + c.Caption;
        boolColumn.Width = 50;
    }
}

```

```

        boolColumn.ReadOnly = false;
        boolColumn.Alignment = HorizontalAlignment.Center ;
        DGStyle.GridColumnStyles.Add(boolColumn);
        sumWidths += boolColumn.Width;
    }
    else
    {
        textColumn= new DataGridTextBoxColumn();
        textColumn.MappingName = c.ColumnName;
        textColumn.HeaderText = "+" + c.Caption;
        textColumn.ReadOnly = false;
        textColumn.NullText = string.empty;
        if(c.DataType == typeof(System.Guid) ||
           c.DataType == typeof(System.String))
        {
            textColumn.Alignment = HorizontalAlignment.Left ;
        }
        else
        if(c.DataType == typeof(System.DateTime))
            textColumn.Alignment =HorizontalAlignment.Center ;
        else
            if(c.DataType == typeof(int))
            {
                textColumn.Alignment =HorizontalAlignment.Center ;
            }
        else
        {
            textColumn.Alignment = HorizontalAlignment.Right ;
            if(c.DataType == typeof(System.Double)
               || c.DataType ==typeof(System.Decimal)
               || c.DataType == typeof(float))
                textColumn.Format = "0.00";
        }
        textColumn.Width = (int)(float)cWidths[c.Ordinal];
        DGStyle.GridColumnStyles.Add(textColumn);
        sumWidths += textColumn.Width;
        if (maxHeight < textColumn.TextBox.Height)
            maxHeight = textColumn.TextBox.Height;
    }
}
dg.TableStyles.Add(DGStyle);
// Adjust width of DataGrid to match calculated widths
// select maximum width of all tables in the dataset
// Adjust width to fit inside the parent container's width
if (prevSumWidths < sumWidths)
{
    prevSumWidths = sumWidths;
    prevSumWidths += dg.RowHeaderWidth;
    if (dg.VisibleRowCount < dt.Rows.Count)
        prevSumWidths += 16; // add width of scrollbar
    // check to see if it is greater than the width of its parent
    // e.g. a panel or tabpage control.
    if (prevSumWidths > (float)dg.Parent.Width)
        // allow room for the scroll bar and provide a little padding.
        prevSumWidths = (float)dg.Parent.Width - 16 - 5;
}
// Adjust height of DataGrid based upon row visibility
if (dg.VisibleRowCount >= dt.Rows.Count)
{
    calcH = (dg.VisibleRowCount)*maxHeight+2*dg.PreferredRowHeight+16;
}

```

```

}
else
{
    calcH = (dt.Rows.Count)*maxHeight+2*dg.PreferredRowHeight+16;
    // make a correction since all rows will now be visible
    if (calcH < dg.Parent.Height)
        prevSumWidths -= 16;
    }
    if (calcH >=dg.Parent.Height)
    {
        calcH = dg.Parent.Height - dg.Top - 25; // some arbitrary value
    }
}
dg.Size = new Size((int)prevSumWidths,calcH);
}
}

```

5.3.5 Defining DataGridViewTableStyles through Visual Studio .NET IDE

In the WinForm UI editor where a DataGridView control is placed onto the Form, the **DataGridViewTableStyle**'s collection property can be selected that will bring up the **DataGridViewTableStyle** Collection Editor dialog box shown in Figure 3. Clicking on the Add button will create a new DataGridViewStyle that can be named. Shown in member's section of the dialog box are two styles names **DGStyleElements** and **DGStyleIsotopes** where the property values for the former are shown. Note that DGStyleElements has a **MappingName** of *Elements* and whenever the DataGridView **MappingName** is also *Elements* then this style will be used to render the **DataTable** with the same name.

Clicking on the **GridColumnStyles** collection property (indicated by Red Arrow) launches the **DataGridViewColumnStyle** Collection Editor dialog shown in Figure 4. In the member's section is a list of column styles, one for each column that is in the Elements Table. Shown in the dialog are the property values for the **TextBoxColumn** style that is linked to the Atomic Number column in the Elements Table through the **MappingName** property.

The two dialog boxes in Figures 3 and 4 clearly show the DataGridView style collections of collections model that the DataGridView uses when rendering tables.

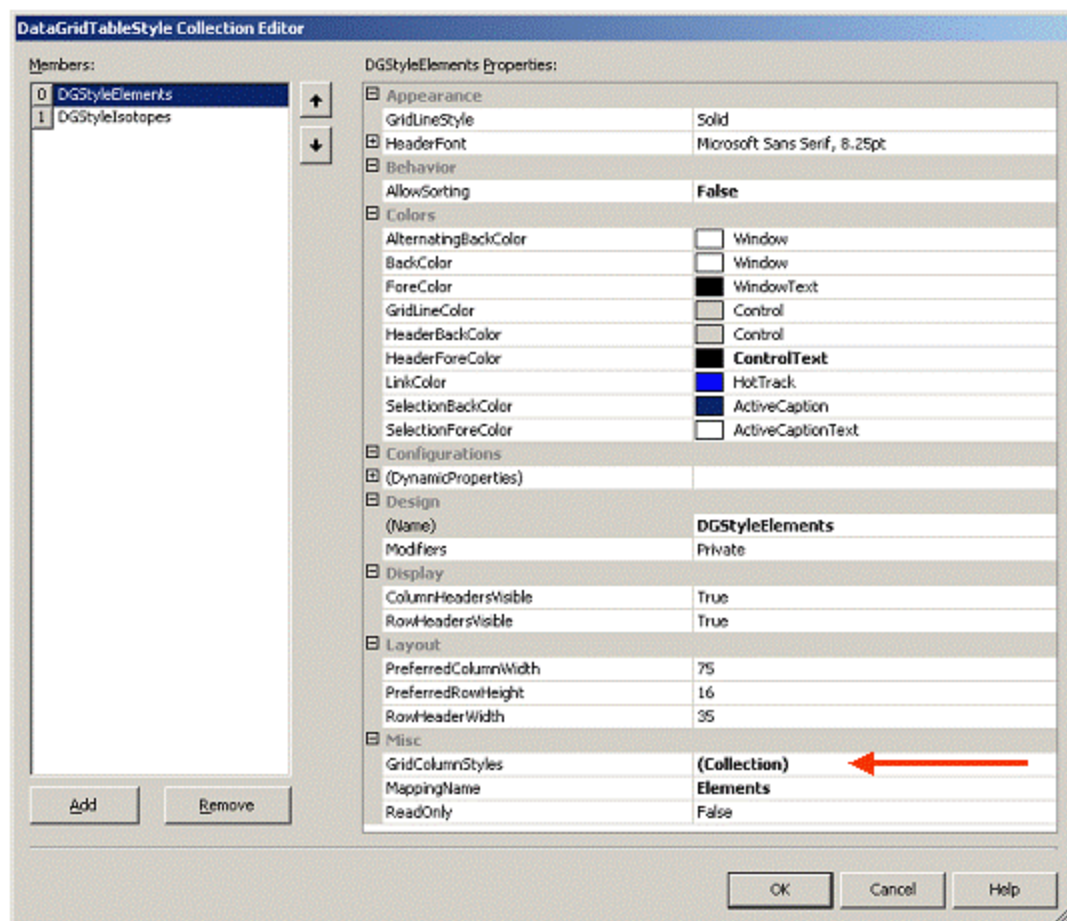


Figure 3 DataGridTableStyle Collection Editor Dialog Box

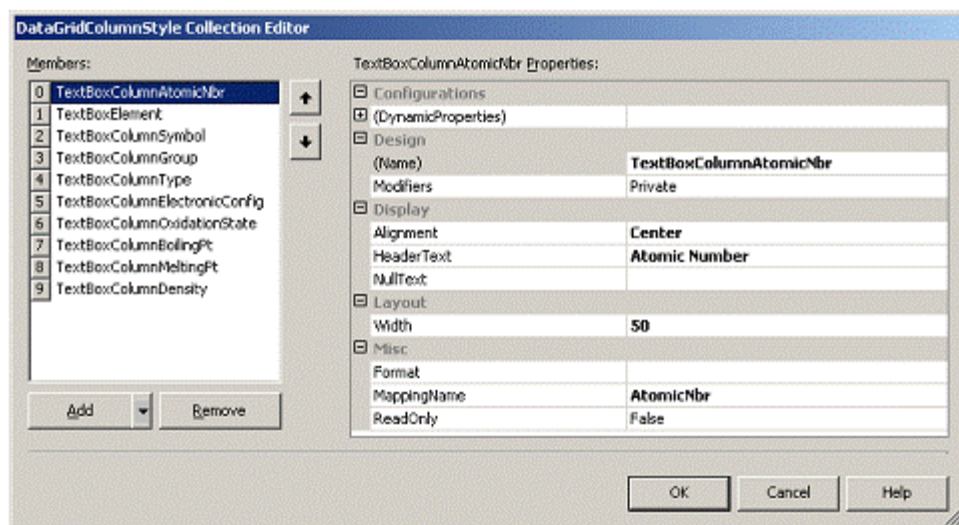


Figure 4 DataGridColumnStyle Collection Editor Dialog Box

5.4 Navigation

5.4.1 CurrencyManager

The **CurrencyManager** is especially useful when the DataSet contains linked tables and there is a need to make sure records are being added correctly or in navigating between them. The **CurrencyManager**'s *Position* property contains the zero-based index row number of the currently selected row of the table that it is bound to.

In a linked system such as the Element -> Isotope table relationship and the DataGrid **DataMember** property is set to the Isotope table, then the DataGrid *CurrentRowIndex* reflects the currently selected row in the Isotope table and the only way to know the row index in the Element table to which it is linked can only be determined by using the **CurrencyManger** for the Elements table.

The CurrencyManager can also be used to move to different rows in its bound table by using the *Count* property in conjunction with changing the value of the *Position* property; thus **MoveNext**, **MovePrev**, **MoveFirst** and **MoveLast** navigation methods can be created. Also when the position is changed the CurrencyManager's **CurrentChanged** and **PositionChanged** events can be monitored.

For example a CurrencyManger object with event handling can be declared as follows:

[Hide](#) [Copy Code](#)

```
private CurrencyManger cmElements;
cmElements = (CurrencyManager)this.BindingContext[
    dg.DataSource, "Elements"];
cmElements.CurrentChanged += new System.EventHandler(
    this.cmElements_CurrentChanged);
cmElements.PositionChanged += new System.EventHandler(
    this.cmElements_PositionChanged);
```

5.4.2 CurrentCell

The DataGridCell class constructor can be used to change the currently selected cell and row in the table currently active as specified by dg.DataMember. It also has two members that can be used to independently get or set the current table column and row numbers. For example:

```
dg.CurrentCell = new DataGridCell(row, column);
```

```
row = dg.CurrentCell.RowNumber;
```

```
column = dg.CurrentCell.ColumnNumber;
```

5.4.3 Selecting Rows

When coordinating a DataGrid with other types of UI components the `CurrentRowIndex`, `Select` and `UnSelect` methods need to be called.

Method	Action
<code>dg.CurrentRowIndex=newRow;</code>	Sets or Gets the new current row and moves the cursor indicator to that row. Refer to figure 3.
<code>dg.Select(newRow);</code>	Select() highlights the NewRow. Refer to figure 3.
<code>dg.UnSelect(previousRow);</code>	UnSelect() un-highlights the previousRow

5.4.4 Expand and Collapse linked tables

Figures 5 to 7 show tables in different states after calling the Collapse and Expand methods.

Elements										
	Atomic Number	Element	Symbol	Group	Type	Electronic Configuration	Oxidation States	Boiling Point (C)	Melting Point (C)	Density (g/cm³)
☐	1	Hydrogen	H	IA	gas	1s1	1	-252.7000	-259.2000	0.0710
▶	2	Helium	He	VIIIA	Noble gas	1s2		-268.9000	-269.7000	0.1260
☐	3	Lithium	Li	IA	Alkali metal	1s2 2s1	1	1330.0000	180.5000	0.5300
☐	4	Beryllium	Be	IIA	Alkaline earth metal	1s2 2s2	2	2770.0000	1277.0000	1.8500
☐	5	Boron	B	IIIA	non-metal	1s2 2s2 2p1	3		2030.0000	2.3400
☐	6	Carbon	C	IVA	non-metal	1s2 2s2 2p2	±4,2	4830.0000	3727.0000	2.2600
☐	7	Nitrogen	N	VA	non-metal	1s2 2s2 2p3	±3,5,4,2	-195.8000	-210.0000	0.8100
☐	8	Oxygen	O	VIA	non-metal	1s2 2s2 2p4	-2	-183.0000	-218.8000	1.1400
☐	9	Fluorine	F	VIIA	Halogen	1s2 2s2 2p5	-1	-188.2000	-219.6000	1.5050
☐	10	Neon	Ne	VIIIA	Noble gas	1s2 2s2 2p6		-246.0000	-248.6000	1.2000

Figure 5 `dg.Collapse(-1);` // Collapses all rows










Elements										
	Atomic Number	Element	Symbol	Group	Type	Electronic Configuration	Oxidation States	Boiling Point (C)	Melting Point (C)	Density (g/ml)
	1	Hydrogen	H	IA	gas	1s1	1	-252.7000	-259.2000	0.0710
	Isotopes									
	2	Helium	He	VIIIA	Noble gas	1s2		-268.9000	-269.7000	0.1260
	Isotopes									
	3	Lithium	Li	IA	Alkali metal	1s2 2s1	1	1330.0000	180.5000	0.5300
	Isotopes									
	4	Beryllium	Be	IIA	Alkaline earth metal	1s2 2s2	2	2770.0000	1277.0000	1.8500
	Isotopes									
	5	Boron	B	IIIA	non-metal	1s2 2s2 2p1	3		2030.0000	2.3400
	Isotopes									
	6	Carbon	C	IVA	non-metal	1s2 2s2 2p2	±4,2	4830.0000	3727.0000	2.2600
	Isotopes									
	7	Nitrogen	N	VA	non-metal	1s2 2s2 2p3	±3,5,4,2	-195.8000	-210.0000	0.8100
	Isotopes									
	8	Oxygen	O	VIA	non-metal	1s2 2s2 2p4	-2	-183.0000	-218.8000	1.1400
	Isotopes									
	9	Fluorine	F	VIIA	Halogen	1s2 2s2 2p5	-1	-188.2000	-219.6000	1.5050
	Isotopes									

Figure 6 `dg.Expand(-1);` // Expands all rows

Elements										
	Atomic Number	Element	Symbol	Group	Type	Electronic Configuration	Oxidation States	Boiling Point (C)	Melting Point (C)	Density (g/ml)
1	1	Hydrogen	H	IA	gas	1s1	1	-252.7000	-259.2000	0.0710
2	2	Helium	He	VIIIA	Noble gas	1s2		-268.9000	-269.7000	0.1260
Isolones										
3	3	Lithium	Li	IA	Alkali metal	1s2 2s1	1	1330.0000	180.5000	0.5300
4	4	Beryllium	Be	IIA	Alkaline earth metal	1s2 2s2	2	2770.0000	1277.0000	1.8500
5	5	Boron	B	IIIA	non-metal	1s2 2s2 2p1	3		2030.0000	2.3400
6	6	Carbon	C	IVA	non-metal	1s2 2s2 2p2	±4,2	4830.0000	3727.0000	2.2600
7	7	Nitrogen	N	VA	non-metal	1s2 2s2 2p3	±3,5,4,2	-195.8000	-210.0000	0.8100
8	8	Oxygen	O	VIA	non-metal	1s2 2s2 2p4	-2	-183.0000	-218.8000	1.1400
9	9	Fluorine	F	VIIA	Halogen	1s2 2s2 2p5	-1	-188.2000	-219.6000	1.5050
10	10	Neon	Ne	VIIIA	Noble gas	1s2 2s2 2p6		-246.0000	-248.6000	1.2000

Figure 7 dg.Expand(1); // Expands only row 2 (zero based index)

Figure 8 illustrates the result of using the DataGrid NavigateTo(arg1, arg1) method where arg1 is an integer row index in the primary table and arg2 is a string specifying the name of the table to Navigate to. In this example, Mercury is at row index = 79 in the Elements table and isotopes are contained in the link table "Isotopes" through the Atomic Number primary key-foreign key relationship.

Elements																	
◀ Elements:		Atomic Number:	80	Element:	Mercury	Symbol:	Hg	Group:	1B	Type:	Transition metal	Electronic Configuration:	[Xe]4f14 5d10 6s2	Oxidation States:	2, 1	Boiling Point (C):	357
	Symbol	Atomic Number	Isotope Number		Percent Abundance												
▶	Hg	80	196		0.1400												
	Hg	80	198		10.0200												
	Hg	80	199		16.8400												
	Hg	80	200		23.1300												
	Hg	80	201		13.2200												
	Hg	80	202		29.8000												
	Hg	80	204		6.8500												

Figure 8 dg.NavigateTo(79, "Isotopes");

5.5 Copy DataGrid to the Clipboard

The following sections contain Copy to Clipboard routines that access all tables or a specified table from a DataGrid and then they call a routine (*TableToString*) that formats table data into a string, which is copied to the Clipboard. Once the clipboard contains this string it can be pasted into Excel or into any other application that accepts text data. For example, in Excel or Winword the string data will be parsed back into tables.

Each of the methods checks the DataGrid data source member to determine whether it is a DataSet containing a collection of Tables or whether it is simply a Table Object.

5.5.1 Copy selected table in DataGrid to clipboard

Hide Copy Code

```
public void CopyDGtoClipboard(DataGrid dg, string tableName)
{
    if (dg.DataSource != null)
    {
```

```

DataTable dt = null;
if (dg.DataSource.GetType() == typeof(DataSet))
{
    DataSet ds = (DataSet)dg.DataSource;
    // need to use tableName when DataSet contains more than
    // one table
    if (ds.Tables.Contains(tableName))
        dt = ds.Tables[tableName];
}
else
if (dg.DataSource.GetType() == typeof(DataTable))
{
    dt = (DataTable)dg.DataSource;
    if (dt.TableName != tableName)
    {
        dt.Clear();
        dt = null;
    }
}
if (dt != null)
Clipboard.SetDataObject(TableToString (dt), true );
}
}

```

5.5.2 Copy all tables in DataGrid to clipboard

Hide Shrink  Copy Code

```

public void CopyDGtoClipboard(DataGrid dg)
{
    if (dg.DataSource != null)
    {
        if (dg.DataSource.GetType() == typeof(DataSet))
        {
            DataSet ds = (DataSet)dg.DataSource;
            if (ds.Tables.Count > 0)
            {
                string strTables = string.Empty;
                foreach (DataTable dt in ds.Tables)
                {
                    strTables += TableToString (dt);
                    strTables += "\r\n\r\n";
                }
                if (strTables != string.Empty)
                    Clipboard.SetDataObject(strTables, true );
            }
        }
        else
        if (dg.DataSource.GetType() == typeof(DataTable))
        {
            DataTable dt = (DataTable)dg.DataSource;
            if (dt != null )
                Clipboard.SetDataObject(TableToString(dt),
                    true );
        }
    }
}
}

```

5.5.3 Format table data into a string

This method returns a string containing the data in a DataTable object. The first line contains the name of the table, the second line contains the name of each column separated by a tab character and the remaining lines, one for each row in the table, contains the corresponding column data separated by a tab character.

Hide Shrink ▲ Copy Code

```
private string TableToString(DataTable dt)
{
    string strData = dt.TableName + "\r\n";
    string sep = string.Empty;
    if (dt.Rows.Count > 0)
    {
        foreach (DataColumn c in dt.Columns)
        {
            if(c.DataType != typeof(System.Guid) &&
                c.DataType != typeof(System.Byte[]))
            {
                strData += sep + c.ColumnName;
                sep = "\t";
            }
        }
        strData += "\r\n";
        foreach(DataRow r in dt.Rows)
        {
            sep = string.Empty;
            foreach(DataColumn c in dt.Columns)
            {
                if(c.DataType != typeof(System.Guid) &&
                    c.DataType != typeof(System.Byte[]))
                {
                    if(!Convert.IsDBNull(r[c.ColumnName]))
                        strData += sep +
                            r[c.ColumnName].ToString();
                    else
                        strData += sep + "";
                    sep = "\t";
                }
            }
            strData += "\r\n";
        }
    }
    else
        strData += "\r\n---> Table was empty!";
    return strData;
}
```

5.6 Exporting to a Tabbed delimited Text File

The following two methods `tabTextFile` and `SaveDataGridData` along with the `TableToString` method defined in an another section can be used to save all data contained in the DataGrid to a tab delimited text file specified through a SaveAs Dialog box.

```

public void tabTextFile(DataGrid dg)
{
    OpenFileDialog openFileDialog1 = new OpenFileDialog();
    openFileDialog1.InitialDirectory = "<A href='file:///c:/'>c:\\</A>";
    openFileDialog1.Filter =
        "Text Files (*.txt)|*.txt| All files (*.*)|*.*";
    openFileDialog1.RestoreDirectory = true ;
    openFileDialog1.Title="Export DataGrid to Text File";
    System.Windows.Forms.DialogResult res = openFileDialog1.ShowDialog();
    if(res == DialogResult.OK)
    {
        DataSet ds = new DataSet();
        DataTable dtSource = null;
        DataTable dt = new DataTable();
        DataRow dr;
        if(dg.DataSource != null)
        {
            if (dg.DataSource.GetType() == typeof(DataSet))
            {
                DataSet dsSource = (DataSet)dg.DataSource;
                // assume DataSet contains desired table at index 0
                if (dsSource.Tables.Count > 0)
                {
                    string strTables = string.Empty;
                    foreach (DataTable dt in dsSource.Tables)
                    {
                        strTables += TableToString (dt);
                        strTables += "\r\n\r\n";
                    }
                    if (strTables != string.Empty)
                        SaveDataGridData (strTables,openFileDialog1.FileName);
                }
            }
            else
            {
                if (dg.DataSource.GetType() == typeof(DataTable))
                {
                    dtSource = (DataTable)dg.DataSource;
                    if (dtSource != null )
                        SaveDataGridData (TableToString(dtSource),
                            openFileDialog1.FileName);
                }
            }
        }
    }
}

private void SaveDataGridData(string strData, string strFileName)
{
    FileStream fs;
    TextWriter tw = null;
    try
    {
        if (File.Exists(strFileName))
        {
            fs = new FileStream(strFileName, FileMode.Open);
        }
        else
        {
            fs= new FileStream(strFileName, FileMode.Create);
        }
        tw = new StreamWriter(fs);
    }
}

```

```

        tw.Write(strData);
    }
    finally
    {
        if (tw != null)
        {
            tw.Flush();
            tw.Close();
            MessageBox.Show("DataGrid Data has been saved to: " + strFileName
                , "Save DataGrid Data As", System.Windows.Forms.MessageBoxButtons.OK
                , System.Windows.Forms.MessageBoxIcon.Information);
        }
    }
}

```

5.7 Cloning Table contained in a DataGrid

This example shows how to clone a table in a DataGrid and then fill the cloned table with the content from the original table and return it in a new DataSet. Note that it checks to determine whether the DataSource is a DataSet or a Table.

Hide Shrink ▲ Copy Code

```

private DataSet CloneDataTable(DataGrid dgTable)
{
    DataSet ds = new DataSet();
    DataTable dtSource = null;
    DataTable dt = new DataTable();
    DataRow dr;
    if(dgTable.DataSource != null)
    {
        if (dgTable.DataSource.GetType() == typeof(DataSet))
        {
            DataSet dsSource = (DataSet)dgTable.DataSource;
            // assume DataSet contains desired table at index 0
            dtSource = dsSource.Tables[0];
        }
        else
        if (dgTable.DataSource.GetType() == typeof(DataTable))
        {
            dtSource = (DataTable)dgTable.DataSource;
            if (dtSource != null)
            {
                dt = dtSource.Clone();
                // dgConversionTable.CaptionText;
                dt.TableName = dtSource.TableName;
                dt.BeginLoadData();
                foreach (DataRow drSource in dtSource.Rows)
                {
                    dr = dt.NewRow();
                    foreach (DataColumn dc in dtSource.Columns)
                    {
                        dr[dc.ColumnName] = drSource[dc.ColumnName];
                    }
                    dt.Rows.Add(dr);
                }
            }
            dt.AcceptChanges();
        }
    }
}

```

```

        dt.EndLoadData();
        ds.Tables.Add(dt);
    }
}
return ds;
}

```

5.8 Comparison of WinForms & WebForms

This section will show differences between using a DataGrid in WinForms and WebForms. A table with a DataGrid caption of *C-Terminal Groups* will be used in the comparisons.

5.8.1 WinForms

In Figures 9 and 10, it can be seen that the C-Terminal table contains text, decimal numbers and integers that have been formatted using a procedure similar to `formatDG(DataGrid dg)` that uses `DataGridTableStyles`. This DataGrid was formatted as follows:

Integers are centered within the column

- Text strings are left justified. In the Formula column, which is generated from the values in the element decomposition columns, the electronic formula string has a space between each element instead of using numeric subscripts
- Decimal numbers have 4 decimal places and are right justified.
- The ID column is the primary key for the table.
- NOTE: `dc.Caption` is used as column headers in the DataGrid, in this case `dc.Caption` is equal to `dc.ColumnName`

ID	Name	AvgMass	MonoMass	C	H	N	O	S	P	Cl	Br	F	I	Formula
1	Free Acid	17.0073	17.0027	0	1	0	1	0	0	0	0	0	0	H O
2	Amide	16.0225	16.0187	0	2	1	0	0	0	0	0	0	0	H2N
3	Hydrazine	31.0371	31.0296	0	3	2	0	0	0	0	0	0	0	H3N2
4	Methyl ester	31.0343	31.0184	1	3	0	1	0	0	0	0	0	0	CH3O
5	N-ethyl Amide	44.0764	44.0500	2	6	1	0	0	0	0	0	0	0	C2H6N
6	Alcohol	3.0237	3.0235	0	3	0	0	0	0	0	0	0	0	H3
7	O-benzyl	107.1328	107.0497	7	7	0	1	0	0	0	0	0	0	C7H7O

Figure 9 WinForm DataGrid displaying C-Terminal Groups

In Figure 10, row 3, Hydrazine in the Name column is being edited. That is, editing is performed directly within each cell by simply moving the cursor to the cell and clicking on the left mouse button to set the focus. The dark blue arrow on the left margin of the table moves to the row being edited. If there are more rows than can be displayed in the DataGrid view then a scroll bar is used to bring other rows into view.

C-Terminal Groups															
	ID	Name	AvgMass	MonoMass	C	H	N	O	S	P	Cl	Br	F	I	Formula
	1	Free Acid	17.0073	17.0027	0	1	0	1	0	0	0	0	0	0	H O
	2	Amide	16.0225	16.0187	0	2	1	0	0	0	0	0	0	0	H ₂ N
▶	3	Hydrazine	31.0371	31.0296	0	3	2	0	0	0	0	0	0	0	H ₃ N ₂
	4	Methyl ester	31.0343	31.0184	1	3	0	1	0	0	0	0	0	0	CH ₃ O
	5	N-ethyl Amide	44.0764	44.0500	2	6	1	0	0	0	0	0	0	0	C ₂ H ₆ N
	6	Alcohol	3.0237	3.0235	0	3	0	0	0	0	0	0	0	0	H ₃
	7	O-benzyl	107.1328	107.0497	7	7	0	1	0	0	0	0	0	0	C ₇ H ₇ O
*															

Figure 10 WinForm DataGridView editing directly in a cell

To add more rows, the procedure is to enter new values into each cell in the row marked with an asterisk. The table class manages all new cell and row entries. One of the available methods will return a value indicating that changes to the table has occurred and then other methods can be used to determine the types of changes and to create a table containing only the changes. Events handlers can also be implemented to respond as changes are being made.

5.8.2 WebForms

Figures 11 – 13 illustrate the same table as in the Web Form, but there are differences. First and foremost is that the DataGridView is really an **HTML Table**. This can be readily seen by looking at the Formula column in Figures 11 and 12. The numeric subscript values are formatted using the HTML _{...} tags with numbers being obtained from the element decompositions columns {C, H, N, O, S, P, Cl, Br, F, I}. That is, all cell entries can be modified using all style elements available in HTML. Below Figure 11 is listed the HTML code generated by Microsoft's Visual Studio .NET IDE for the DataGridView. It contains user defined and default property values. Notice in the HTML code that the blue background with white letters is defined for the header as well as the additional Edit and Delete columns that are used when modifying row values. Also, since an HTML table does not have scroll bars, a paging mechanism (Prev, Next) can be used with the number of lines per page defined with the PageSize attribute.

C-Terminal Groups																
Edit	Delete	ID	Name	AvgMass	MonoMass	C	H	N	O	S	P	Cl	Br	F	I	Formula
Edit	Delete	1	Free Acid	17.0073	17.0027	0	1	0	1	0	0	0	0	0	0	HO
Edit	Delete	2	Amide	16.0225	16.01872	0	2	1	0	0	0	0	0	0	0	H ₂ N
Edit	Delete	3	Hydrazine	31.0371	31.0296	0	3	2	0	0	0	0	0	0	0	H ₃ N ₂
Edit	Delete	4	Methyl ester	31.0343	31.01838	1	3	0	1	0	0	0	0	0	0	CH ₃ O
Edit	Delete	5	N-ethyl Amide	44.0764	44.05	2	6	1	0	0	0	0	0	0	0	C ₂ H ₆ N
Edit	Delete	6	Alcohol	3.0237	3.0235	0	3	0	0	0	0	0	0	0	0	H ₃
Edit	Delete	7	O-benzyl	107.1328	107.0497	7	7	0	1	0	0	0	0	0	0	C ₇ H ₇ O
Edit	Delete	8	[New CTerm-Group]	0	0	0	0	0	0	0	0	0	0	0	0	
Prev Next																

Figure 11 WebForm DataGridView being displayed as an HTML Table

HTML code that was generated for the DataGridView by Visual Studio .Net

```

<asp:datagrid id="dgCTerminalGroups" style="Z-INDEX: 103; LEFT: 72px;
POSITION: absolute; TOP: 464px" runat="server"
Font-Size="Small" Font-Names="Arial" HorizontalAlign="Left"
Width="656px" AllowPaging="True" PageSize="10" CellPadding="3">
<AlternatingItemStyle BackColor="Gainsboro"></AlternatingItemStyle>
<ItemStyle Font-Size="Smaller" Font-Names="Arial" HorizontalAlign="Center">
</ItemStyle>
<HeaderStyle Font-Size="Smaller" HorizontalAlign="Center"
ForeColor="White" BackColor="Navy">
</HeaderStyle>
<Columns>
<asp:EditCommandColumn ButtonType="LinkButton" UpdateText="Update"
HeaderText="Edit" CancelText="Cancel" EditText="Edit">
</asp:EditCommandColumn>
<asp:ButtonColumn Text="Delete" HeaderText="Delete"
CommandName="Delete"></asp:ButtonColumn>
</Columns>
<PagerStyle NextPageText="Next" Font-Size="Smaller" PrevPageText="Prev"
HorizontalAlign="Center" ForeColor="White" BackColor="Navy"></PagerStyle>
</asp:datagrid>

```

5.8.2.1 Edit a row

When a row's Edit linkbutton is clicked each cell in the row is converted to a TextBox control with its TextBox.text property initialized to the value obtained in the table cell. This is illustrated in row 3 of Figure 12 where each TextBox Control has been formatted as presented in the section of the dg_PreRender(object sender, System.EventArgs e, string strTableName) procedure code that follows Figure 13 where the EditRowIndex member is greater than -1. Note that the ID and Formula columns TextBox control is readonly, DarkGray colored text and the Border Width is zero.

Also when a row's Edit linkbutton is clicked the EditCommand handler is called. All that this handler does is set the EditRowIndex member and then calls the DataBind() method.

```

private void dgCTerminalGroups_EditCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
    dgNTerminalGroups.EditRowIndex = e.Item.ItemIndex;
    dgNTerminalGroups.DataBind();
}

```

5.8.2.2 Cancel editing a row

The edit operation can be canceled by clicking on the Cancel linkbutton where its CancelCommand event handler sets the EditRowIndex to -1 and then calls the DataBind() method.

```

private void dgNTerminalGroups_CancelCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e)
{

```

```

dgNTerminalGroups.EditItemIndex = -1;
dgNTerminalGroups.DataBind();
}

```

C-Terminal Groups

Edit	Delete	ID	Name	AvgMass	MonoMass	C	H	N	O	S	P	Cl	Br	F	I	Formula
Edit	Delete	1	Free Acid	17.0073	17.0027	0	1	0	1	0	0	0	0	0	0	HO
Edit	Delete	2	Amide	16.0225	16.01872	0	2	1	0	0	0	0	0	0	0	H ₂ N
Update	Cancel	Delete	3	Hydrazine	31.0371	31.0296	0	3	2	0	0	0	0	0	0	H₃</sub>N₂
Edit	Delete	4	Methyl ester	31.0343	31.01838	1	3	0	1	0	0	0	0	0	0	CH ₃ O
Edit	Delete	5	N-ethyl Amide	44.0764	44.05	2	6	1	0	0	0	0	0	0	0	C ₂ H ₆ N
Edit	Delete	6	Alcohol	3.0237	3.0235	0	3	0	0	0	0	0	0	0	0	H ₃
Edit	Delete	7	O-benzyl	107.1328	107.0497	7	7	0	1	0	0	0	0	0	0	C ₇ H ₇ O
Edit	Delete	8	[New CTerm-Group]	0	0	0	0	0	0	0	0	0	0	0	0	

Prev Next

Figure 12 Editing C-Terminal Group row ID equal 3

5.8.2.3 Entering a new row

In the WinForm DataGridView a new row can be entered in the line designated with an asterisk, but this mechanism does not exist in WinForms. This implementation has created a special last row with a Key Phrase "[New CTerm-Group]" that lets the user know that this line is similar to the WinForm line with an asterisk and Edit will have the same behavior as the other lines. This line empty line was added to the DataTable after the object was filled from the database query and the update handler will remove this line prior to updating the database if no changes were made to the line other wise it will be added to the database table.

C-Terminal Groups

Edit	Delete	ID	Name	AvgMass	MonoMass	C	H	N	O	S	P	Cl	Br	F	I	Formula
Edit	Delete	1	Free Acid	17.0073	17.0027	0	1	0	1	0	0	0	0	0	0	HO
Edit	Delete	2	Amide	16.0225	16.01872	0	2	1	0	0	0	0	0	0	0	H ₂ N
Edit	Delete	3	Hydrazine	31.0371	31.0296	0	3	2	0	0	0	0	0	0	0	H ₃ N ₂
Edit	Delete	4	Methyl ester	31.0343	31.01838	1	3	0	1	0	0	0	0	0	0	CH ₃ O
Edit	Delete	5	N-ethyl Amide	44.0764	44.05	2	6	1	0	0	0	0	0	0	0	C ₂ H ₆ N
Edit	Delete	6	Alcohol	3.0237	3.0235	0	3	0	0	0	0	0	0	0	0	H ₃
Edit	Delete	7	O-benzyl	107.1328	107.0497	7	7	0	1	0	0	0	0	0	0	C ₇ H ₇ O
Update	Cancel	Delete	8	[New CTerm-Group]	0	0	0	0	0	0	0	0	0	0	0	

Prev Next

Figure 13 Adding a new C-Terminal Group row

5.8.2.4 Updating a row

This UpdateCommand event handler does the following:

Delete the last row ([New Cterm-Group]) from the DataSet Table if it is not the one being updated

- If it was deleted, then the **AcceptChanges()** is called so there will be no attempt to delete from the table in the database since it really does not exist there.
- Next get the value from each TextBox control and update the corresponding cell in the DataSet Table. Note the TextBox control has index equal zero in the Cells control collection.

- Update the CTerminal Database table with the row changes or new row addition
- Reset the EditItemIndex to -1, Set the DataSource to the modified DataSet and Call **DataBind()**

Note: if the update does not appear to work even though all of the event handlers are being called, check to see if you are using the **IsPostBack** method in the **Page_OnLoad()** event handler. For example

Hide Shrink ▲ Copy Code

```
if (!Page.IsPostBack)
    dgCTerminalGroups.DataBind();

private void dgCTerminalGroups_UpdateCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
    DataRow dr;
    DataSet ds;
    ds = (DataSet)dgCTerminalGroups.DataSource;
    DataTable dt = ds.Tables["CTerminal"];
    // this code segment simply deletes the blank row at the
    // end of the table if it is not the row being modified/updated.
    if (e.Item.DataSetIndex < dt.Rows.Count-1)
    {
        dt.Rows[dt.Rows.Count-1].Delete();
        dt.AcceptChanges();
    }
    dr = dt.Rows[e.Item.DataSetIndex];
    for(int i=0; i< dt.Columns.Count; i++)
    {
        // The TextBox is the 0th element of the Controls collection.
        // the edit, delete columns are at cells i=0
        // and i=1 respectively, skip those
        TextBox tbox = (TextBox)e.Item.Cells[i+2].Controls[0];
        if (!tbox.ReadOnly) // skip readonly cells;
            //set in PreRender() event handler.
            dr[dt.Columns[i].Caption] = tbox.Text;
    }
    if (ds.HasChanges())
    {
        DataSet dsChanges = ds.GetChanges();
        // call an update database function (code for method not shown)
        // merge the changes
        // with the DataSet contained by the DataGrid
        UpdateTerminalTable(dsChanges, "CTerminal") != null)
        ds.Merge(dsChanges);
    }
    // Switch out of edit mode.
    dgCTerminalGroups.EditItemIndex = -1;
    dgCTerminalGroups.DataSource=ds;
    dgCTerminalGroups.DataBind();
}
```

5.8.2.5 Setting Cell Style values

When individual cells in the DataGrid Table need to be formatted beyond global grid and column settings, a user defined DataGrid PreRender event handler can be defined. The following code

provides an example of a handler that can be used to set individual cell style values. The result of formatting based upon data types contained in the DataGrid can be seen by comparing Figures 13 and 14. The formatting includes

- setting horizontal alignment properties for all cells
- setting the [New Term Group] row to be in Italic and to have the **ForeColor** and **BackColor** properties set to **Color.DarkGray** and **Color.White** respectively
- setting the **ForeColor** and **BackColor** to **Color.DarkBlue** and **Color.LightGray** respectively for decimal numbers.

This is a powerful technique that can be used for example to reflect cell edit changes by changing the individual cell font and colors of a changed cell.

This example further illustrates formatting using HTML **CssStyles** and control properties in the section of the code where it specifically handles the row currently being edited which is contained in the **EditItemIndex** member.

Note

In the **dg_PreRender** code, the **for**-loop

Hide Copy Code

```
for (int j=0; j < dg.Items.Count; j++)
```

the maximum number of Items will be less than or equal to the **dg.PageSize**. To use **j** as an index for a table row, **j** needs to be augmented by **dg.PageSize*dg.CurrentPageIndex**. For example, column **i** for Item **j** in table **dt** is accessed as

```
dt.Rows[j + dg.PageSize*dg.CurrentPageIndex][i]
```

where **dg.CurrentPageIndex** contains a zero-based index.

Hide Shrink ▲ Copy Code

```
private void dg_PreRender(object sender,
    System.EventArgs e, string strTableName)
{
    DataGrid dg = (DataGrid)sender;
    DataTable dt = null;
    if (dg.DataSource.GetType() == typeof(DataSet))
    {
        DataSet ds = (DataSet)dg.DataSource;
        dt = ds.Tables[strTableName];
    }
    else
    if (dg.DataSource.GetType() == typeof(DataTable))
    {
        dt = (DataTable)dg.DataSource;
        if (dt != null)
        {
            for (int j=0; j < dg.Items.Count; j++)
            {
```

```

        if (j == (dg.Items.Count-1) &&
            ((string)dt.Rows[j+dg.PageSize*dg.CurrentPageIndex][
                "Name"]).IndexOf("[New]") > -1)
            bNewGroupRow = true;
        else
            bNewGroupRow = false;
        for(int i=0; i<(dt.Columns.Count); i++)
        {
            // accentuate [New Term Group] row with font
            // and color modifications
            if(bNewGroupRow && dg.EditItemIndex == -1)
            {
                dg.Items[j].Cells[i+2].ForeColor = Color.DarkGray;
                dg.Items[j].Cells[i+2].BackColor = Color.White;
                FontInfo fi = dg.Items[j].Cells[i+2].Font;
                fi.Italic = true;
            }
            if(dt.Columns[i].DataType == typeof(System.Guid) ||
                dt.Columns[i].DataType == typeof(System.String))
            {
                dg.Items[j].Cells[i+2].HorizontalAlign =
                    HorizontalAlign.Left ;
                dg.Items[j].Cells[i+2].Wrap = true ;
            }
            else
            if(dt.Columns[i].DataType == typeof(System.DateTime))
                dg.Items[j].Cells[i+2].HorizontalAlign =
                    HorizontalAlign.Center ;
            else
            if(dt.Columns[i].DataType == typeof(int))
            {
                dg.Items[j].Cells[i+2].HorizontalAlign =
                    HorizontalAlign.Center ;
            }
            else
            {
                if(dt.Columns[i].DataType == typeof(System.Double)
                    || dt.Columns[i].DataType ==typeof(System.Decimal)
                    || dt.Columns[i].DataType == typeof(float))
                {
                    dg.Items[j].Cells[i+2].HorizontalAlign =
                        HorizontalAlign.Right;
                    // set the text color to DarkBlue and background color
                    // LightGray
                    dg.Items[j].Cells[i+2].ForeColor = Color.DarkBlue;
                    dg.Items[j].Cells[i+2].BackColor = Color.LightGray;
                }
            }
        }
    }

    // check to see if a row is being edited, if it is
    // then the EditItemIndex member is non-negative
    // Each cell in the row now has an edit control,
    // this code gets the EditControl object for
    // the cell and then sets its cell specific attributes.
    // This example uses the Caption text as
    // a filter to select the cell type and then sets its
    // individual style properties. Two cells
    // ID and Formula are readonly and their text color is
    // set to DarkGray. The example also uses

```

```

// CssStyle to set global style attributes for each cell.
if (dg.EditItemIndex > -1)
{
    string strCaption;
    for(int i=2; i<=(dt.Columns.Count+1); i++)
    {
        TextBox tbox =
        (TextBox)dg.Items[dg.EditItemIndex].Cells[i].Controls[0];
        tbox.BackColor = Color.White;
        tbox.ForeColor = Color.DarkBlue;
        System.Web.UI.AttributeCollection
            tboxAttributes = tbox.Attributes;
        tboxAttributes.CssStyle.Add("font-weight", "bold");
        tboxAttributes.CssStyle.Add("text-align", "center");
        strCaption = dt.Columns[i-2].Caption.Trim();
        if (strCaption == "ID")
        {
            tbox.ReadOnly = true;
            tbox.ForeColor = Color.DarkGray;
            tbox.BorderWidth = 0;
            tbox.Width = 30;
        }
        else
        if (
            strCaption == "C" || strCaption == "H" ||
            strCaption == "N" || strCaption == "O" ||
            strCaption == "S" || strCaption == "P" ||
            strCaption == "Br" || strCaption == "Cl" ||
            strCaption == "F" || strCaption == "I")
        {
            tbox.Width = 30;
        }
        else
        if (strCaption == "Formula")
        {
            tbox.Width = 150;
            tbox.ReadOnly = true;
            tbox.ForeColor = Color.DarkGray;
            tbox.BorderWidth = 0;
            tbox.Text = string.Empty;
        }
        else
        if (strCaption == "AvgMass" || strCaption == "MonoMass")
        {
            tbox.Width = 75;
        }
        else
        if (strCaption == "Name")
        {
            tbox.Width = 150;
        }
        else
        {
            tbox.Width = 75;
        }
    }
}
}

```

C-Terminal Groups

Edit	Delete	ID	Name	AvgMass	MonohMass	C	H	N	O	S	P	Cl	Br	F	I	Formula
Edit	Delete	1	Free Acid	17.0073	17.0027	0	1	0	1	0	0	0	0	0	0	HO
Edit	Delete	2	Amide	16.0225	16.01872	0	2	1	0	0	0	0	0	0	0	H ₂ N
Edit	Delete	3	Hydrazine	31.0371	31.0296	0	3	2	0	0	0	0	0	0	0	H ₃ N ₂
Edit	Delete	4	Methyl ester	31.0343	31.01838	1	3	0	1	0	0	0	0	0	0	CH ₃ O
Edit	Delete	5	N-ethyl Amide	44.0764	44.05	2	6	1	0	0	0	0	0	0	0	C ₂ H ₅ N
Edit	Delete	6	Alcohol	3.0237	3.0235	0	3	0	0	0	0	0	0	0	0	H ₃
Edit	Delete	7	O-benzyl	107.1328	107.0497	7	7	0	1	0	0	0	0	0	0	C ₇ H ₇ O
Edit	Delete	8	[New CTerm-Group]	0	0	0	0	0	0	0	0	0	0	0	0	
Prev Next																

Figure 14 Formatting DataGrid using PreRender method

6 References

1. Microsoft Visual Studio .NET IDE help system
2. Microsoft Developer Network online help system
3. Programming Microsoft Windows with C#, Charles Petzold, Microsoft Press, 2002.
4. ASP.NET