

---

ComponentOne

# ADO.NET DataExtender

**ComponentOne, a division of GrapeCity**

201 South Highland Avenue, Third Floor  
Pittsburgh, PA 15206 USA

**Website:** <http://www.componentone.com>

**Sales:** [sales@componentone.com](mailto:sales@componentone.com)

**Telephone:** 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

## Trademarks

The ComponentOne product name is a trademark and ComponentOne is a registered trademark of GrapeCity, Inc. All other trademarks used herein are the properties of their respective owners.

## Warranty

ComponentOne warrants that the media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for \$25 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. ComponentOne is not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

## Copying and Distribution

While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only.

## Table of Contents

ADO.NET DataExtender Overview	3
Help with WinForms Edition	3
ADO.NET DataExtender Samples	3-5
Key Features	6-7
Differences between DataObjects for WinForms and ADO.NET DataExtender	7-8
Benefits of Using ADO.NET DataExtender	8-9
ADO.NET DataExtender Quick Start	10
Step 1 of 4: Connect to a Data Source	10-11
Step 2 of 4: Define One or More Views	11-13
Step 3 of 4: Connect the View to a Grid Control	13-14
Step 4 of 4: Run Your Quick Start Application	14
Design-Time Support	15
C1DataViewSet Smart Tag	15-16
C1DataViewSet Context Menu	16
C1DataViewSet Designer	16-19
Working with C1DataViewSet	20-21
C1DataViewSet Data Sources	21-22
Connecting a C1DataViewSet Component to a Typed or Untyped ADO.NET Dataset	22
C1DataView Definitions	22-23
Simple View Definition	23
Creating a Simple View Definition	23-25
Composite View Definition	25-26
Creating a Composite View Definition	26-28
C1ViewColumn Definitions	29
Defining Column Value Calculations	29-30
Defining Column Styles	30-31
Row and Column Level Constraints	31-32
Defining Constraints	32-34
C1DataView Relations	35
Creating Table and Column Aliases	35-37
Creating a Master-Detail Relationship Between C1DataViews	37-38
Specifying a Parent View	38-39
Interaction with an Underlying ADO.NET Dataset	39-40
Data Fetching	40-42

Working with DataSetExtender	43-44
Connection Information	44-45
Working with Typed DataSet	46
Data Driven Application Paradigm	46
Data Library Approach	46
Creating a Data Library	46-47
Creating a Typed DataSet	47-48
ADO.NET DataExtender Task-Based Help	49
Working with the C1ViewSetDesignerForm	49
Adding a C1DataView	49-50
Defining C1DataView	50-51
Previewing a C1DataView	51-52
Removing a C1DataView	52
Working with the C1DataView Definition Statement Builder	52
Accessing the C1DataView Definition Statement Builder	52-53
Adding a Child Table Node	53
Accessing and Changing Join Properties	53-55
Specifying or Changing the Table Alias	55
Specifying a Filter Condition	55
Adding Specific Columns to the View	55-56
Viewing and Changing the Definition Statement	56
Adding a Child View	57
Committing Changes	57
Sorting and Filtering Data	57-58
Updating a C1DataView Definition	58-59
Composite Row Editing	60
DataView Row Actions for an Outer Join	60-62
Modifying the Base DataTable(s)	62-63
DataView Row Actions for an Inner Join	63-64
Using C1DataViewSet with an Untyped ADO.NET DataSet	64-65
Fetching Data from the Server at Run Time	65-68

## ADO.NET DataExtender Overview

**ADO.NET DataExtender** is a powerful component that works with ADO.NET datasets and provides additional rich data views in the native ADO.NET datasets. **ADO.NET DataExtender** allows you to set up a data model for a Windows Form by means of a single component named `C1DataViewSet`. **ADO.NET DataExtender** simplifies working with ADO.NET in fact **ADO.NET DataExtender** mediates with ADO.NET, automating reading the schema and data from the database, exposing database tables and views as regular ADO.NET objects and providing an ability to set up customized views of their data.

**ADO.NET DataExtender** leverages ADO.NET features introduced with .NET 2.0, such as the ability to create entire datasets based on information contained in the database schema, including not only the data but also relations and constraints. In contrast to native ADO.NET, **ADO.NET DataExtender** allows you to reuse a single typed `DataSet` class that represents the whole database schema, with turned on constraints (including foreign key constraints), but populated with only a subset of data necessary in a certain application form. In general **ADO.NET DataExtender** simplifies data modeling and programming for Windows Forms applications.

### Getting Started

- [ADO.NET DataExtender Quick Start](#)
- [Design-Time Support](#)
- [ADO.NET DataExtender Samples](#)

## Help with WinForms Edition

### Getting Started

For information on installing **ComponentOne Studio WinForms Edition**, licensing, technical support, namespaces and creating a project with the control, please visit [Getting Started with WinForms Edition](#).

## ADO.NET DataExtender Samples

Please be advised that this ComponentOne software tool is accompanied by various sample projects and/or demos, which may make use of other development tools included with the ComponentOne Studio.

All samples use the standard Microsoft Access sample Northwind database, `C1NWind.mdb`. Note that the sample projects have the database location hard coded in the connection string:

### Documents\ComponentOne Samples\Common

If you have the Northwind database installed in a different location, you can change the connection strings, or copy the `C1NWind.mdb` file to the required location.

Please refer to the pre-installed product samples through the following path:

### Documents\ComponentOne Samples\WinForm

Click one of the following links to view a list of **ADO.NET DataExtender** samples:

#### To write code in Visual Basic

ADO.NET DataExtender includes the following Visual Basic samples:

Sample	Description
ViewSet_Define_Fill_Update	Explains the basic functionality of the <b>C1DataViewSet</b> component. Namely, it depicts how to define <b>C1DataViewSet</b> to work against ADO.NET typed <code>DataSet</code> , fill it with data and commit changes made by the user back to the database server. This sample uses the <b>C1DataViewSet</b> component connected to a typed

	DataSet object with two <b>C1DataView</b> objects joined by a <b>C1ViewRelation</b> object.
ClientFilterSortFind	Demonstrates filtering, sorting and searching of <b>C1DataView</b> rows, which are performed on a client. This sample uses the <b>C1DataViewSet</b> component connected to a typed DataSet object with a single <b>C1DataView</b> .
SchemaExtender	Demonstrates the usage of ADO.NET typed DataSet with DataSetExtender and explains the usage of constraints defined in an arbitrary .NET programming language (represented by the ConstraintsForm form). This sample uses the <b>C1DataViewSet</b> component connected to a typed DataSet object with a single <b>C1DataView</b> .
CompositeRowEditing	Depicts some details of composite rows editing behavior. This sample uses the <b>C1DataViewSet</b> component connected to a typed DataSet object with a single <b>C1DataView</b> .
UntypedDataSet	Demonstrates how to use <b>C1DataViewSet</b> in conjunction with untyped ADO.NET DataSet. <b>C1DataViewSet</b> allows the user to show and edit data from untyped DataSet tables according to its <b>C1DataView(s)</b> definitions. This sample uses the <b>C1DataViewSet</b> component connected to an untyped DataSet object with a single <b>C1DataView</b> .
Programmatic	Demonstrates how to create and use <b>C1DataViewSet</b> in run time programmatically. This sample creates the <b>C1DataViewSet</b> component connected to a typed DataSet object with two <b>C1DataView</b> objects joined by a <b>C1ViewRelation</b> object.
ColumnStyles_In_FlexGrid	Demonstrates how to set up column styles, which are honored by the <b>C1FlexGrid</b> control in run time, using an ADO.NET DataSet Extender.
ColumnStyles_In_TrueDBGrid	Demonstrates how to set up column styles, which are honored by the <b>C1TrueDBGrid</b> control in run time, using an ADO.NET DataSet Extender.

## To write code in C#

ADO.NET DataExtender includes the following C# samples:

Sample	Description
ViewSet_Define_Fill_Update	Explains the basic functionality of the <b>C1DataViewSet</b> component. Namely, it depicts how to define <b>C1DataViewSet</b> to work against ADO.NET typed DataSet, fill it with data and commit changes made by the user back to the database server. This sample uses the <b>C1DataViewSet</b> component connected to a typed DataSet object with two <b>C1DataView</b> objects joined by a <b>C1ViewRelation</b> object.
ClientFilterSortFind	Demonstrates filtering, sorting and searching of <b>C1DataView</b> rows, which are performed on a client. This sample uses the <b>C1DataViewSet</b> component connected to a typed DataSet object with a single <b>C1DataView</b> .
SchemaExtender	Demonstrates the usage of ADO.NET typed DataSet with DataSetExtender and explains the usage of constraints defined in an arbitrary .NET programming language (represented by the ConstraintsForm form). This sample uses the <b>C1DataViewSet</b> component connected to a typed DataSet object with a single <b>C1DataView</b> .

CompositeRowEditing	Depicts some details of composite rows editing behavior. This sample uses the <b>C1DataViewSet</b> component connected to a typed DataSet object with a single <b>C1DataView</b> .
UntypedDataSet	Demonstrates how to use <b>C1DataViewSet</b> in conjunction with untyped ADO.NET DataSet. <b>C1DataViewSet</b> allows the user to show and edit data from untyped DataSet tables according to its <b>C1DataView(s)</b> definitions. This sample uses the <b>C1DataViewSet</b> component connected to an untyped DataSet object with a single <b>C1DataView</b> .
Programmatic	Demonstrates how to create and use <b>C1DataViewSet</b> in run time programmatically. This sample creates the <b>C1DataViewSet</b> component connected to a typed DataSet object with two <b>C1DataView</b> objects joined by a <b>C1ViewRelation</b> object.
ColumnStyles_In_FlexGrid	Demonstrates how to set up column styles, which are honored by the <b>C1FlexGrid</b> control in run time, using an ADO.NET DataSet Extender.
ColumnStyles_In_TrueDBGrid	Demonstrates how to set up column styles, which are honored by the <b>C1TrueDBGrid</b> control in run time, using an ADO.NET DataSet Extender.

## Key Features

**ADO.NET DataExtender** allows you to set up a data model for a Windows Form by means of a single component named `C1DataViewSet`. Some of the key benefits of **ADO.NET DataExtender** include:

- **Simplicity**

A single **C1DataViewSet** component serves as the binding agent for all tables and relations. Without **ADO.NET DataExtender**, two components (a `BindingSource` and a `DataAdapter`) are required to represent each table or relation.

- **Connectivity**

`C1DataViewSet` can represent data from the following types of data sources:

- - Typed ADO.NET DataSet
  - Untyped ADO.NET DataSet
  - Connection String
- **Programmability**

The `C1DataViewSet` component represents a rich set of events for custom processing of row navigation and changing occurrences. For the convenience of writing event handlers in design time, all `C1DataView` events are propagated by the owning `C1DataViewSet` component. `C1DataViewSet` event data determines in what specific `C1DataView` an event has occurred.

The **C1DataViewSet** component supports the following events, which are useful for responding to changes in row currency, current row values, and connections:

Event	Description
<b>ColumnChanged</b>	Occurs when a <code>C1ViewColumn</code> value is being retrieved. This event allows process editing actions made on <code>C1ViewColumn</code> objects.
<b>ColumnChanging</b>	Occurs after a value has been changed for the specified <code>C1ViewColumn</code> in a <code>C1ViewRow</code> . This event allows process editing actions made on <code>C1ViewColumn</code> objects.
<b>CurrentChanged</b>	This special event, <code>CurrentChanged</code> , is useful when you need to process current row change independent of the reason for this change in the case that another row becomes current, due to navigation, sorting, filtering, and so on, or due to a change of current row column values.
<b>PositionChanged</b>	<code>C1DataView</code> objects provide data navigation capabilities, so you don't need to retrieve <code>CurrencyManager</code> for managing row navigation. You can use the <code>PositionChanged</code> event to process navigation occurrences (along with <code>Current</code> and <code>Position</code> properties).
<b>RowChanged</b>	Occurs after editing of a <code>C1ViewRow</code> has been completed. This event allows process editing actions made on <code>C1ViewRow</code> objects.
<b>RowChanging</b>	Occurs before editing of <code>C1ViewRow</code> objects. This event allows process editing actions made on <code>C1ViewRow</code> objects.
<b>ViewListChanged</b>	If you need to process changes in view rows that occurred due to changes made by means of a <code>C1DataView</code> and underlying <b>DataTable</b> objects, use the <code>ViewListChanged</code> event.

Within these event handlers, the following properties can be used to address the current row for the affected **C1DataView** object:

Property	Description
<b>Current</b>	Returns the C1ViewRow of the CurrencyManager that services this C1DataView.
<b>Position</b>	Returns the position of the current C1ViewRow in the CurrencyManager that services this C1DataView.

- **Smart Update**

The Update method of the C1DataViewSet component automatically determines the correct order in which rows should be committed to the server. It also refreshes client row columns with new values generated on the server, including the cases of server-generated auto-increment columns in conjunction with master-detail relationships between tables.

Without **ADO.NET DataExtender**, the user must call the Update method of one or more DataAdapters in order to commit changes from a DataSet to the underlying data source. In many circumstances, the order in which DataSet changes are sent to the data source is crucial. For example, if a primary key value for an existing row is updated, and a new row has been added with the new primary key value, the update must be processed before the insertion.

- **Composite Views**

Composite views (similar to a SQL join) combine multiple DataTables in a single DataTable object. As updates are made to the composite view, they are automatically reflected in the constituent tables, and vice versa.

- **Column Styles**

Column styles are UI-related attributes that can be assigned to data columns in the **C1ViewSetDesignerForm**, and then realized at run time when bound to a ComponentOne control. The current set of controls that support column styles includes **C1TrueDBGrid**, **C1FlexGrid**, and **C1Input**.

You can define, format, or edit mask for a certain column. You can also set it up to have a lookup combo box or any other ComponentOne control that supports this feature. Simply by being connected to this column, the ComponentOne control will automatically reflect those definitions.

- **Calculated Columns**

Read-only calculated columns can be specified in the **C1ViewSetDesignerForm** using any .NET-compatible language, or as part of the SQL-like statement that defines the view. The column expressions are evaluated at run time as navigation and updates occur through interaction with *any* bound control.

- **Constraint Expressions**

Column and row level constraints can be specified in the **C1ViewSetDesignerForm** using any .NET-compatible language. The expressions are evaluated at run time as updates occur through interaction with *any* bound control.

- **Column and row level constraints usage:**

- Column level constraints are useful for enforcing upper and lower bounds and non-null values.
- Row level constraints are commonly used to express validation criteria involving multiple columns.

## Differences between DataObjects for WinForms and ADO.NET DataExtender

The following describes the differences between **DataObjects for WinForms** and **ADO.NET DataExtender**.

### DataObjects for WinForms

**C1DataObjects** provides a powerful replacement for ADO.NET. It has its own dataset layer, where data integrity

constraints and other business logic can be defined. **C1DataObjects** is available for .NET 1.0 and 2.0.

## ADO.NET DataExtender

**ADO.NET DataExtender** works with and extends ADO.NET. It works with ADO.NET datasets and provides additional rich data views in the native ADO.NET datasets. **ADO.NET DataExtender** is available for .NET 2.0 only.

## Differences between DataObjects for WinForms and ADO.NET DataExtender

Although both products are related to the data layer of .NET applications, they are fundamentally different. **ADO.NET DataExtender** is not just a new, enhanced version of **DataObjects for WinForms**. It delegates the data storage and business logic to ADO.NET and concentrates on enhancing its features, rather than trying to replace it.

## DataObjects for WinForms users migrating to ADO.NET DataExtender

We expect many **DataObjects for WinForms** users to migrate to **ADO.NET DataExtender** (plus ADO.NET 2.0). However, users who rely on certain **DataObjects for WinForms** features (specifically 3-tier application configuration and virtualized record sets) will not be able to migrate immediately, and that's why we will continue to maintain **DataObjects for WinForms**.

## Benefits of Using ADO.NET DataExtender

**ADO.NET DataExtender** leverages ADO.NET features introduced with .NET 2.0 that were not available before. For example, the ability to create entire datasets based on information contained in the database schema, including not only the data but also relations and constraints. In contrast to native ADO.NET, **ADO.NET DataExtender** allows you to reuse a single typed DataSet class that represents the whole database schema, with turned on constraints (including foreign key constraints) but populated with only a subset of data necessary in a certain application form.

Other key benefits of **ADO.NET DataExtender** include:

- **Rich data views**

**ADO.NET DataExtender** concentrates on providing rich data views using ADO.NET as the underlying data storage and business logic engine. For example, you can use a SQL-like syntax to create views that combine data from different tables. These are different from regular SQL queries because the views are connected to the client-side source tables. Changing the view affects the table and vice versa.

- **Ability to attach presentation attributes to data columns**

You can also attach presentation attributes to data columns, including data entry masks, display formats, and value-translation maps (for example, to show a customer name instead of their ID in an Orders table).

- **Capability to surpasses view level limitations**

**ADO.NET DataExtender** goes beyond the view level boundaries. For example, it can fetch data from the server on demand and update the data back to the server, as well as, define additional constraints on the data (using the **DataSetExtender** class).

- **Mediation with the DataSet object**

Although it works with ADO.NET, **ADO.NET DataExtender** doesn't force programmers to deal with the DataSet object

at all. At a minimum, you can simply specify a connection string and the **ADO.NET DataExtender** object will automatically read the schema and data from the database, exposing database tables and views as regular ADO.NET objects and providing an ability to set up customized views of their data.

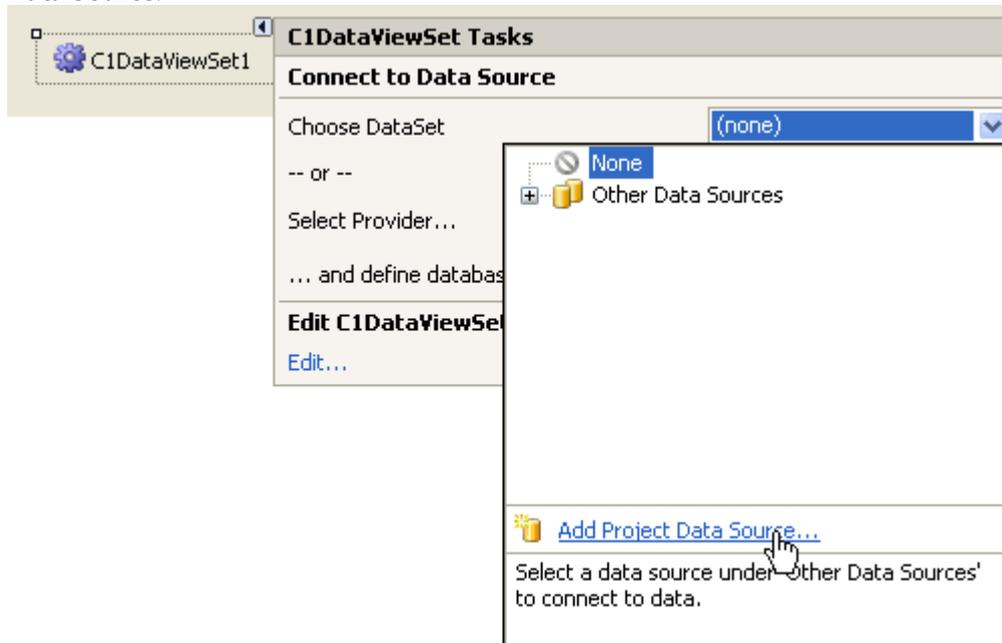
## ADO.NET DataExtender Quick Start

This quick start tutorial shows you how to add the **C1DataViewSet** component to your form, connect to a data source, and define some views. By following the steps outlined in the quick start, you will be able create a rich data view.

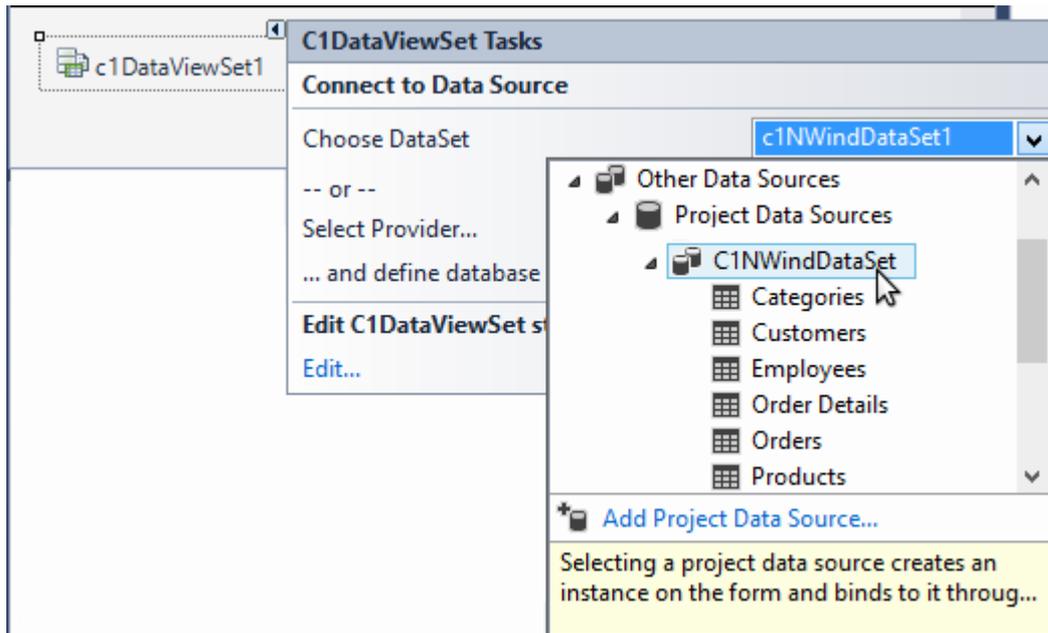
### Step 1 of 4: Connect to a Data Source

To begin, [create a .NET project](#) and [add the C1DataViewSet component to your form](#). To set up your new Windows form and connect to a data source, complete the following steps:

1. Click the smart tag (🔗) located above the **C1DataViewSet** component to open its **C1DataViewSet Tasks** menu.
2. From the **C1DataViewSet Tasks** menu, select the **Choose DataSet** drop-down arrow and select **Add Project Data Source**.



3. The **DataSource Configuration Wizard** appears, and **DataBase** is selected. Click **Next**.
4. Click **New Connection** to locate and connect to a database.
5. Leave the Data source set to **Microsoft Access Database File**.
6. Click **Browse** and select **C1NWind.mdb** located in the samples directory that is located in one of the following directories: **Documents\ComponentOne Samples\Common**
7. Then click **Open**. You can test the connection and then click **OK**.
8. The connection string appears in the drop-down list. Click **Next**.
9. Since it is not necessary to copy the database to your project, click **No** in the dialog box.
10. Click **Next** to save the connection string as **C1NWindConnectionString**.
11. Select **Tables**, and then click **Finish** to complete your datasource configuration.
12. Select **C1NWindDataSet** to bind the database to your project.



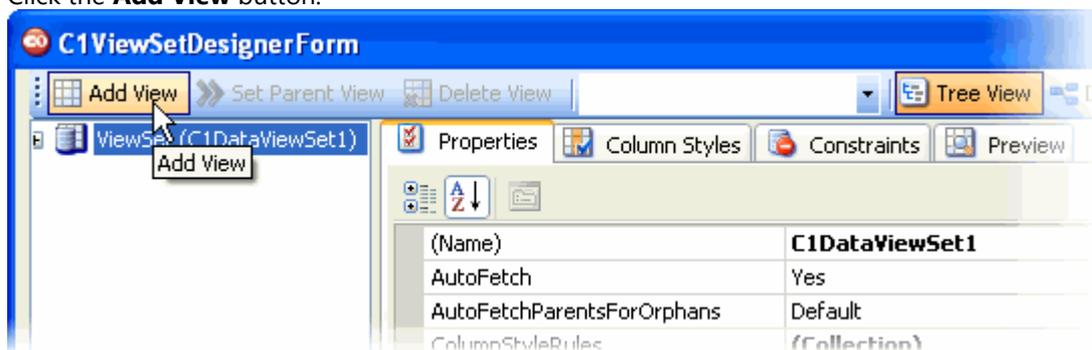
You have successfully added the **C1DataViewSet** component to your Windows form and connected to a data source. The next topic shows how to define a view.

## Step 2 of 4: Define One or More Views

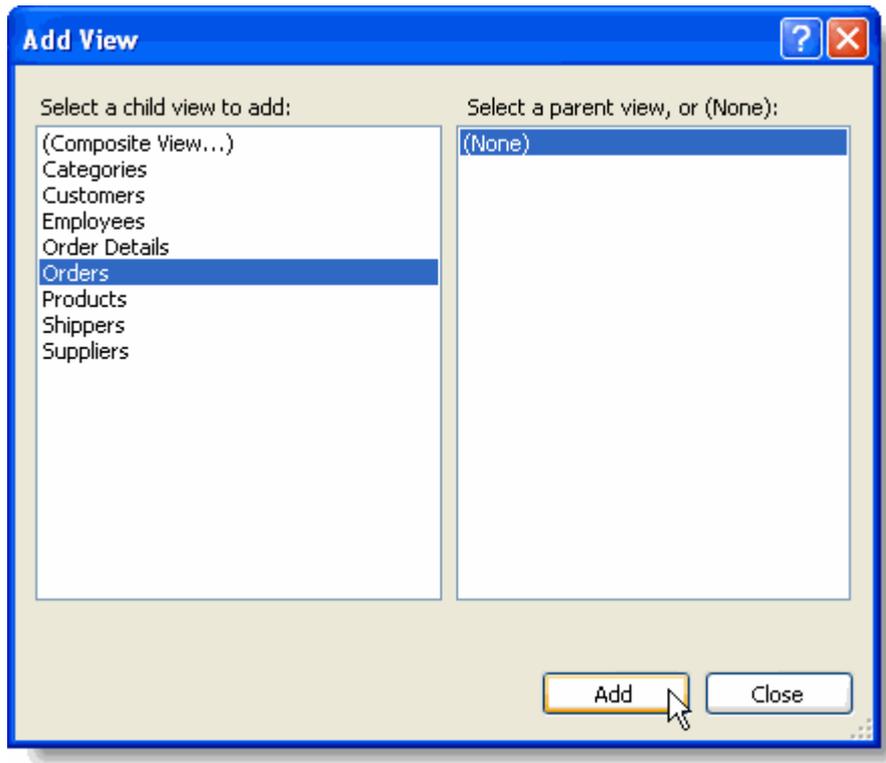
At this point, the **C1DataViewSet** component does not have any views defined. Using the **C1ViewSetDesignerForm**, you can define one or more views at design time. If the underlying data source has table relationships defined, then you can add individual tables accordingly.

To define a data table, for example, **Orders**, complete the following steps:

1. Click the smart tag (📌) located above the **C1DataViewSet** component to open its **C1DataViewSet Tasks** menu.
2. From its Tasks menu, select **Edit** to edit the **C1DataViewSet**'s structure. The **C1ViewSetDesignerForm** appears.
3. Click the **Add View** button.

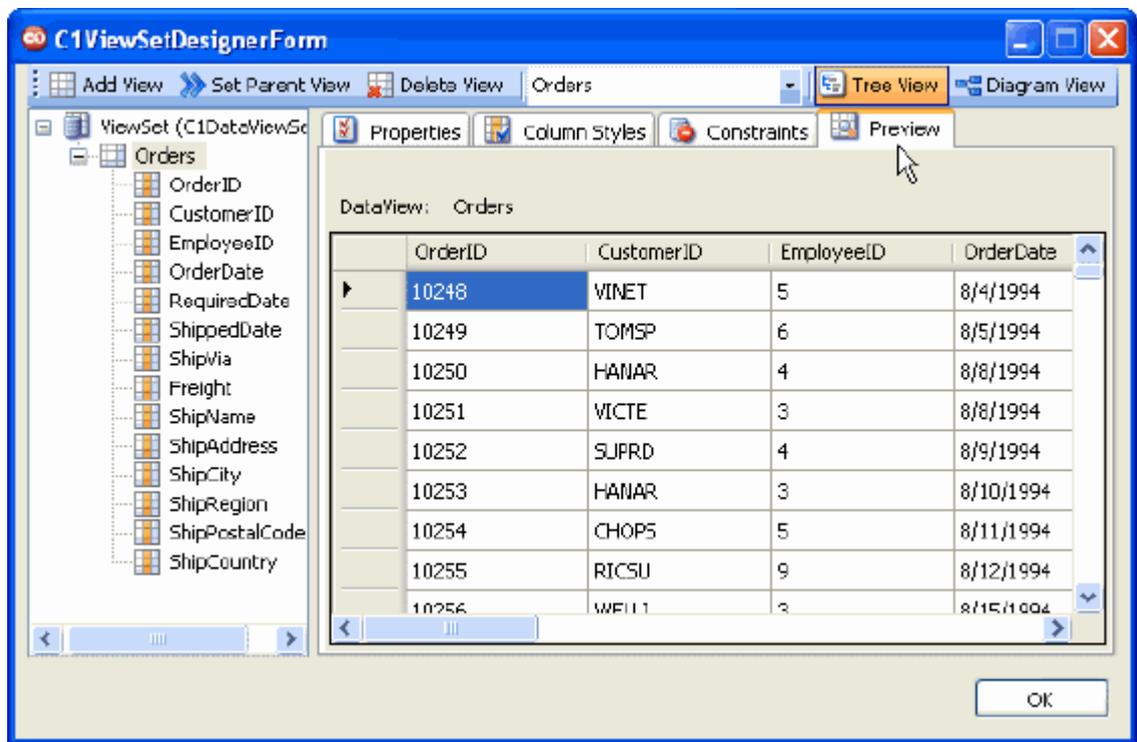


4. In the **Add View** dialog box that appears, select **Orders** from the left pane and then click **Add**.



5. Click **Close** to close the **Add View** dialog box.

The **Preview** tab of the Designer reveals the view that you just created:



**Note:** The left side of the **C1ViewSetDesignerForm** shows the hierarchical **C1DataView** structure, while the right side provides a design surface for editing the selected element, such as a **C1DataView** or one of its **C1ViewColumn** objects (derived from the underlying **DataColumn**). The **Preview** tab is provided for examining the views that will be exposed to bound controls.

6. Click **OK** to close the designer.

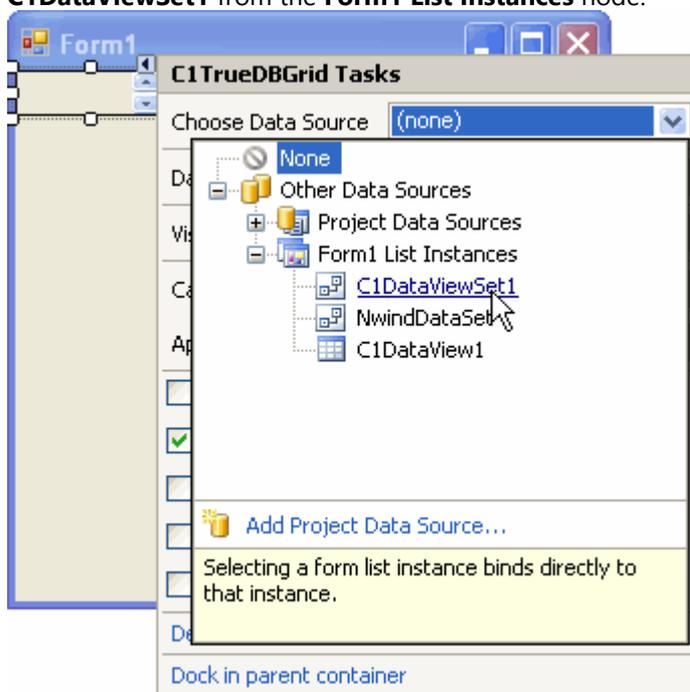
You have successfully defined a data view, **Orders**. The next topic shows how to connect the data table to a grid.

## Step 3 of 4: Connect the View to a Grid Control

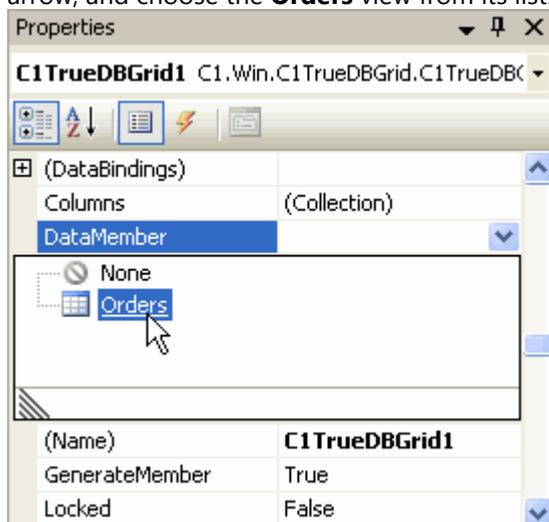
At this point you have added the **Orders** view, but you do not have a grid to show the data. In order to expose the **Orders** view, you have to bind the data view to a control.

To add a grid to your form and connect the data view to the grid, complete the following steps:

1. From the Toolbox, double-click **C1TrueDBGrid** to add it to your form.
2. With the **C1TrueDBGrid Tasks** menu open, select the **Choose Data Source** drop-down arrow and select **C1DataSetView1** from the **Form1 List Instances** node.



3. From the **C1TrueDBGrid Tasks** menu, select **Dock in parent container**.
4. From the Visual Studio **Properties** window, locate the **C1TrueDBGrid.DataMember**, select the drop-down arrow, and choose the **Orders** view from its list.



This binds the data to the grid.

5. Click **Yes** to replace the column layout.

You have successfully added a grid control to your form and connected the **Orders** view to the control. The next topic shows how to run the application.

## Step 4 of 4: Run Your Quick Start Application

Click the **Start Debugging** button to run your application. The following image shows the **ADO.NET DataExtender** Quick Start form after completing each main step in the quick start (steps 1 - 3):



CustomerID	EmployeeID	OrderDate	RequiredDate	Status
Vins et alcools Ch	Buchanan	8/4/1994	9/1/1994	8/16/1994
Toms Spezialität	Suyama	8/5/1994	9/16/1994	8/10/1994
Hanari Carnes	Peacock	8/8/1994	9/5/1994	8/12/1994
Victuailles en stock	Leverling	8/8/1994	9/5/1994	8/15/1994
Suprêmes délices	Peacock	8/9/1994	9/6/1994	8/11/1994
Hanari Carnes	Leverling	8/10/1994	8/24/1994	8/16/1994
Chop-suey Chines	Buchanan	8/11/1994	9/8/1994	8/23/1994
Richter Supermark	Dodsworth	8/12/1994	9/9/1994	8/15/1994
Wellington Importa	Leverling	8/15/1994	9/12/1994	8/17/1994
HILARIÓN-Abasto	Peacock	8/16/1994	9/13/1994	8/22/1994
Ernst Handel	Davolio	8/17/1994	9/14/1994	8/23/1994
Centro comercial	Peacock	8/18/1994	9/15/1994	8/25/1994
Ottilies Käseladen	Peacock	8/19/1994	9/16/1994	8/29/1994
Que Pasa	Peacock	8/19/1994	9/16/1994	8/29/1994

## Congratulations!

You have successfully added the **C1DataViewSet** component to your form, connected to a data source, and defined a view.

## Design-Time Support

**ADO.NET DataExtender** provides visual editing to make it easier to create data views. The following section details each type of support available in **ADO.NET DataExtender**.

### Invoking the Smart Tags

In Visual Studio, the **C1DataViewSet** component includes a smart tag. A smart tag represents a short-cut tasks menu that provides the most commonly used properties. You can invoke the smart tag by clicking on the smart tag (▾) in the upper-right corner of the component. For more information on how to use the smart tag for the **C1DataViewSet**, see [C1DataViewSet Smart Tag](#).

### Design Time Editors

**ADO.NET DataExtender** provides the **C1ViewSetDesignerForm**. For more information about the designer, see [C1DataViewSet Designer](#).

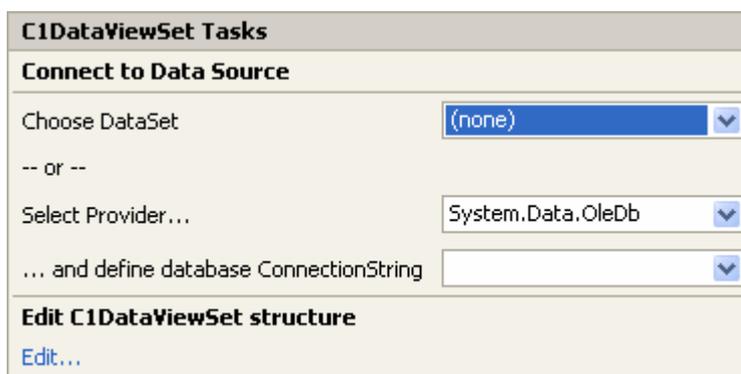
### C1DataViewSet Component Properties

You can access the properties for the **C1DataViewSet** component simply by right-clicking on the component and selecting **Properties** or by selecting the class from the drop-down list box of the **Properties** window. Additionally, you can view the properties in the **Properties** page of the **C1ViewSetDesignerForm**.

### C1DataViewSet Smart Tag

The **C1DataViewSet** component provides quick and easy access to the **C1ViewSetDesignerForm** and data source connections through its smart tag.

To access the **C1DataViewSet Tasks** menu, click on the smart tag (▾) in the upper-right corner of the **C1DataViewSet** component. This will open the **C1DataViewSet Tasks** menu.



The **C1DataViewSet Tasks** menu operates as follows:

### Connect to Data Source

The **C1DataViewSet Tasks** menu lists the following options to connect to a data source:

- **Choose DataSet**

Click the drop-down arrow in the **Choose Dataset** drop-down box to select a data source to connect to data. Note that you must first create a typed or untyped ADO.NET dataset.

- **Select Provider**

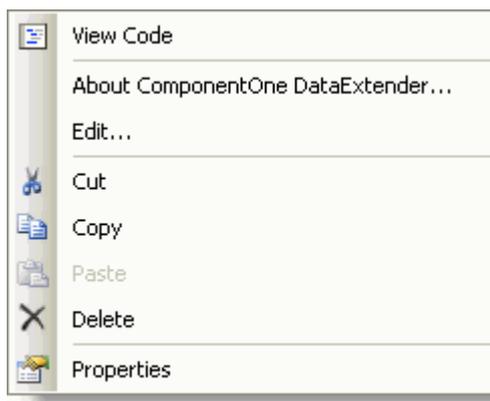
Click the drop-down arrow in the **Select Provider** text box to select a data provider. Then click the drop-down arrow in the **and define database ConnectionString** text box to specify a ConnectionString.

## Edit C1DataViewSet structure

Clicking on **Edit** opens the **C1ViewSetDesignerForm**. For more information on the designer, see [C1DataViewSet Designer](#).

## C1DataViewSet Context Menu

The **C1DataViewSet** component provides a context menu for additional functionality to use at design time. Right-click on the **C1DataViewSet** component to open the following context menu:



## About DataExtender

Clicking on the About DataExtender item displays the About DataExtender dialog box, which is helpful in finding the ADO.NET DataExtender version number and online resources.

## Edit C

Clicking on the Edit item opens the C1ViewSetDesignerForm. For more information on the designer, see [C1DataViewSet Designer](#).

## C1DataViewSet Designer

To view the **C1ViewSetDesignerForm**:

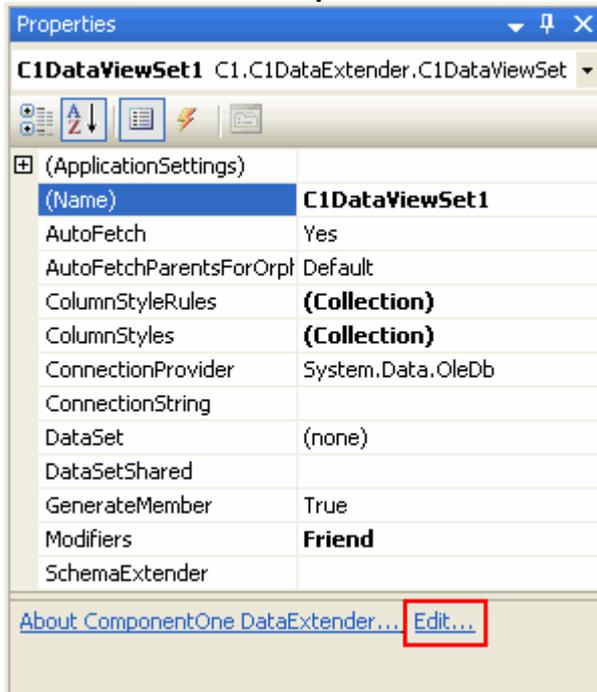
- Click on the smart tag (📌) in the upper-right corner of the C1DataViewSet component and select **Edit**. See [C1DataViewSet Smart Tag](#) for more information.

OR

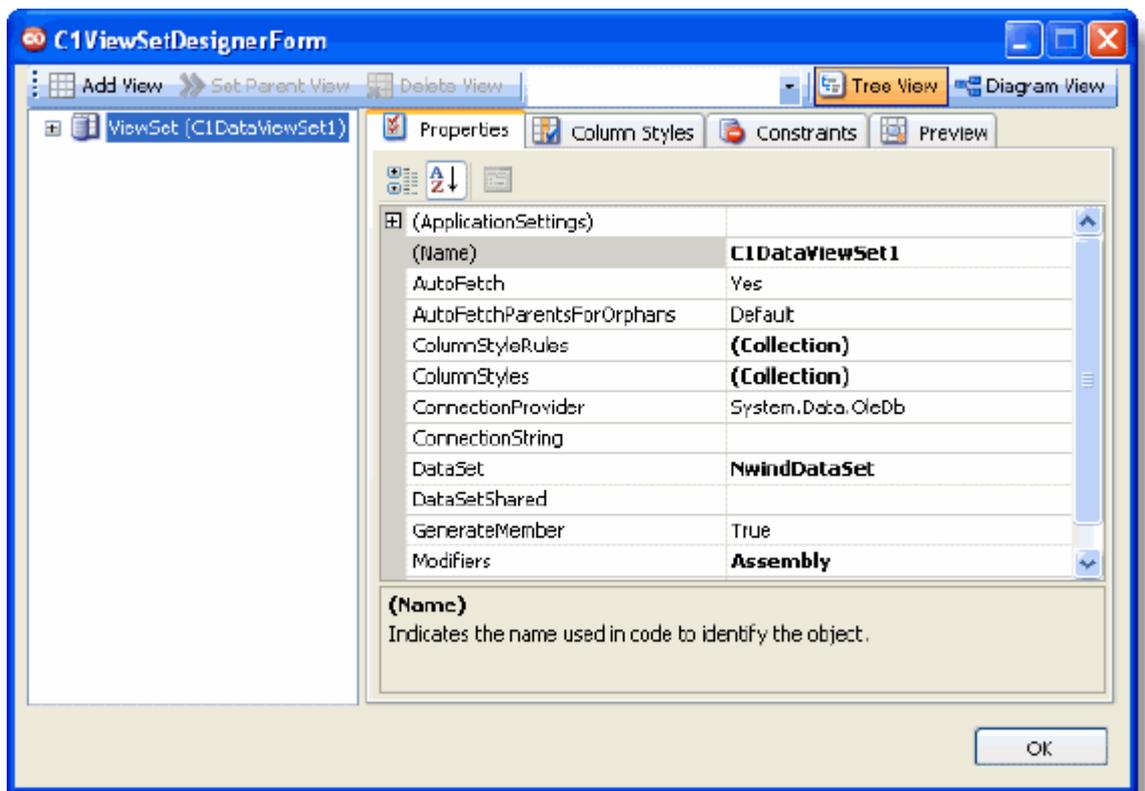
- Select **Edit** from the **C1DataViewSet** Context Menu. See [C1DataViewSet Context Menu](#) for more information.

OR

- From the Visual Studio **Properties** window, select the **Edit** link.



The following designer appears:



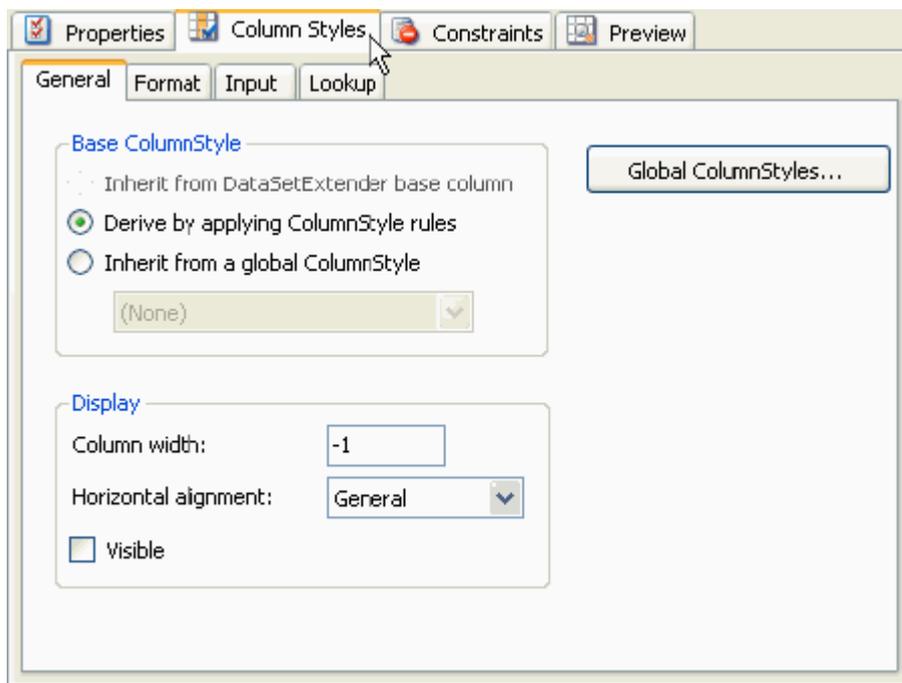
## Properties Tab

The designer's **Properties** tab lists the C1DataViewSet component's properties.

## Column Style Tab

The designer's **Column Styles** tab lists different elements of column data representation that can be defined using ColumnStyle subproperties, for example, Format, EditMask, and Visible.

Here is the **Column Styles** page:

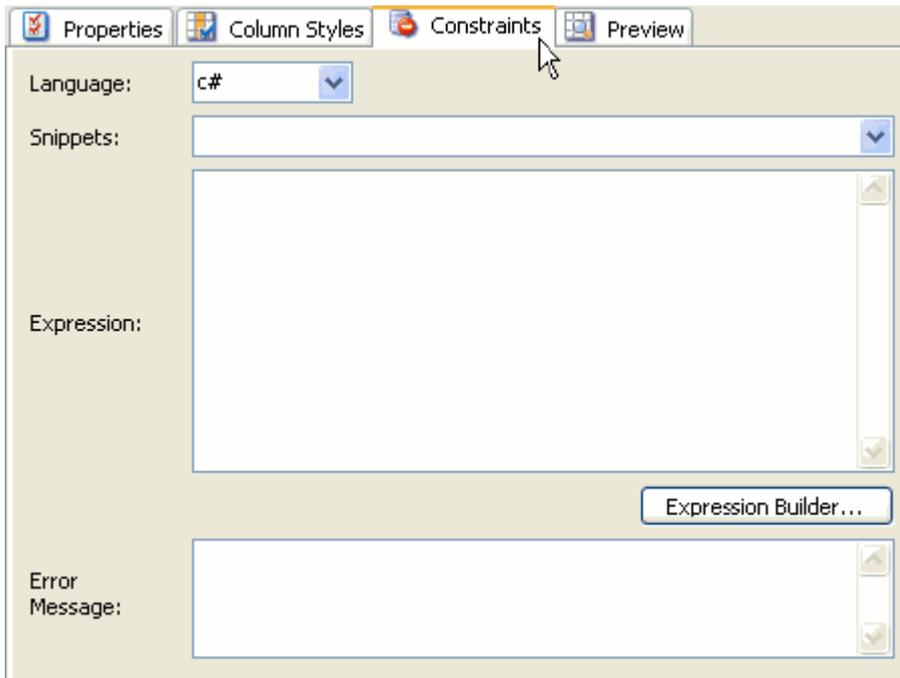


Select the **General**, **Format**, **Input**, and **Lookup** tabs to modify the column styles.

## Constraints Tab

The designer's **Constraints** tab allows you to define row and column level constraints declaratively, that is, you can specify a logical expression evaluating validity of row or column data as a property value of the corresponding row or column.

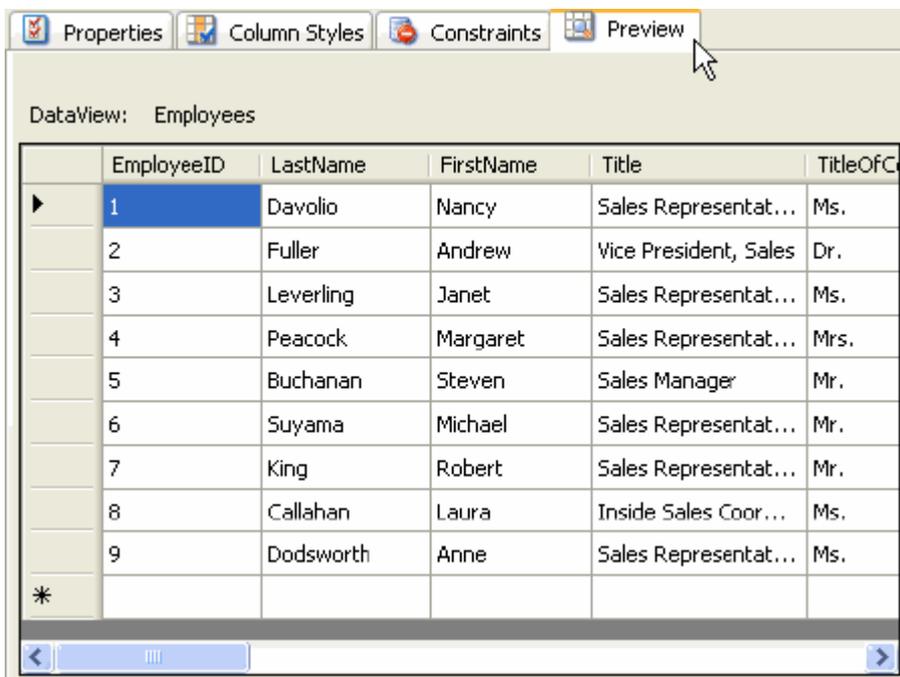
Here is the **Constraints** page:



## Preview Tab

With the time-saving designer interface, the preview pane enables you to preview changes made to the component without having to run the program to see the changes.

Here is a preview of the **Employees** data view:



## Working with C1DataViewSet

A single **C1DataViewSet** component on a Form represents a set of data views based on DataTables from an ADO.NET dataset, with master-details relationships between them. The **C1DataViewSet** component also provides clear navigational and data changing event handlers for each view defined in a view set. Additionally, it performs automatic data fetching and committing, or updating, to a server in only one method call. This approach makes the Form's data model definition and data manipulation highly manageable and easy to use.

To define a data model used in a Form, you have to add a C1DataViewSet component to a Form and choose one of the following data sources:

- Typed ADO.NET Dataset
- Untyped ADO.NET Dataset
- A connection string referencing a particular database

After defining a data source, you must define a set of data views, or C1DataView objects, which can be done using the **C1ViewSetDesignerForm**.

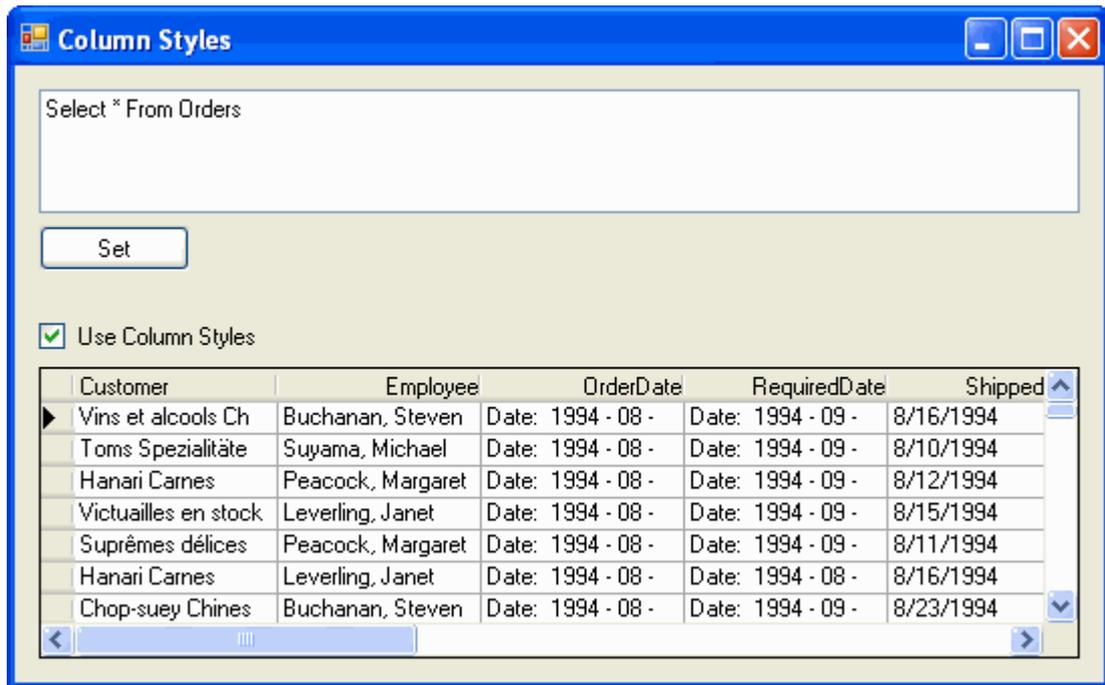
**C1DataView** is a customized view of a DataTable, either single, multiple, or joined tables, designed for UI controls for binding, sorting, filtering, searching, navigating, and editing. Each **C1DataView** object can represent data from either a single table of a data source (simple view) or from a row set constructed as a join of multiple tables from a data source (composite view), similar to a SQL server view.

**C1DataView** objects can be connected with master-detail relationships between them, which provide detail views with automatic child row filtering and foreign key inheritance.

A UI control can then be bound to a specific **C1DataView** of a **C1DataViewSet** by setting its **DataSource** property to the **C1DataViewSet** and setting its **DataMember** property to the **C1DataView** name.

**C1DataViewSet** also provides events that allow custom processing of row navigation and data changing occurrences in each individual **C1DataView**.

Once a **C1DataView** object has been specified within a **C1DataViewSet**, you can run the program. In the following example, the Form contains a grid bound to a **C1DataView** object that represents the **Orders** DataTable from a typed dataset.



The data is automatically fetched from the server for the C1DataView; no special effort is required. After the user makes corrections to the client data, the changes made in the C1DataView, or multiple C1DataViews, can be committed to a database server with a single method call. When performing database updates, C1DataViewSet takes into account the types of corrections performed on data rows and the interrelations between data tables in order to commit data rows in the correct order, thus preventing possible conflicts on a server. As a part of the database updating process, C1DataViewSet retrieves auto-generated primary key values (and additional row values which may be changed on the server) back to the client rows, and correctly updates the foreign key values of child rows.

## C1DataViewSet Data Sources

C1DataViewSet can represent data from typed or untyped ADO.NET dataset or a connection string. Here are these three types of data sources briefly introduced:

- **Typed ADO.NET Dataset**

The DataSet property connects C1DataViewSet to a typed dataset component that has been placed on a Form. In this case, C1DataViewSet retrieves Table Adapters, defined as a part of typed dataset definition, for each DataTable. To fetch data and commit changes back to the server, C1DataViewSet uses the information represented by adapters.

- **Untyped ADO.NET Dataset**

The DataSet property connects C1DataViewSet to an untyped dataset component that has been placed on a Form.

In this case, C1DataViewSet is not able to fetch data and commit changes back to the server. You should create DataTable objects, and manually fill and update their data back to the server. Note that the **C1DataView(s)** defined in a C1DataViewSet are only able to represent the data according to their definitions.

- **Connection String**

In this case, the ConnectionString and ConnectionProvider properties define a connection to a database. The ConnectionString property defines a database that is used as datasource. The ConnectionProvider property defines the name of the ADO.NET Data Provider that is used when connecting to the database. C1DataViewSet retrieves database schema information, automatically creates an internal ADO.NET dataset, which does not appear on the form, and works against it. This dataset is still accessible at run time through the DataSet property.

To connect to these data sources you must have a C1DataViewSet component on your form. The following sections explain the necessary steps you must take to make each connection.

## Connecting a C1DataViewSet Component to a Typed or Untyped ADO.NET Dataset

Connecting a C1DataViewSet to a typed or untyped ADO.NET dataset involves the following basic operations:

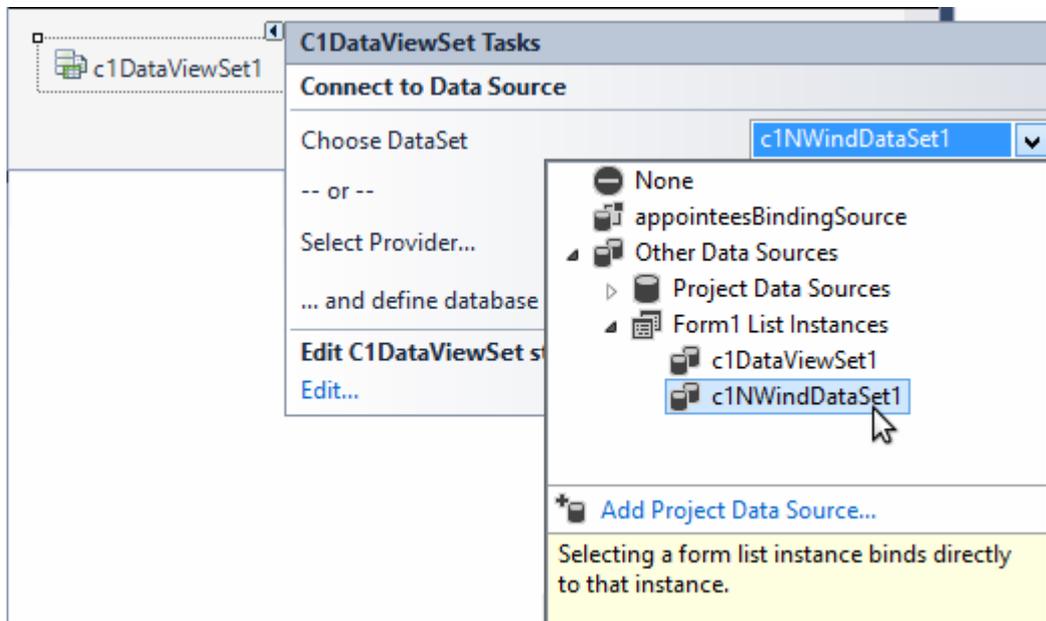
1. Create a typed or untyped ADO.NET dataset. Consult the Microsoft Visual Studio Documentation for more information on this process. In this example, we created a typed ADO.NET dataset, C1NWindDataSet1, based on the Northwind database included with this product.
2. From the Toolbox, add a C1DataViewSet component to your form.

 **Note:** If you have not already added the C1DataViewSet component to the Toolbox, see [Adding ADO.NET DataExtender Components to a Project](#).

The C1DataViewSet properties appear in the **Properties** window in the lower right pane of the Visual Studio IDE.

3. Set the DataSet property to **C1NWindDataSet1**.

You can also click the smart tag (📌) above the C1DataViewSet component to open the **C1DataViewSet Tasks** menu, where you can select the DataSet from the **Choose DataSet** drop-down list, as shown in the following image:



## C1DataView Definitions

Each C1DataView of a C1DataViewSet can represent data from a single **DataTable** of an underlying ADO.NET dataset (simple view), or a row set constructed as a join of multiple DataTables (composite view). The latter case is similar to SQL Views, except that the join is constructed against client data.

Once a data source has been specified for C1DataViewSet, you can define C1DataView by assigning a special SQL SELECT-like statement to the Definition property. The formal syntax of this statement is as follows:

```
SELECT <column list>FROM <table1> [AS <table1_alias>] [ [INNER|OUTER] JOIN <table2>
[AS <table2_alias>] [ON <relation_condition>] ]*
WHERE <condition>
```

## Simple view definition

To define a view representing data from a single **DataTable** named **Orders**, for example, you would use the following statement:

```
SELECT * FROM Orders
```

## Composite view definition

To define a composite view joining data from the **Orders** and **Order Details** DataTables, you would use the following statement:

```
SELECT * FROM Orders JOIN [Order Details]
```

This is a simplified version of the statement. The full form of this statement can be written as:

```
SELECT * FROM [Orders] OUTER JOIN [Order Details] ON Orders.OrderID = [Order
Details].OrderID
```

The *ON Orders.OrderID = [Order Details].OrderID* construction explicitly defines *DataRelation* connecting the specified DataTables that will be used in a join. Note that expression in the ON clause cannot be an arbitrary one, but it must explicitly map on a join condition of one of the *DataRelations* connecting the joining DataTables. If this clause is omitted, then an appropriate relation is determined automatically. Note that if there is no relation connecting the tables, then the statement is treated as incorrectly defined.

*JOIN*, as in a SQL SELECT statement, can be OUTER or INNER, with the same semantics as used in SQL. If this keyword is omitted, *JOIN* is treated as OUTER. In contrast to SQL, the outer joins in a *C1DataView* definition are always LEFT, and there is no way to define RIGHT or FULL joins.

You have the option of entering these definition statements manually at design time or you can use the *C1DataViewSet*'s **Definition Statement Builder**. For more information on using the **Definition Statement Builder**, see [Working with the C1DataView Definition Statement Builder](#).

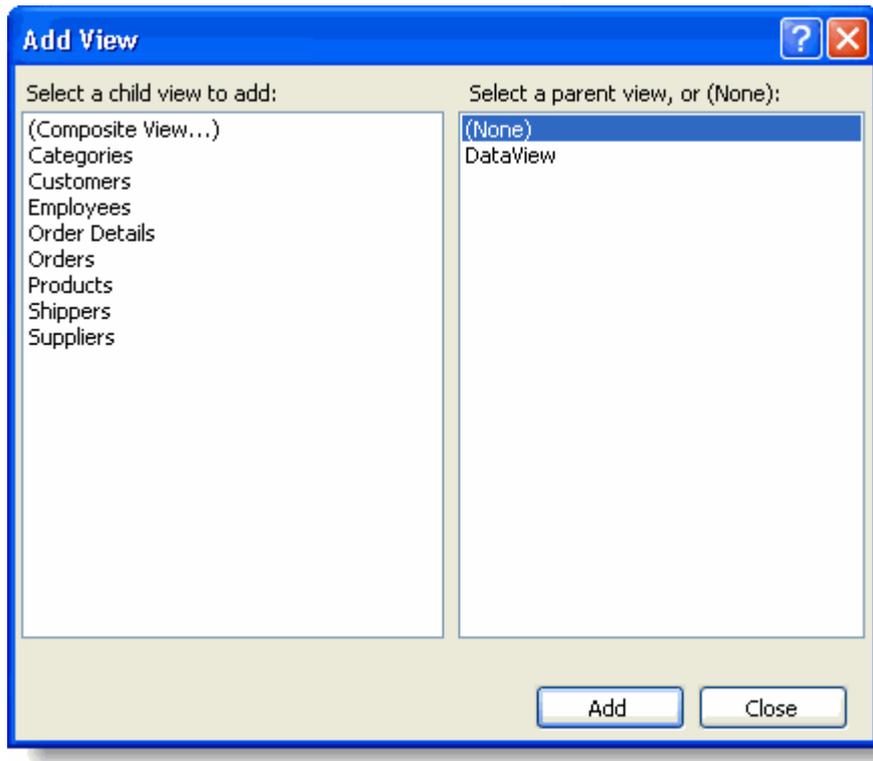
## Simple View Definition

Simple tables are the basic data objects in a schema. We distinguish between simple and composite tables, although both are **DataTable** objects and for the most part can be used interchangeably. Composite tables combine multiple simple tables in a single **DataTable** object. A composite table row contains fields from different tables, for details see [Composite View Definition](#).

## Creating a Simple View Definition

To create a simple view definition, complete the following steps:

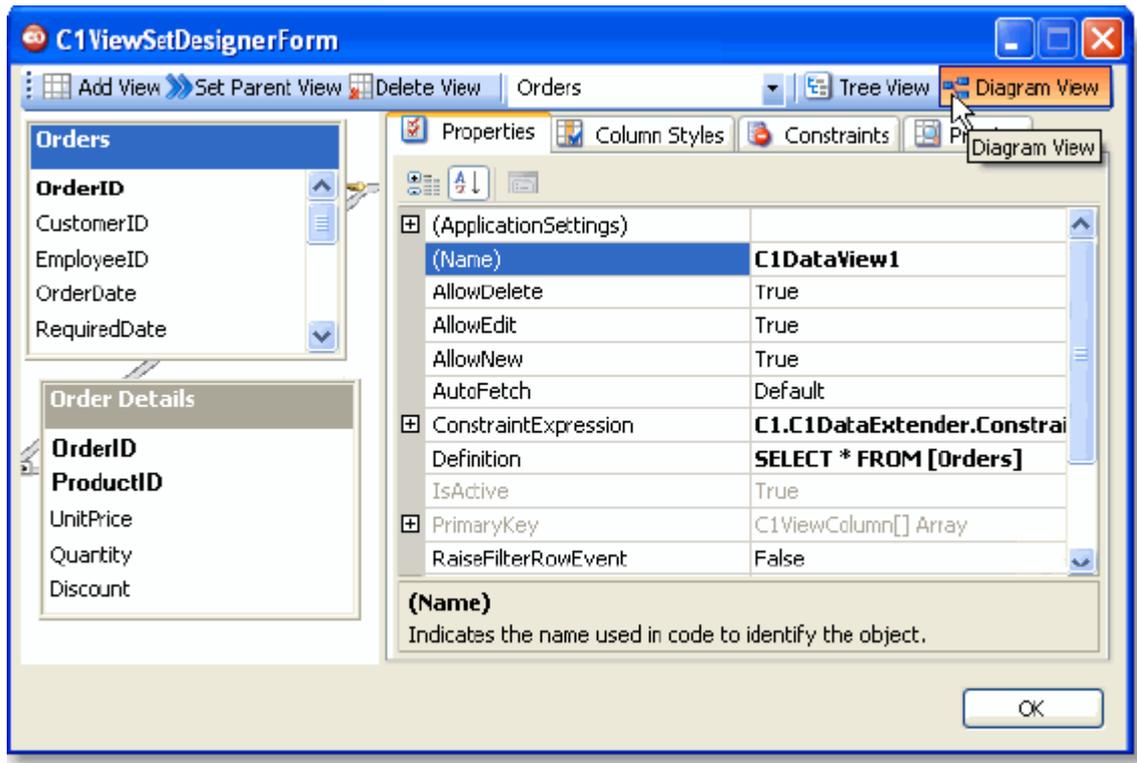
1. Click the smart tag (📌) above the *C1DataViewSet* component and select **Edit** from its tasks menu. The **C1ViewSetDesignerForm** appears.
2. Click the **Add View** button. The **Add View** dialog box opens.
3. Select a parent view or **None** under **Select a parent view** from the list of available views in the right pane.



4. Then select a table from the list of available dataset tables under **Select a child view to add**. Click **Add**, and close the **Add View** dialog box.
5. The definition statement appears in the Definition property text box of the **C1ViewSetDesignerForm** dialog box.

## Using the Diagram View to Create a Simple View Definition

Alternatively, you can select the **Diagram View** button to create a simple view definition:



While working in **Diagram View**, the user can perform the following tasks:

- Select a view by clicking on the view "box" title.



- Select a column by clicking on the column in the view "box".
- Select a relation by clicking on the relation in the view "box".
- Move and resize view "box" using the mouse.

## Composite View Definition

A composite view represents data from multiple joined tables as a single rowset, similar to a view in SQL server. Each view row, or **C1ViewRow** object, doesn't actually store column values; it only references **DataRow** objects from underlying base **DataTable** objects. These view rows are accessible through the **C1ViewRow.BaseRows** collection. The base **DataRow** can be retrieved from this collection by an alias of a corresponding **DataTable**. Note that in the case of an OUTER join it's possible that there is no base **DataRow** for a certain table; instead, a null value is stored in the **C1ViewRow.BaseRows** collection for this table alias.

A join between two **DataTable** objects can have slightly different semantics and is classified in **ADO.NET DataExtender** as the following:

- **Main**

Relation between the left and right table of a join is one-to-many.

- **Lookup**

Relation between the left and right table of a join is many-to-one.

We'll call the right table of the Main join the **Main** table and the right table of the Lookup join the **Lookup** table. The first table of a view definition statement (which has no left joined table) is always treated as **Main** table.

The left table cannot have more than one right table with relations to the Main join. If this requirement is violated then the view definition statement is considered incorrectly defined. On the contrary, the left table can have any number of "Lookup" joins with the right tables.

For example, consider a join between the **Customers** and **Orders** tables. Note that the relation between **Customers** and **Orders** is one-to-many. If you set the view definition as follows:

```
SELECT * FROM Customers JOIN Orders
```

then you have the Main join between **Customers** and **Orders**. However, if you define it as:

```
SELECT * Orders FROM JOIN Customers
```

then you get the Lookup join.

**Main/ Lookup** table semantics affect the row deletion behavior of C1DataView. When you perform a C1ViewRow deletion, only **DataRow** objects that correspond to **Main** tables can be deleted (**DataRow** objects of **Lookup** tables are always kept untouched). In general, the right most Main **DataRow** is always deleted. Suppose that this row is in a **DataTable** named *T*. Then a Main **DataRow** which is left to the already deleted row is investigated if this **DataRow** has no other related **DataRow** objects from **DataTable** *T*, then this row is deleted as well and this process is repeated for the next left row. Otherwise, if this row has related rows from *T*, it leaves the row untouched.

When you add a new C1ViewRow, **ADO.NET DataExtender** can create corresponding base **DataRow** objects and/or reference the existing ones. By default, new **DataRow** objects are created. The special case is when you enter a value of a column that represents a foreign key column of base DataTable and a parent DataTable that is referenced by this foreign key is in the set of base DataTables joining by C1DataView. For example, consider the following definition for the C1ViewColumn representing the Orders.CustomerID DataColumn in the view:

```
SELECT * FROM Customers JOIN Orders
```

When a value of such a foreign key column is being set, **C1DataView** first looks whether DataRow with the corresponding primary key value exists in the parent DataTable. If so, the modified C1ViewRow starts to reference those parent DataRow. If not, a new parent DataRow is created and its primary key value is set to equal the entered foreign key value.

If a view definition has at least one **INNER join**[INNER join means that only matching records present in both views (tables) will appear in the result] , new base **DataRow** objects will be created for each base **DataTable** by default. In the case when all joins in the view definition are **OUTER**[OUTER join means that records of the first view (table) will appear in the result even if there are no matching records in the second view (table)] , a new base **DataRow** will be created only if you enter a value for a view column representing that base **DataRow** column. In this case, base rows will be created also for each of the Main **DataTable** objects which are at the left side for this **DataTable**.

When you edit an existing C1ViewRow object, the same rules as for row addition apply.

## Creating a Composite View Definition

A composite table can contain multiple table views based on a single table. In other words, a simple table can occur several times in a composite table diagram. Users can expose multiple related tables as a single rowset by defining a composite view, either visually or using an SQL-like statement.

 For an example showing how to set up a data model for a Windows form and create a C1DataViewSet with a

composite view, see the **C1DataExtender** video, which is available for download from the [ComponentOne HelpCentral Videos](#) page.

To create a composite view definition that joins data from multiple tables together in one view, complete the following steps:

1. Click the smart tag (📌) above the C1DataViewSet component and select **Edit** from its task menu. The **C1ViewSetDesignerForm** appears.
2. Click the **Add View** button. The **Add View** dialog box appears.
3. Select a parent view from the right pane, and select **Composite view** from the list of child tables.
4. Click **Add**.

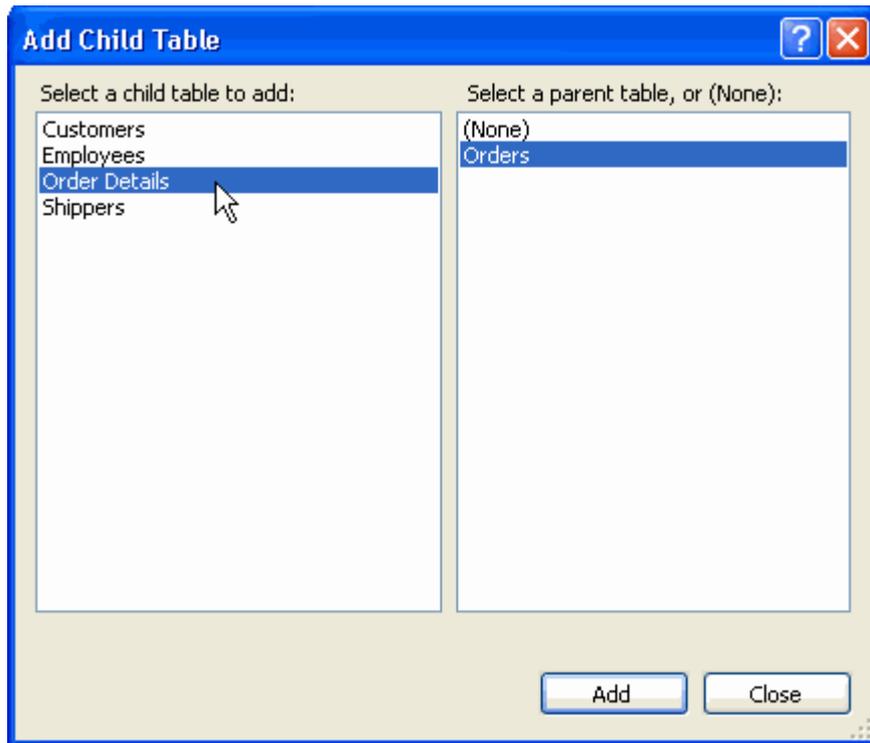
The **C1DataView Definition Statement Builder** opens. The **Diagram View** represents a view definition as a diagram, which allows you to add tables and define table and relation properties.

 **Note:** Alternatively, you can access the **C1Data View Definition Statement Builder** by clicking the **ellipsis** button  next to the Definition property.

5. Click the **Add Child Table** button. The **Add Child Table** dialog box appears.
6. Select **Orders** from the list of available tables under **Select a child table to add**.

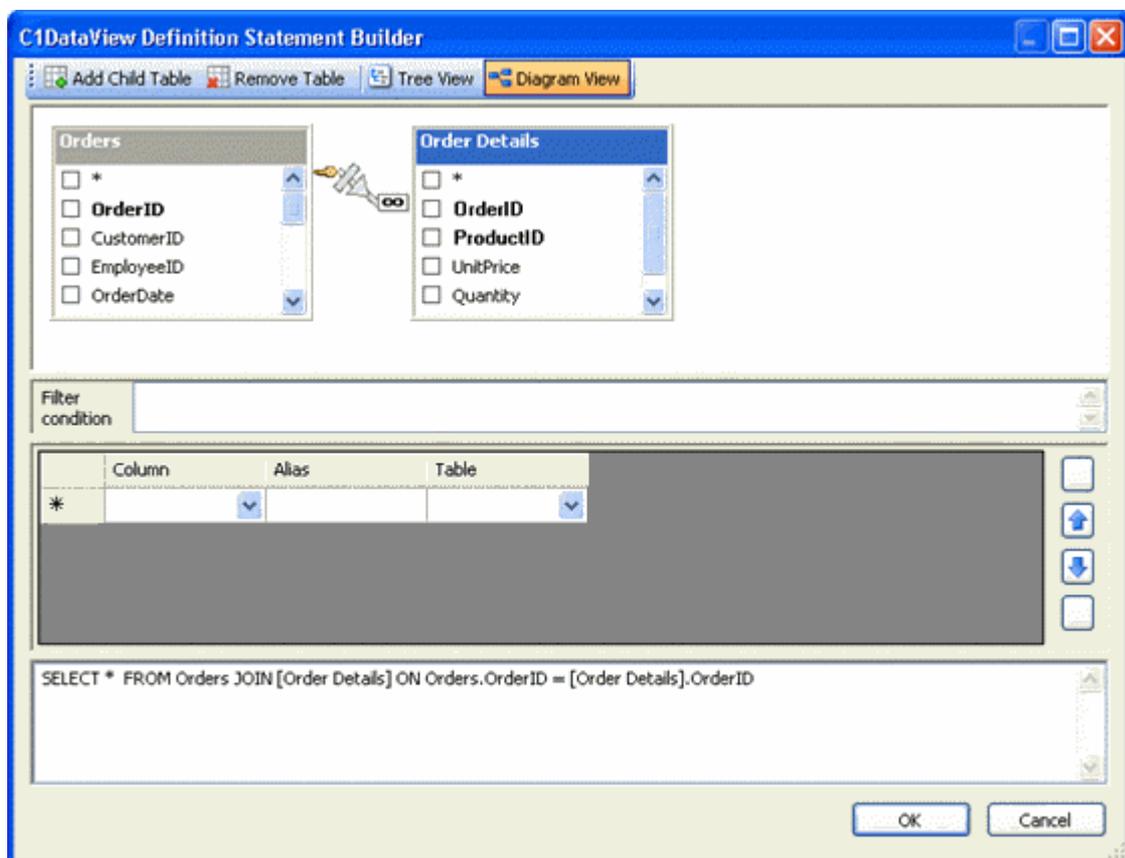


7. Click **Add**.
8. With **Orders** selected as the parent table in the right pane, select **Order Details** from the list of tables under **Select a child table to add**. In this example we are creating an OUTER join, which is selected by default.



9. Click **Add**, and then **Close**.

The **C1DataView Definition Statement Builder** now contains the definition we just created. It does not contain the OUTER keyword, so JOIN is treated as OUTER.



## C1ViewColumn Definitions

**ADO.NET DataExtender** allows *defining* of different aspects of visual data representation in ComponentOne UI controls on a data level. For example, you can define, format, or edit mask for a certain column. You also set it up to have a lookup combo box or any other ComponentOne control that supports this feature. Simply by being connected to this column, the ComponentOne control will automatically reflect those definitions.

In order to allow defining of UI related properties, **ADO.NET DataExtender** offers the **C1ViewColumn.ColumnStyle** property. The **C1ViewColumn.ColumnStyle** property contains subproperties, such as Format, EditMask and Visible that make it possible to define different aspects of column data representation in UI.

The current set of controls that support column styles includes **True DBGrid for WinForms**, **FlexGrid for WinForms**, and **Input for WinForms**.

In addition to defining visual data representation, calculations can be defined to determine the final value of C1ViewColumn. The calculations can be defined in three ways:

- By the DataColumnExtender.CalculationExpression property of the **DataColumnExtender** (DataSetExtender column) and C1ViewColumn (C1DataView column) objects. In this case an expression can be written using the syntax of an arbitrary .NET Framework programming language available on your computer.
- By an event handler method of the C1DataViewSet.CalculateColumn event.
- By the **C1DataView Definition Statement Builder** of the Definition property. See [C1DataView Definitions](#) for additional information.

## Defining Column Value Calculations

### Calculation expression

Similar to constraint expressions (see [Row and Column Level Constraints](#) for more information), calculation expressions defined through the C1ViewColumn.CalculationExpression property are treated as the code of some class method. It returns a value of a type appropriate to a column data type and it is expressed in a specified .NET Framework programming language. Such a class has the following members which can be used in the calculation:

Members	Description
row	References a row on which column calculation is performed. If the expression is defined for a C1DataView the data type is C1ViewRow. If the expression is defined for a <b>DataColumn</b> the data type is type System.Data.DataRow.
column	References a column on which the value is calculated. If the expression is defined for a C1ViewColumn the data type is C1ViewColumn. If the expression is defined for a <b>DataColumn</b> in DataSetExtender the data type is System.Data.DataColumn.
baseValue	Contains the value before the calculation is applied which is the base value of a column.

### Column value transformation process

In general, when a C1ViewColumn value in a certain C1ViewRow is being retrieved, the following chain of column value transformations occurs:

- If **C1ViewColumn** represents a certain **DataColumn**:
  1. A value of base **DataColumn** is retrieved.
  2. A calculation expression (defined in the **DataColumnExtender.CalculationExpression** property) of the corresponding **DataColumnExtender** of **DataSetExtender** is evaluated. The value of the **baseValue**

property accessible from this expression is set to the value produced in step 1.

- If **C1ViewColumn** is calculated (that is, defined through expression in the View Definition statement):
  3. A value of C1ViewColumn is calculated according its expression definition.
- In any case:
  4. A calculation expression defined in the CalculationExpression property of C1ViewColumn is evaluated. The **baseValue** property accessible from this expression is set to a value produced either in step 2 or step 3 (depending on the kind of C1ViewColumn).
  5. If the RaiseCalculateColumnEvents property value is set to **True** then the CalculateColumn event is triggered. The value of event arguments **Value** property is set to a value produced in step 4; this property value can be changed in the event handler code and it becomes a value of the C1ViewColumn.

 **Note:** A calculated column value is not stored in the base **DataRow**, the calculation is just a function on a base value stored in the row.

## Defining Column Styles

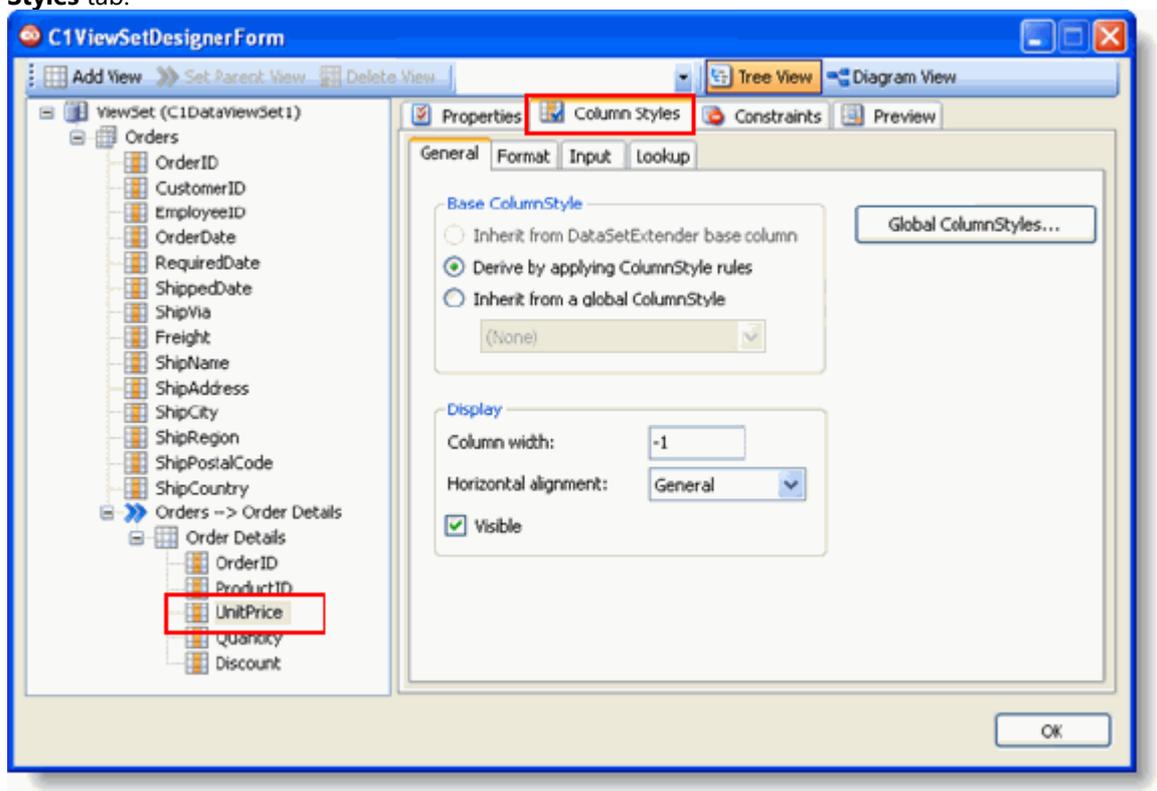
Different elements of column data representation in UI can be defined using the ColumnStyle sub properties such as:

- Format
- EditMask
- Visible

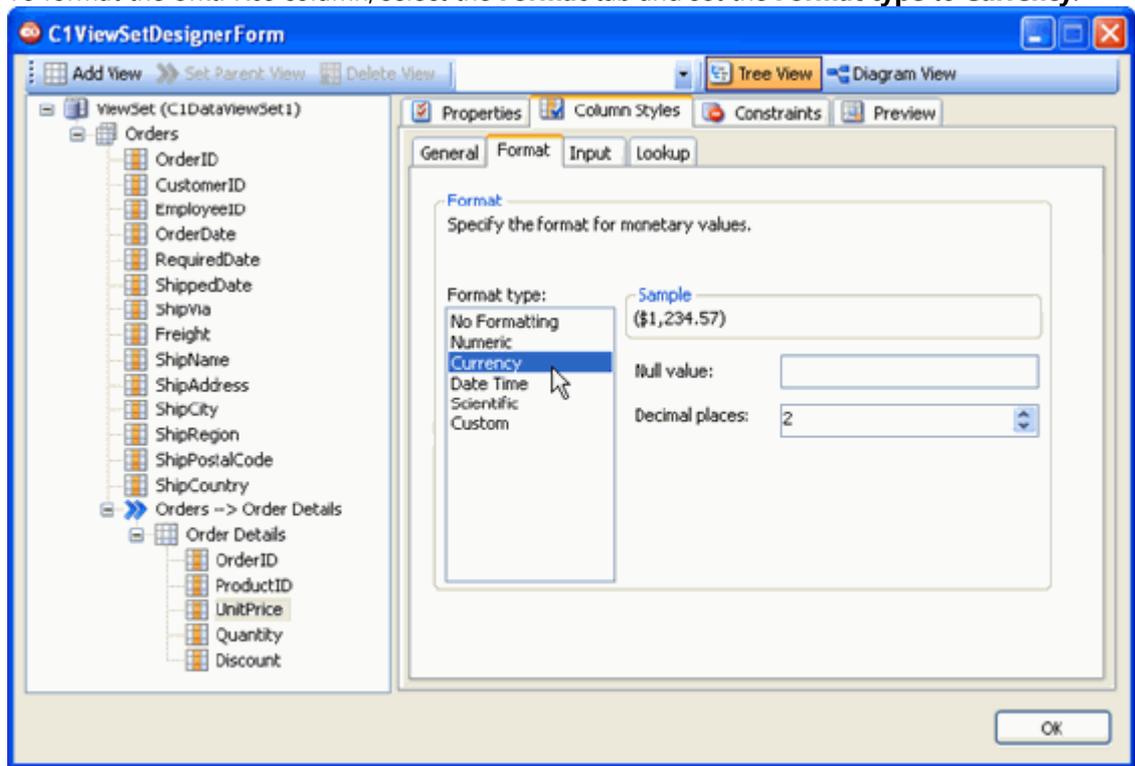
## Editing column styles:

You can easily edit each column's style by means of the **Column Styles** tab located in the **C1ViewSetDesignerForm**. Editing a column style involves the following basic operations:

1. From the **C1ViewSetDesignerForm**, select the *UnitPrice* column from the left pane and then click the **Column Styles** tab.



- To format the *UnitPrice* column, select the **Format** tab and set the **Format type** to **Currency**.



- Run the application and notice that the *UnitPrice* column is formatted accordingly.

## Row and Column Level Constraints

C1DataView objects, constituting C1DataViewSet, allow you to define row and column level constraints declaratively, that is, you can specify a logical expression evaluating validity of row or column data as a property value of the corresponding row or column. In order to make these expressions powerful, **ADO.NET DataExtender** allows you to use syntax from any .NET Framework programming language available on your computer. Thus, you are able to write multi-line expression code which includes logical branches, loops and so on. When defining row and column level constraints, consider the following:

- Row level constraints are evaluated at the end of row editing (when the EndEdit method is being called on a row).
- Column level constraints are evaluated in an attempt to set new value to a certain column of a row.
- If a constraint's expression is evaluated to **False**, it is considered violated.

Constraints can be defined both in C1DataViewSet (in C1DataView and constituting C1ViewColumn objects) and in the **DataSetExtender** (for table and column items), with the place of constraint definition influencing its semantics. Constraints defined in C1DataViewSet are evaluated only when changes are being made through a C1DataView row, or C1ViewRow object. The changes performed through the **DataTable** rows of an underlying dataset are not checked by these constraints. A constraint expression is based on a C1DataView row's values in this case, but not on values of base **DataTable** rows.

In contrast, constraints defined in the DataSetExtender are evaluated when changes are made to underlying **DataTable** rows, directly or through C1DataView row (note that changes made to C1DataView row are reflected in underlying **DataTable** rows). In this case, the constraint expression uses values of a **DataTable** row (**DataRow** object).

Any constraint expression is treated as the code of some class method returning a logical value and expressed in a specified .NET Framework programming language. Therefore, expression code must have a line which returns a value from the method (for example, RETURN statement for Visual Basic and C#).

For row level constraint expressions the following class properties are accessible from the expression code:

- row (of type `C1ViewRow` for a constraint defined for `C1DataView` and of type `System.Data.DataRow` for a constraint defined for a **DataTable** in `DataSetExtender`). References a row for which the constraint is checking.

Example expression code in Visual Basic:

```
RETURN row("UnitPrice") * row("Quantity") < 100
```

For column level constraints you should treat expression as a method code of a class with the properties as follows:

- row (of type **C1ViewRow** for a constraint defined for `C1DataView` and of type `System.Data.DataRow` for a constraint defined for a **DataTable** in `DataSetExtender`). References a row for which column constraint is checking.
- column (of type `C1ViewColumn` for a constraint defined for `C1ViewColumn` and of type `System.Data.DataColumn` for a constraint defined for a **DataColumn** in `DataSetExtender`). References a column for which the value is about to be changed.

The new Value (of type `System.Object`) contains a new proposed column value.

Example expression code in Visual Basic:

```
RETURN newValue > 0
```

Constraint expressions are defined through the **ConstraintExpression** property of corresponding classes (`C1DataView`, `C1ViewRow`, **DataTableExtender** and **DataColumnExtender**; **DataTableExtender** and **DataColumnExtender** define **DataTable** and **DataColumn** extended properties in the `DataSetExtender`) with the following nested properties:

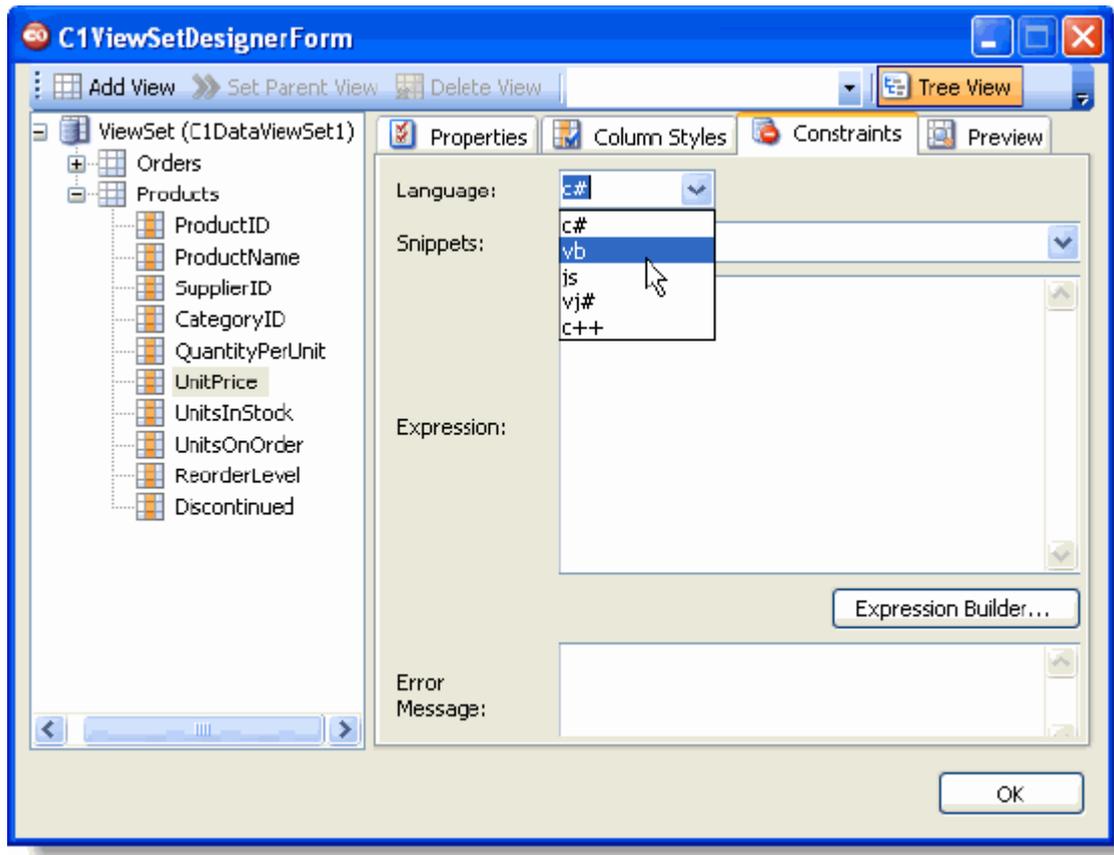
`ConstraintExpression.Expression.Language` defines the name of the .NET Framework programming language, whose syntax is used to define an expression.

- `ConstraintExpression.Expression.Text` defines an expression code.
- `ConstraintExpression.ErrorDescription` allows defining an optional message for exception which will be raised if the constraint is violated.

## Defining Constraints

You can define constraints using the **Constraints** tab located in the **C1ViewSetDesignerForm**. To define a constraint expression for the *UnitPrice* field of the **Products** table, perform the following tasks:

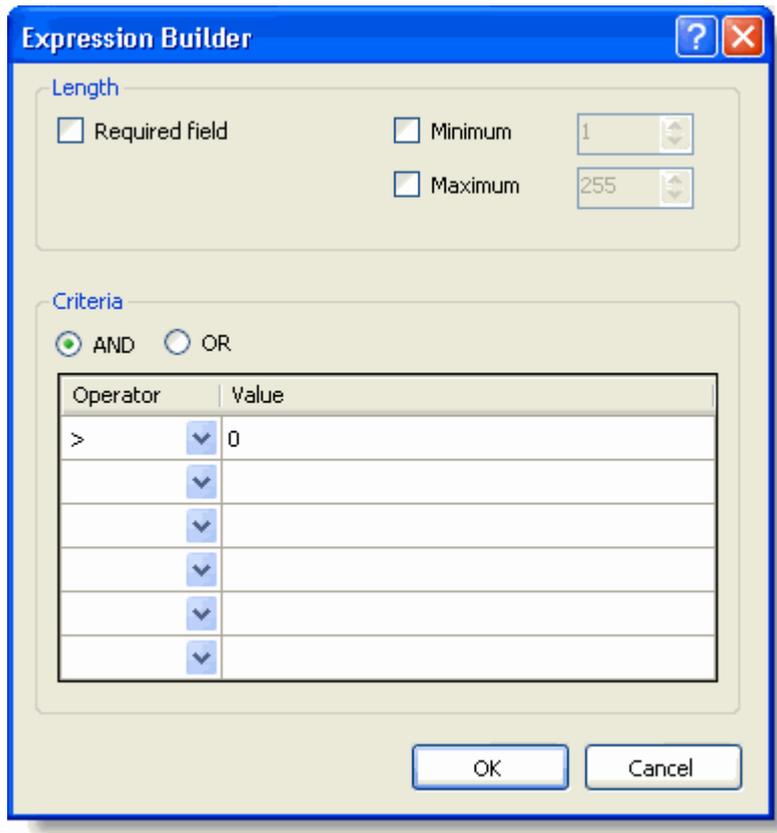
1. Right-click the `C1DataViewSet` component and select **Edit** from its Tasks menu. The **C1ViewSetDesignerForm** appears.
2. From the left pane, select the *UnitPrice* column of the **Products** table and click the **Constraints** tab.
3. Select **vb** from the **Language** drop-down list to specify Visual Basic as the programming language used to define expression.



4. Type the following expression in the **Expression** text box: `Return newValue > 0`

OR

Click the **Expression Builder** button to open the **Expression Builder** dialog box:



Select the greater than operator and enter 0 for the **Value**. Click **OK** to close the dialog box.

5. Type the following string in the **Error Message** box: `UnitPrice must be a positive number`
6. Click **OK** to close the dialog box.

If the user attempts to enter a negative number in the *UnitPrice* column of the **Products** grid, the following error message appears:



Negative numbers are not allowed in accordance with the constraint we defined for the `Products.UnitPrice` field.

## C1DataView Relations

**C1DataViews** of **C1DataViewSet** can be set up to operate either as independent rowsets or to be involved in a master-detail relationship. In the latter case, the behavior of the detail view has the following specifics:

- The set of rows accessible by means of detail view (that is, represented to bound controls and programmatically in the **C1DataView.Rows** collection) is restricted by the rows which are children for the current record of the master view. Note that the current view record is programmatically accessible through the **Current** property and can be changed through the **Position** property. You can still retrieve and use the **CurrencyManager** for these purposes, but **C1DataView** provides everything you need, making accessibility much easier.
- When a new row is added to a detail view, its foreign key field value is automatically filled with the corresponding primary key value of the current row of the master view (foreign key inheritance).

You can easily add a view as a detail of another view by selecting a master view in the **C1ViewSetDesignerForm** and pressing the **Add View** toolbar button. The master view is one of the **C1DataViews**, either simple or composite, that you create when you click the **Add View** button. See [C1DataView Definitions](#) for additional information on adding **C1DataViews**.

## Creating Table and Column Aliases

You can create aliases to make it easier to work with table and column names. The table and column occurrences in a definition statement are differentiated by their aliases. Aliases come in handy when:

- You want to make the statement in the SQL pane shorter and easier to read.
- You refer to the same table or column more than once, such as in a master-detail relationship.

For example, you can create an alias *o* for a table name **Orders**, and then refer to the table as *o* throughout the rest of the query.

The following statement shows an example of table and column aliases:

```
SELECT o.OrderID AS [Order ID], o.OrderDate AS [Date of Order], od.*
FROM Orders AS o JOIN [Order Details] AS od ON o.OrderID = od.OrderID
```

In this statement the **Orders** table has been given the alias *o*, and the **Order Details** table has the alias *od*. The *OrderID* column has the alias *Order ID* while the *OrderDate* column has the alias *Date of Order*.

To create this statement using the **C1ViewSetDesignerForm**, complete the following steps:

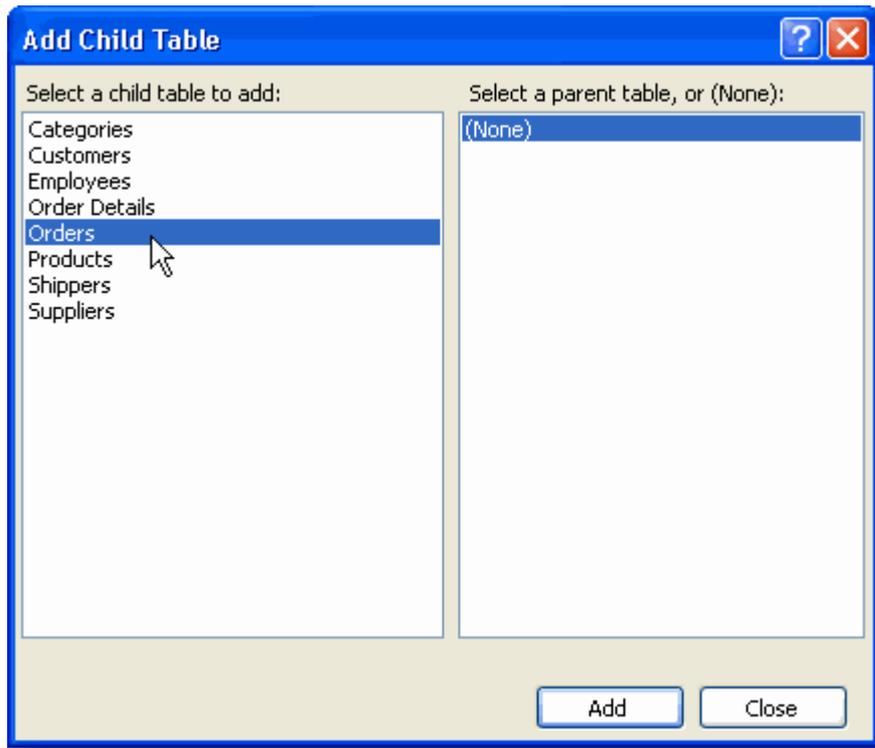
1. Right-click the **C1DataViewSet** component and select **Edit** from its context menu. The **C1ViewSetDesignerForm** appears
2. Click the **Add View** button. The **Add View** dialog box opens.
3. Select a parent view from the right pane, and under **Select a child view to add**, select **Composite view** from the list.
4. Click **Add**.

The **C1DataView Definition Statement Builder** opens.

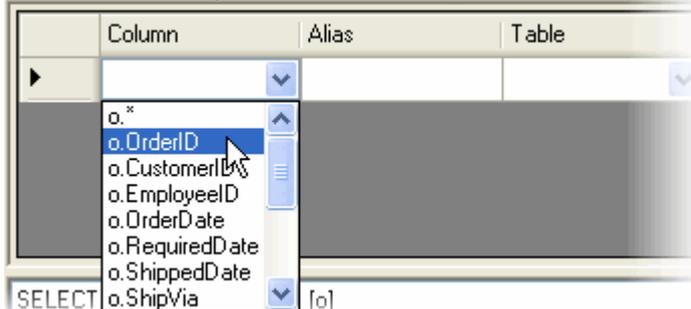
5. Click the **Tree View** button, this represents a view definition as a tree and allows you to add tables.

 **Note:** Alternatively, you can access the **C1Data View Definition Statement Builder** by clicking the **ellipsis** button  next to the **Definition** property.

6. Click the **Add Child Table** button to open the **Add Child Table** dialog box and select **Orders** from the list of available tables. Click **Add**, and then **Close**.



7. Under **Table properties** enter **o** in the **Alias** text box.
8. To create the first part of our definition, select **o.OrderID** from the **Column** drop-down list.



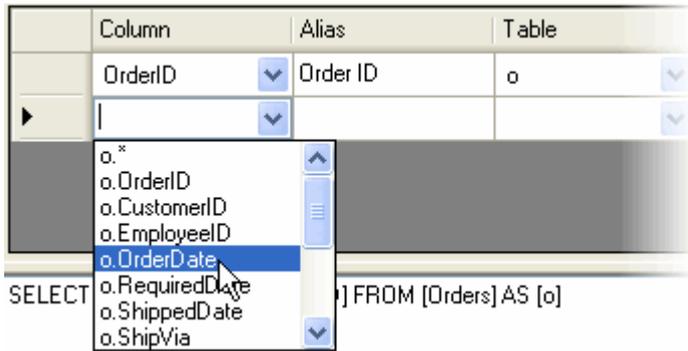
The column qualifier in the SELECT column list can have one of the following forms:

- o \* denotes all columns of all base **DataTable**(s). If it's defined it must be the only column qualifier in the column list.
- o [**<table alias>**].\* denotes all columns from the specified base **DataTable**.
- o [**<table alias>**].**<column name>** [**AS**] [**<column alias>**] denotes the specific  **DataColumn** with optional column alias, with optional indication of the base **DataTable** to which this column belongs.
- o **<expression>** [**AS**] **<column alias>** defines a calculated column, whose value is determined by evaluating the specified expression. The expression can include constants, column references, arithmetical, logical and comparison operations, and IS [NOT] NULL constructions, <match\_expression> [ NOT ] LIKE <pattern\_expression> [ ESCAPE <escape\_character> ] constructions, and subexpressions can be grouped by means of parentheses. Column references of the expression must indicate columns from the base **DataTable**(s), but not the columns from the SELECT list.

9. Enter **Order ID** in the *Alias* column to give the *OrderID* column of the **o** table the alias *Order ID*.

 **Note:** The table automatically changes to **o** in the *Table* column.

10. To add another column to our C1DataView, select **o.OrderDate** from the second row of the *Column* column.



11. Enter **Date of Order** in the *Alias* column.

 **Note:** The table automatically changes to **o** in the *Table* column.

12. In this example, you will create a JOIN to the **Order Details** table. Click the **Add Child Table** button.

13. Select **Order Details** from the **Select a child table to add** list, and enter **od** in the **Table Alias** text box.

14. In the *Column* column, select **od.\*** from the drop-down list. This allows you to select all of the columns from the **Order Details** table for our **C1DataView**.

The table automatically changes to **od** in the *Table* column, and our definition is complete:

```
SELECT o.OrderID AS [Order ID], o.OrderDate AS [Date of Order], od.* FROM
Orders AS o JOIN [Order Details] AS od ON o.OrderID = od.OrderID
```

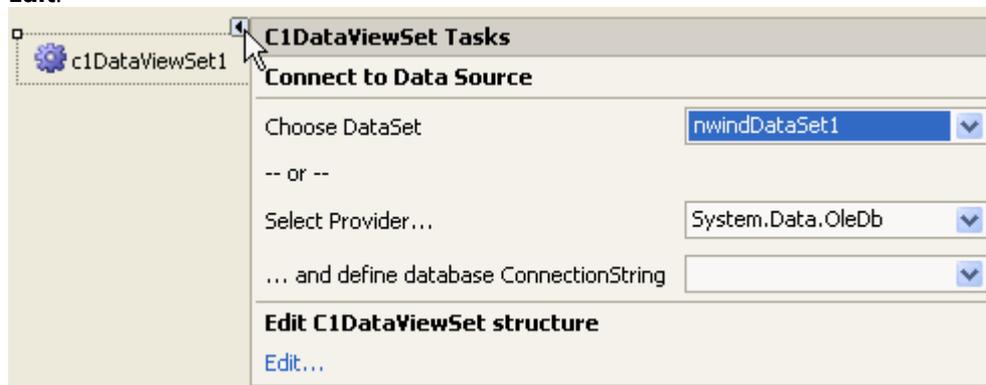
15. Click **OK**, and then close the **Add View** box.

The new **Orders** view is added to the **C1ViewSetDesignerForm**, and the definition is assigned to the *Definition* property.

## Creating a Master-Detail Relationship Between C1DataViews

To create a master-detail relationship between **C1DataViews**, complete the following steps:

1. Click the arrow above the C1DataViewSet component to open the **C1DataViewSet Tasks** menu and select **Edit**.

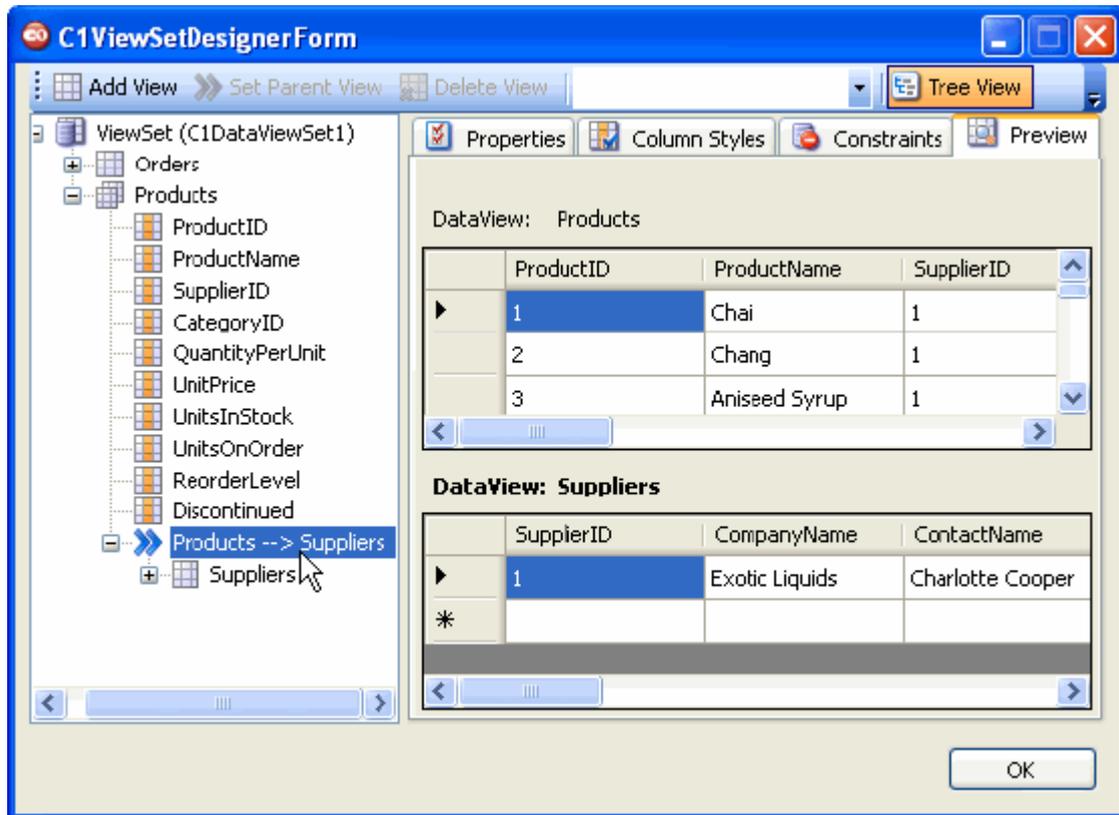


2. In the **C1ViewSetDesignerForm**, select a master view and click the **Add View** button.

3. In the **Add View** dialog box, set up a C1DataView definition, either simple or composite, and click **OK**.

The child, or details, view is added to the editor under the master view. The blue arrows indicate a master-detail relationship; the master view name is on the left of the single arrow, and the detail view name is on the

right. The child view is listed below the master-detail relationship indicator.



In this example, the **Products** view has a child view, **Suppliers**.

## Specifying a Parent View

Two views can be tied by a master-detail relationship only if one of the base DataTables, which has the view on which the definition is based, of the master view has a DataRelation defined in the underlying dataset which connects it to some base **DataTable** of the detail view.

When you specify a parent view for a certain detail view in the **Add View** dialog box of the **C1ViewSetDesignerForm**, you can specify this DataRelation explicitly.

If you choose not to specify it, the C1DataViewSet automatically selects some DataRelation which can connect those views. Namely, this DataRelation determines the fields which are used to filter out detail rows for a current master row.

There are cases when more than one DataRelation can connect views, for example, suppose that we have views with the following definitions:

MasterView SELECT \* Customers JOIN Orders

DetailView SELECT \* Orders JOIN [Order Details]

These views can be tied based on two DataRelation "Customers -> Orders" (Orders from the DetailView here) and "Orders -> Order Details" (Orders from the MasterView here).

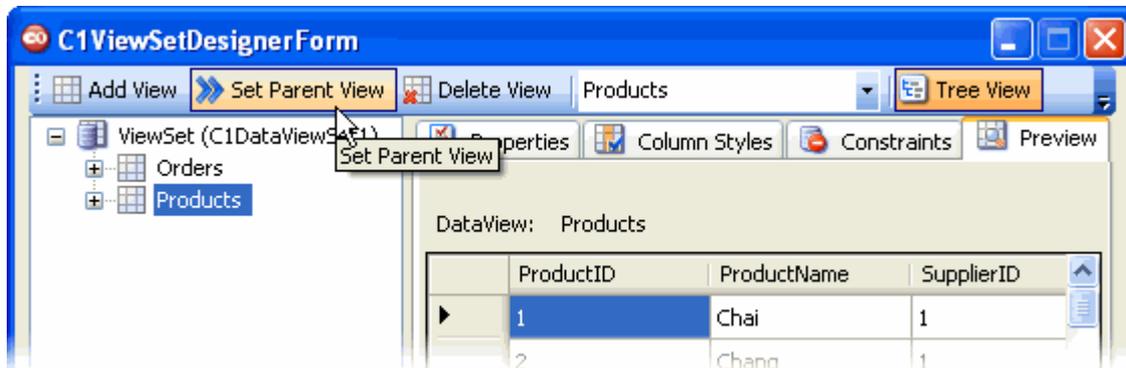
In this case it's reasonable to define a base DataRelation explicitly because C1DataViewSet could select a DataRelation other than the one you are supposed to use.

You can specify a parent view in the **Add View** dialog box when you add a **C1DataView**. To do this:

1. Add a view to your **C1DataViewSet** and specify the definition in the **Add View** dialog box.

- From the left pane select a child view, and under **Select a parent view**, select one of the possible parent view tables from the list.

You can access the **Set Parent View** dialog box from the **C1ViewSetDesignerForm** by clicking the **Set Parent View** button.



Another complex case when more than one **DataRelation** connects views is when some or both of the views involved in a master-detail relationship reference the same **DataTable** more than once (in this case the tables' occurrences in a definition statement are differentiated by their aliases). If the base **DataRelation** corresponds to such a **DataTable**, you should have the ability to specify which occurrence of the **DataTable** must be used (that is, rows of what table alias participate in the relationship). You can do this by explicitly defining an involved **DataTable** alias in the **Set Parent View** dialog box. To do this:

- Click the **Set Parent View** button. The **Set Parent View** dialog box opens.
- Select a **Base DataRelation** that corresponds to the **DataTable** from the drop-down box.



- Specify the Parent **DataTable** alias and Child **DataTable** alias.

**Note:** Set any of the **Parent relation details** drop-down lists to **<auto detected>** if you want the **C1ViewSetDesignerForm** to automatically detect them for you.

## Interaction with an Underlying ADO.NET Dataset

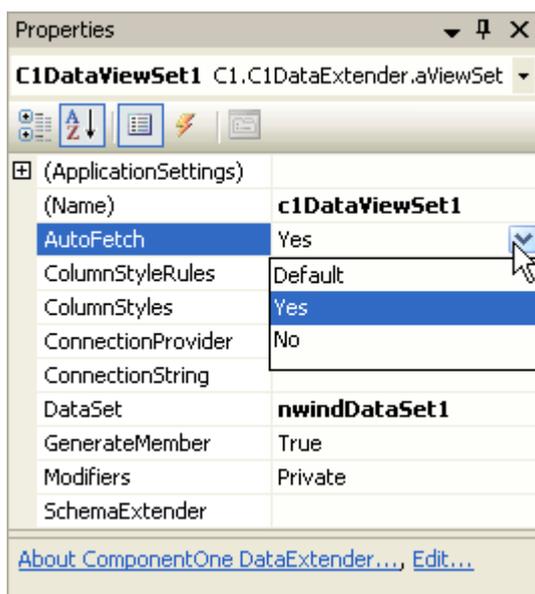
Interaction between `C1DataViewSet` and an underlying dataset, either defined explicitly through the `DataSet` property or internally created based on the `ConnectionString` / `ConnectionProvider` property values, is absolutely clear in the sense that a rowset of `C1DataView` and of base `DataTables` (the ones referenced in the `C1DataViewSet` definition statement) is always in sync.

When `C1DataView` performs [data fetching](#) from the database server, it actually fetches rows to the base `DataTables` and constructs its own rowset based on their rows.

After this is done, the corrections made to `C1DataView` rows are immediately reflected in the base `DataTables`' rows, and vice versa you can edit (modify/add/delete) rows directly through `DataTables`, and those corrections will be reflected in rowsets of all **`C1DataViews`** based on modified `DataTables`.

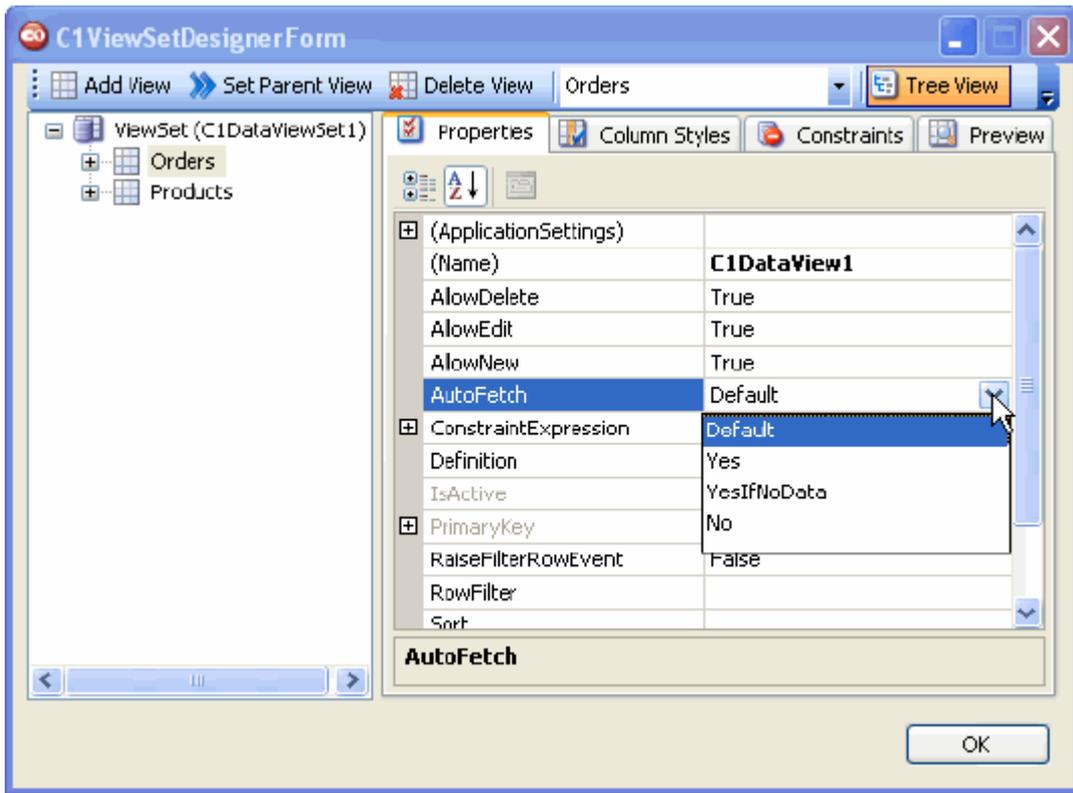
## Data Fetching

Fetching is the process of retrieving rows from the result set and returning them to the application. You can turn on or off `AutoFetching` for all views in `C1DataViewSet` by means of the **`C1DataViewSet.AutoFetch`** property (turned on by default).



You can control `AutoFetching` of a specific `C1DataView` data using the **`C1DataView.AutoFetch`** property. If **`C1DataView.AutoFetch`** is set to its default value, **`AutoFetchModeEnum.Default`**, then the `AutoFetch` mode is determined by the **`C1DataViewSet.AutoFetch`** property value of an owning `C1DataViewSet`.

The following capture shows how to access the **`C1DataView.AutoFetch`** property through the **`C1ViewSetDesignerForm`**:

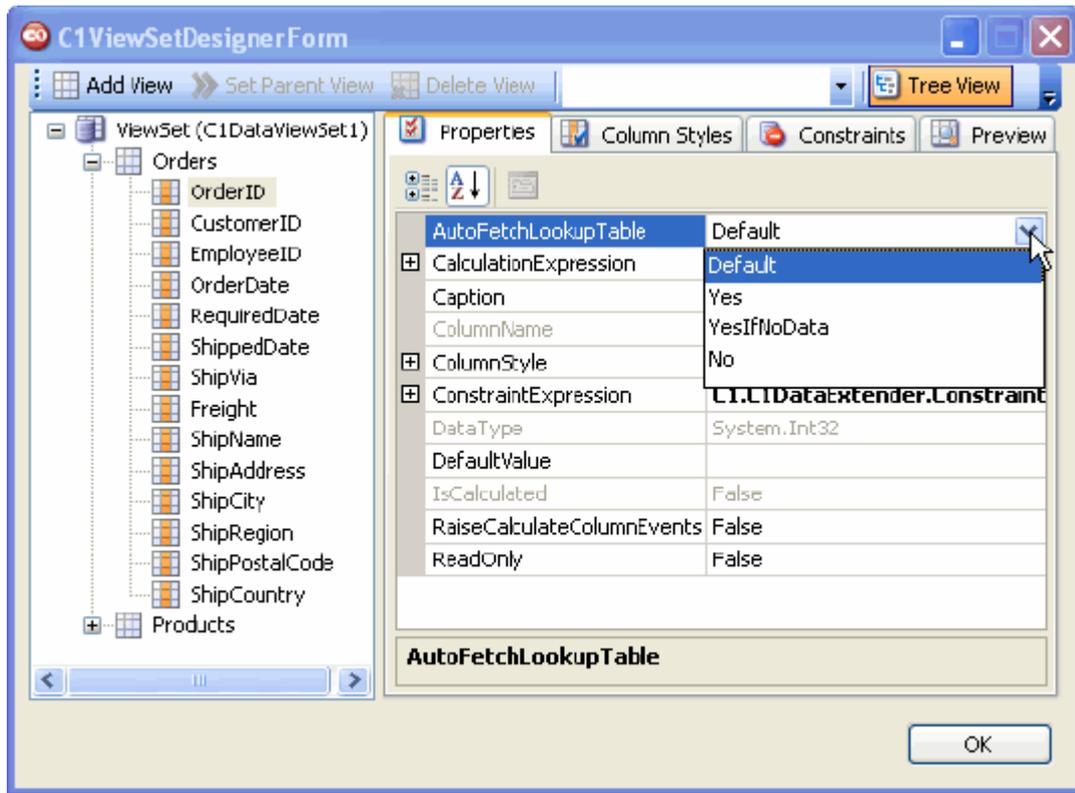


If the effective value of the AutoFetch property is set to **AutoFetchModeEnum.Yes**, then data for underlying DataTables is fetched even if the data has already been fetched by another C1DataView.

By changing this property value to **AutoFetchModeEnum.YesIfNoData**, you instruct **ADO.NET DataExtender** to check whether data necessary for this **C1DataView** has already been fetched by other fetch requests. If this is the case, then data will not be fetched repeatedly.

Automatic fetching of foreign key lookup tables is controlled by the AutoFetchLookupTable. It has an effect only for view columns that represents a foreign key DataColumn and has the ItemListType property set to **ItemListTypeEnum.ForeignKey**.

The following capture shows how to access the AutoFetchLookupTable property through the **C1ViewSetDesignerForm**:



If this property is set to its default value, **AutoFetchModeEnum.Default**, then an effective value of this property is determined by the **C1DataView.AutoFetch** property value of **C1DataView** that this column belongs to.

## Using the Fill method

To load data into your application, you must call the **Fill** method of a data adapter. When automatic data fetching is suppressed, or in order to refresh client rowsets with data from a server, you can use any of the following **Fill** methods:

- The Fill (including LookupTables as Boolean) method fills all the views in the viewset, and a Boolean parameter indicates whether foreign key lookup tables should be filled.
- The Fill () method is equivalent to C1DataViewSet.Fill (False).
- The C1DataView.Fill (Boolean) method allows the user to fill a specific C1DataView, with an indication whether its lookup tables should be filled.
- The Fill () method is equivalent to Fill (False).
- The FillLookupTable () method fills a lookup table of the column.

This method has an effect only if the column represents a foreign key DataColumn and its ItemListType property is assigned with ItemListTypeEnum.**ForeignKey** value

## Working with DataSetExtender

Typed ADO.NET dataset provides a set of properties for constituting DataTables and their DataColumn, which are propagated by C1DataViewSet to its **C1DataViews** and **C1ViewColumns**. For example, **DataColumn.ReadOnly** is used as a default value for the ReadOnly property of a column that is based on the corresponding **DataColumn**.

However, **C1DataViews** and their constituting C1ViewColumn objects provide an extended set of properties (compared to an ADO.NET **DataTable** and **DataColumn**, which are not necessary, but useful for some properties) when defining them one time in one single place, similar to what is done for typed dataset properties.

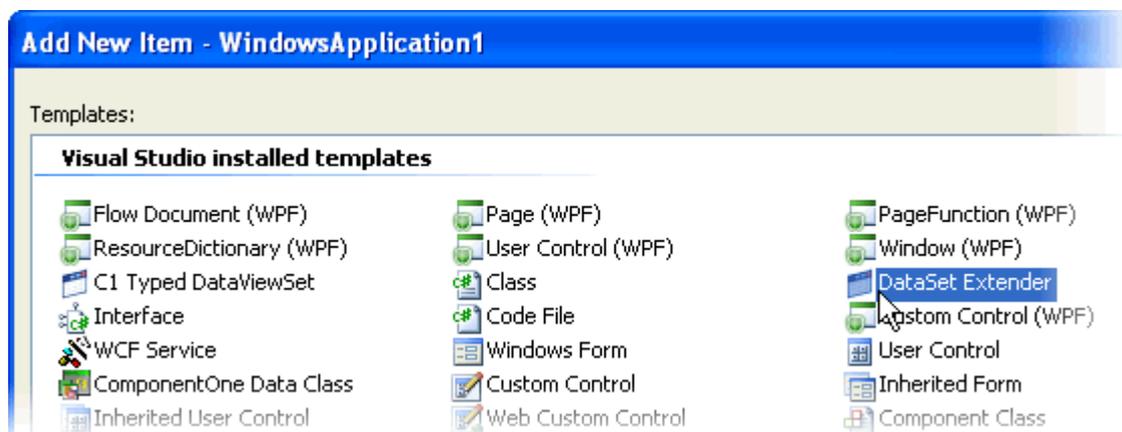
Such a place is provided by **ADO.NET DataExtender** and is called **DataSetExtender**. **DataSetExtender** provides storage for specifying an extended set of property values for the specific typed dataset. For example, **DataSetExtender** represents **DataTable** and **DataColumn** related properties which are not defined on these classes, but which C1DataViewSet would like to have and is capable of consuming. The core class that represents **DataSetExtender** is **DataSetExtender**.

## Creating a DataSetExtender

To create a **DataSetExtender**, perform the following tasks:

1. Select **Add New Item** from the **Project** menu in Visual Studio. Alternatively, right-click the project node in the Solution Explorer and select **Add | New Item** from its context menu.

The **Add New Item** window appears.



2. Select **DataSet Extender** for the dataset, enter a name for the DataSetExtender in the **Name** text box, and click **Add**.

The DataSetExtender is created as a global project item, similar to typed ADO.NET dataset, and its designer appears in a separate window of the VS IDE.

3. From the **Properties** window, set the **DataSetExtender.DataSet** property value to an existing typed ADO.NET dataset within the project.

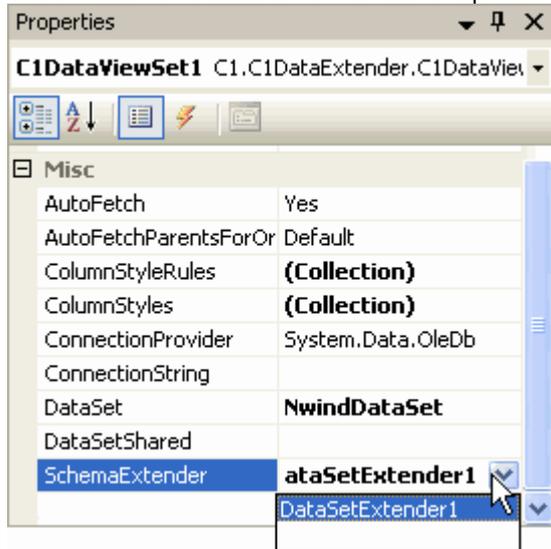
The **DataSetExtender** represents an additional set of properties for this dataset. The **DataSetExtender** automatically creates all items representing **DataTable** and **DataColumn** objects from the specified typed dataset, and will keep them in sync when they are updated in the future. For instance, if you add a new **DataTable** or **DataColumn** to the typed dataset a corresponding item is automatically created in DataSetExtender the next time you open its designer.

To set the properties for one of the items listed in the **DataSetExtender**:

1. Select an item which represents either a table or column in the **DataSetExtender** designer.
2. Modify its properties through the Visual Studio **Properties** window.

To apply DataSetExtender property values to a certain **C1DataViewSet** component:

1. Select one of the C1DataViewSet components on the form.
2. Set the C1DataViewSet.SchemaExtender property to the name of the **DataSetExtender** in the Visual Studio **Properties** window. Note that you have to rebuild your project for the name of the DataExtender to appear in the C1DataViewSet.SchemaExtender drop-down list.



## Connection Information

**DataTable** objects of a typed dataset usually have a corresponding Table Adapter that defines a connection (a DbConnection derived object) used to retrieve and update the data of a **DataTable** from/to the database server, along with other attributes devoted to the client and server data interchange.

The **C1DataViewSet** needs some additional information concerning the specifics of the database server, which is referenced in the connection of a Table Adapter. This additional information helps **C1DataViewSet** to provide the entirety of its capabilities; for example, automatically refreshing the server generated primary key value in a client row.

This information is represented in the **DataConnectionExtenderBase** derived class, **DataTableConnectionExtender**, belonging to the **DataTableExtender** objects of **DataSetExtender** and is accessible through the **DataTableExtender.ConnectionInfo** property.

The property values of the **DataTableConnectionExtender** objects can be:

- Determined automatically

To detect property values automatically, set the **DataConnectionExtenderBase.AutoDetect** property to **True**. In this case **ADO.NET DataExtender** will detect a type of database server represented by a corresponding Table Adapter connection. If this database server is familiar to **ADO.NET DataExtender**, then property values of the connection information object will be assigned automatically. The type of server is identified through the **DataConnectionExtenderBase.ServerType** property, if the server type is not recognized by **ADO.NET DataExtender** then this property is ServerTypeEnum.Unknown.

- Defined explicitly

When the server type is unknown you can manually set the connection information property values. To do this, ensure that **DataConnectionExtenderBase.AutoDetect** is set to **False** and **DataTableConnectionExtender.BaseConnection** to a null value, and then you can set property values

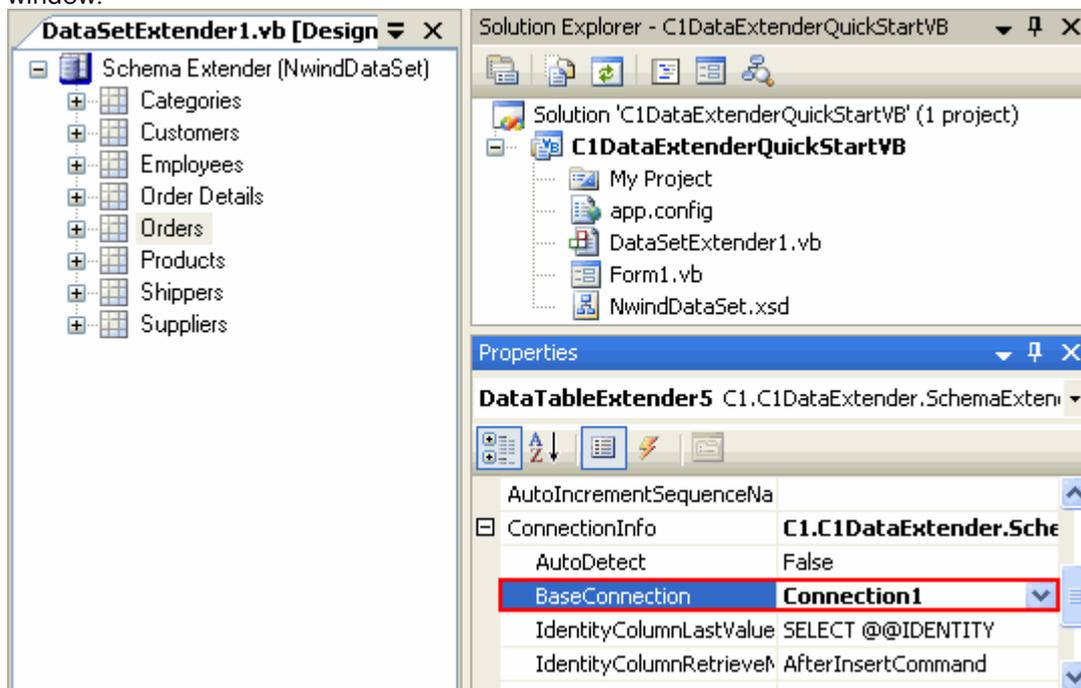
explicitly.

- Inherited from property values of the global connection information objects of type **DataSetConnectionExtender** contained in the **DataSetExtender.ConnectionInfo**'s collection

Usually, the only connection is used for all or most Table Adapters of a typed dataset. To make connection information management more flexible, **ADO.NET DataExtender** allows you to define one or more **DataSetConnectionExtender** objects in the **DataSetExtender.ConnectionInfos** collection and simply inheriting their values in **DataTableExtender.ConnectionInfo** objects. In order to use this capability, just set the **DataTableExtender.ConnectionInfo.DataTableConnectionExtender.BaseConnection** property value to an instance of a **DataSetConnectionExtender** object from the **DataSetExtender.ConnectionInfos** collection.

## To connect DataSetExtender to the specific typed dataset:

1. Select **DataSetExtender1.cs** (or **DataSetExtender1.vb**) from the Solution Explorer window and double-click it to open it.
2. From the Schema Extender tree, select the **DataTableExtender** that you wish to define.
3. Modify its **DataTableConnectionExtender.BaseConnection** property through the Visual Studio **Properties** window:



When **DataSetExtender** is initialized for the first time after connecting it to the specific typed dataset, it investigates connections defined in Table Adapters and creates a distinct list of corresponding **DataSetConnectionExtender** objects in the **DataSetExtender.ConnectionInfos** collection. Usually it will create a single **DataSetConnectionExtender** object. For each **DataTableExtender** object contained in the DataSetExtender the **DataTableConnectionExtender.BaseConnection** property value is set to the instance of the corresponding DataSetConnectionExtender object from the **DataSetExtender.ConnectionInfos** collection. Doing this will ease management in the future. If you need to make database server adjustments, simply change the property values of the global DataSetConnectionExtender object which will automatically change the property values of all **DataTableExtender.ConnectionInfo** objects.

## Working with Typed DataSet

**ADO.NET DataExtender** also provides a facility to create strongly typed view set definitions that are global for your project. Such a definition resides as a project level item and can be reused in multiple forms of your application. This feature is called typed **DataViewSet** and, compared to a locally defined **DataViewSet**, has the following benefits:

- Allows defining a view set structure in a single centralized place and the ability to reuse it in multiple places of your application. Being defined in a .dll, it can be reused in multiple applications.
- In addition to the view set structure, you can create event handler code for views and view set itself that will be a part of the view set definition, that is will be reusable as well.
- Provides a strongly typed object model for a convenient and safe access to a view set data in a code.

For example, the following VB code retrieves current Employee's name as a string from untyped view set:

```
CType(northWindViewSet.Views("Employees").Current("FirstName"), String)
```

The following VB code represents a typed view set:

```
northWindViewSet.EmployeesView.Current.FirstName
```

Taking into account the fact that during typing the code for the untyped view set you must type view and column names manually, whereas for typed view set you will have a full IntelliSense support, you may realize that programming against the typed view set is far more productive.

## Data Driven Application Paradigm

In order to create a robust and well-manageable data driven application which uses **ADO.NET DataExtender**, we suggest the following application architecture:

- Create a typed dataset that represents the whole or a significant part of your database. Define business logic here by writing event handlers for the **ColumnChanging/RowChanging/RowDeleting** events of DataTables.
- Create a **DataSetExtender** that extends this typed dataset.
- Use C1DataViewSet components connected to the dataset and referencing its **DataSetExtender** to define the form's data model. Bind UI controls to C1DataView objects defined in this C1DataViewSet, and write event handlers for events represented by C1DataViewSet to provide custom processing of data navigation and changing specific for this Form.
- If you would like to use the same view set definition in multiple places of the application, or if you just want a more convenient strongly typed access to the view set object model then you may want to create a project level typed **DataViewSet**.

## Data Library Approach

You have the opportunity to raise manageability of the data representation layer even greater by creating a Data Library that can be reused between multiple applications. To create a Data Library, perform the following steps:

1. Create a regular Class Library project.
2. Add typed **DataSet**, **DataSetExtender** and a number of typed **DataViewSet** components that to the library.

Now you can reference this Class Library in your applications and use components defined in the Class Library, see [Creating a Data Library](#) for details.

## Creating a Data Library

The ability to create project level view set definitions gives you the option to create a data library that can be reused among multiple applications. This can be done by means of a regular .dll (Class Library) project where you can add:

- Typed ADO.NET DataSet
- A number of typed DataSet definitions that uses those DataSet as a data source

Optionally, you can create:

- **ADO.NET DataExtender** that represents the DataSet and can be referenced in typed DataSet

You may then reference this .dll from you application's project and use the view sets defined in the library in a usual manner.

In the case when Data Library is included as a Project in your solution and you reference it as a project (**References**|**Add Reference** and select the **Projects** tab); rebuilding the solution makes all the typed view sets from the Data Library available in the Toolbox.

If Data Library project is not a part of the solution then you will need to install it in the Toolbox manually. Simply select the Data Library assembly in the **Choose Toolbox Items** dialog box of VS IDE. For more information, see [Adding ComponentOne Components to a Project](#).

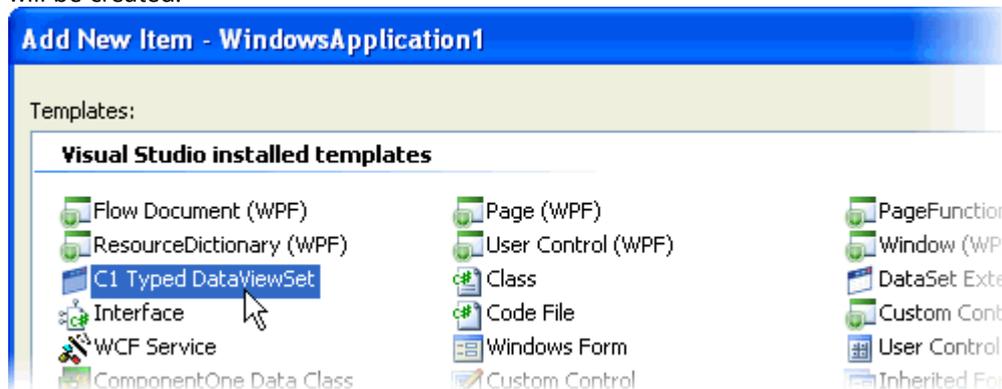
## Creating a Typed DataSet

To create a new typed DataSet, complete the following basic operations:

1. Select **Add New Item** from the **Project** menu in Visual Studio. Alternatively, right-click the project node in the Solution Explorer and select **Add | New Item** from its context menu.

The **Add New Item** dialog box of VS IDE appears.

2. Select the **C1Typed DataSet** item and click **Add**. A new project item that represents a typed DataSet will be created.



The **C1TypedDataSet** designer appears.

3. From the Toolbox, add a **DataSet** control to the component tray at the bottom of the **C1TypedDataSet** designer by performing a drag-and-drop operation.
4. The **Add DataSet** dialog box appears. The **Typed dataset** option is selected, click **OK**.
5. From the **Properties** window, assign the **DataSet** control to the DataSet property of the **C1TypedDataSet**.

The **C1TypedDataSet** designer is similar to the untyped **C1DataSet**, but it has added support that allows you to create specifically typed event handlers for views and the typed **C1DataSet** itself, as well as for any component placed on the component tray.

To create the specifically typed event handlers for views, simply double-click the event item in the **Events** list of the VS

IDE **Properties** window, which is accessed by selecting the **Events** button . An event handler method definition will be created in the code file.

As for other types of project level components, typed **DataViewSet** is represented by the following code files:

- A code file for a user's custom code with a filename.ext name template (you can define additional members for viewset and dataview classes, event handler methods are also here).
- A code file with the filename.Designer.ext name template that contains auto generated code and should not be changed manually.

## To use a typed **DataViewSet** instance in forms:

- Rebuild your application. A Toolbox item that represents this ViewSet appears under the **<Project Name> Components** tab at the top of the Toolbox.
- From there you can place it on a Form using a drag-and-drop operation.

## ADO.NET DataExtender Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio environment, and know how to use bound controls in general. Each topic provides a solution for specific tasks using **ADO.NET DataExtender**. The help uses an Access database, C1NWind.mdb, as the data source. It is assumed C1NWind.mdb is in the ComponentOne Samples\Common directory (see [ADO.NET DataExtender Samples](#) for more information) where it is installed when **ADO.NET DataExtender** is installed, and it is referred to by filename instead of the full pathname for the sake of brevity.

Each task-based help topic also assumes that you have created a new .NET project, added a **C1DataViewSet** component to the form and defined a data source for it. For additional information, see [Creating a .NET Project](#) and [C1DataViewSet Data Sources](#).

## Working with the C1ViewSetDesignerForm

The following topics demonstrate how to add and modify **C1DataViews** using the **C1ViewSetDesignerForm**. Using the **C1ViewSetDesignerForm**, you can define one or more views at design time. If the underlying data source has table relationships defined, then you can add individual tables accordingly.

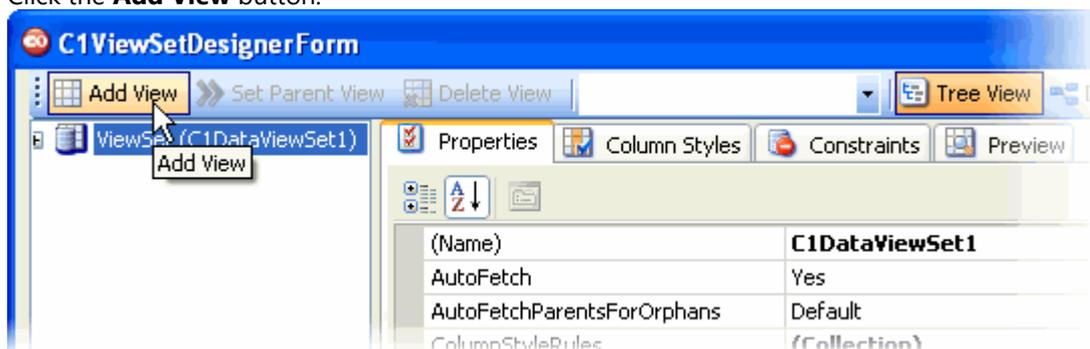
### Complete the following steps to open the C1ViewSetDesignerForm:

1. Click the smart tag (📌) located above the **C1DataViewSet** component to open its **C1DataViewSet Tasks** menu.
2. From its Tasks menu, select **Edit** to edit the **C1DataViewSet**'s structure. The **C1ViewSetDesignerForm** appears.

## Adding a C1DataView

To add a **C1DataView**, complete the following steps:

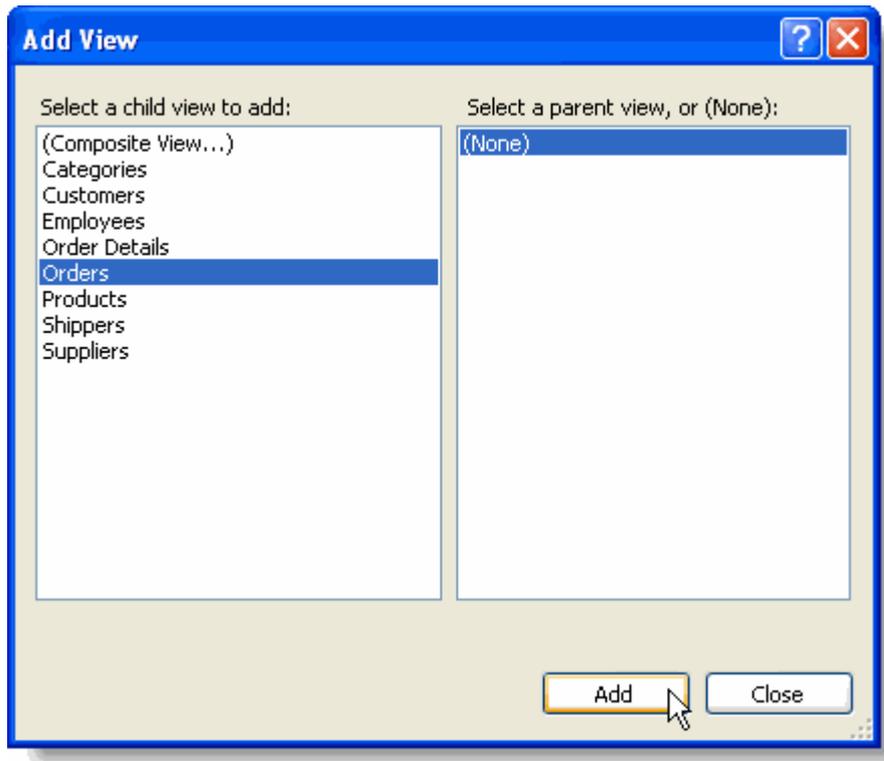
1. Open the **C1ViewSetDesignerForm**.
2. Click the **Add View** button.



The **Add View** dialog box opens.

You can create a definition at this time or add one later. You can also specify a parent view, if creating a child view.

3. Click **Add**, and then **Close**.



The view you have added will appear in the left pane of the **C1ViewSetDesignerForm**.

## Defining C1DataView

To create a **C1DataView** definition, use the Definition property or the **Add C1DataView** dialog box.

### Using the **C1DataView.Definition** property:

Select an existing **C1DataView**. You have two options when using the Definition property:

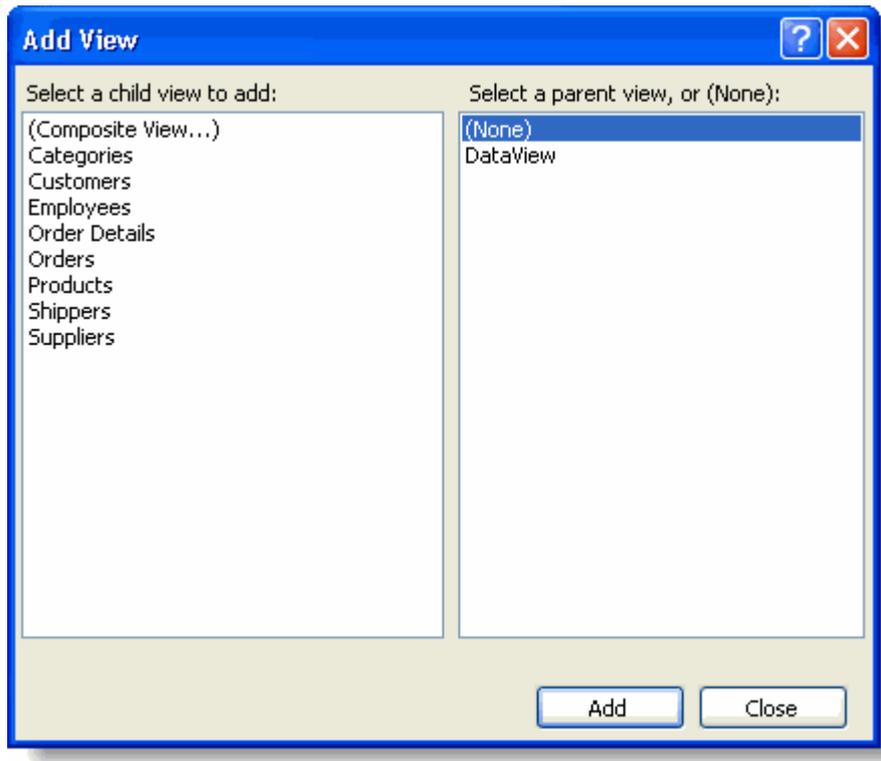
- Click the **ellipsis** button  next to the Definition property to open the **C1DataView Definition Statement Builder** and create the definition.

OR

- Enter a definition directly into the text box next to the **Definition** property.

### Using the **C1DataView** dialog box:

1. Open the **C1ViewSetDesignerForm**, and select **Edit** from the **C1DataViewSet Tasks** menu. The **C1ViewSetDesignerForm** appears.
2. Click the **Add View** button. The **Add View** dialog box opens.

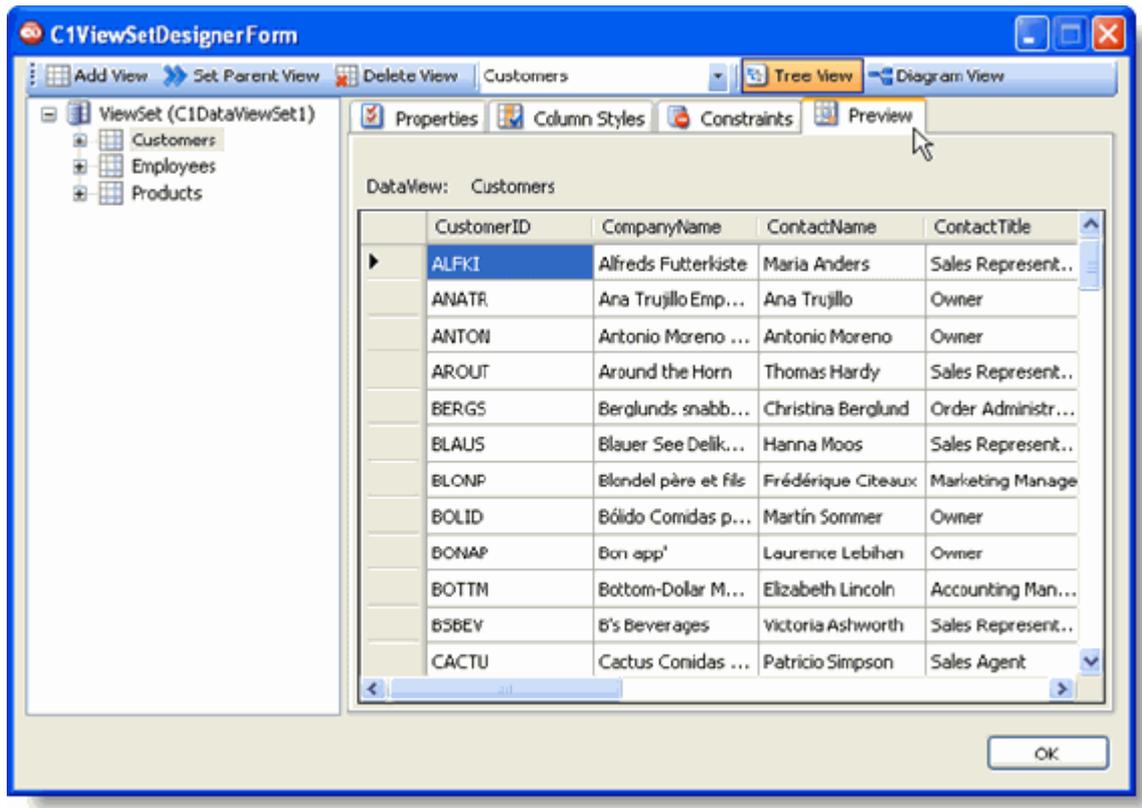


3. Select a table from the list of available dataset tables under *Select a child view to add* if creating a simple view definition.

## Previewing a C1DataView

You can view your changes and edits at design time by means of the **Preview** tab located in the **C1ViewSetDesignerForm**. To preview a C1DataView.Fill (Boolean) , complete the following steps:

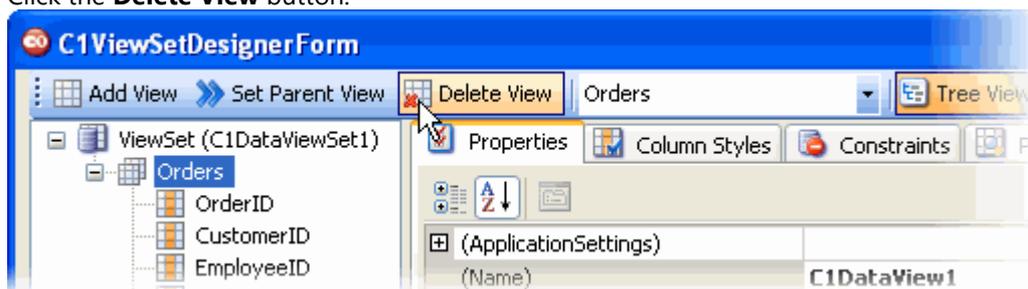
- Open the **C1ViewSetDesignerForm**.
- Select the view to preview.
- Click the **Preview** tab.



## Removing a C1DataView

To remove a view, complete the following steps:

1. Open the **C1ViewSetDesignerForm**.
2. Select the view to be removed.
3. Click the **Delete View** button.



## Working with the C1DataView Definition Statement Builder

The following topics show how to define **C1DataViews** using the **C1DataView Definition Statement Builder**.

## Accessing the C1DataView Definition Statement Builder

To open the **C1DataView Definition Statement Builder**, you have two options:

If you have not created a view:

1. From the **C1ViewSetDesignerForm**, select the **Add View** button to open the **Add View** dialog box.
2. From the **Add View** dialog box, select **Composite View** under *Select a child view to add* and click **Add**.

If you have created a view:

1. From the **C1ViewSetDesignerForm**, select the view (table) from the left pane.
2. In the **Properties** grid, click the **ellipsis** button  next to the Definition property.

## Adding a Child Table Node

To add a child table node, complete the following steps:

1. Open the [C1DataView Definition Statement Builder](#).
2. Click the **Add Child Table** button to open the **Add Child Table** dialog box.
3. Select a table from the list of available tables under *Select a child table to add*.



 **Note:** The available tables in the list depend on the **DataRelations** in your **DataSet**. If specifying the initial table, all tables within the **DataSet** are available. If specifying a child table node for an existing table, only tables that have a relation to that table are available. Open the **DataSet Designer** to view table relations. For more information on relations and the **DataSet Designer**, see the Microsoft Visual Studio Documentation.

4. Click **Add**, and **Close**.

## Accessing and Changing Join Properties

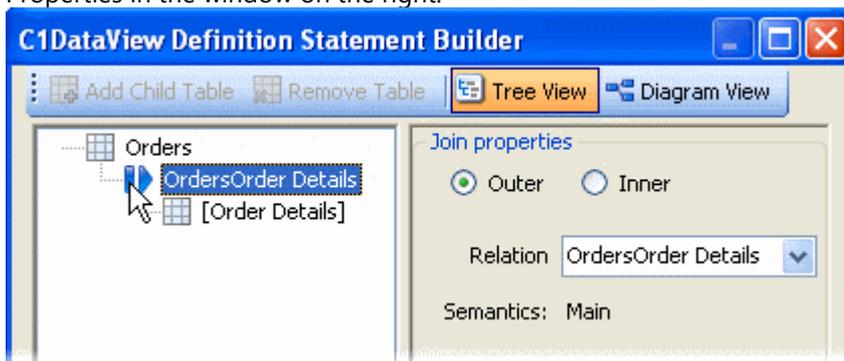
You can specify whether a join is an OUTER or INNER join and set the relation between the tables through the **Join properties** dialog box. To change join properties from the **C1DataView Definition Statement Builder**, complete the following steps:

1. Open the [C1DataView Definition Statement Builder](#).

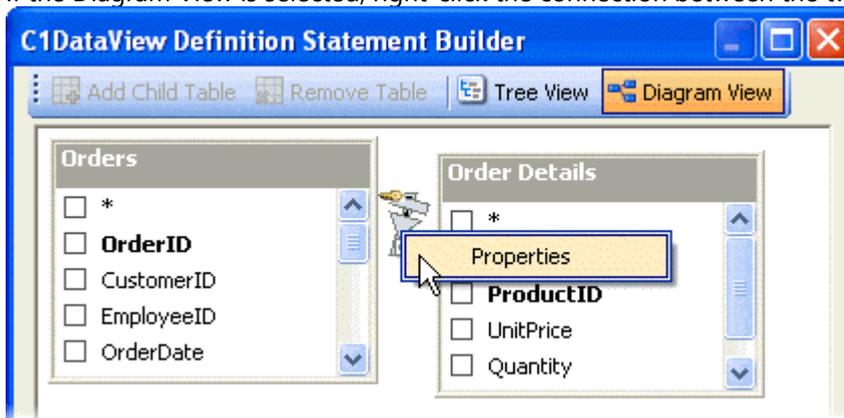
2. Add a [Child Table Node](#).

When adding the initial table, the **Join properties** dialog box is not available. Add a child table to the current table for the **Join properties** dialog box to become available.

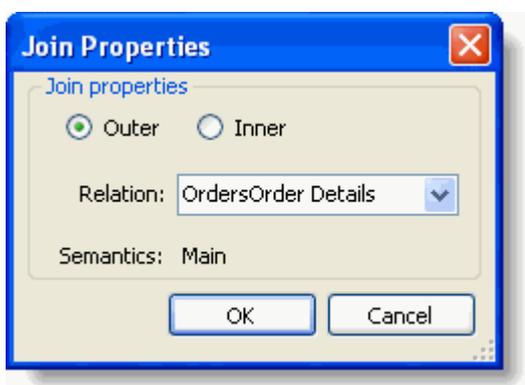
3. Click the **Add Child Table** button to open the **Add Child Table** dialog box.
4. Select the table you just added from the list of available tables under *Select a parent view, or None*.
5. Select a related table from the list of available tables under *Select a child table to add*.
6. Click **Add**, and **Close**.
7. Open the **Join properties** dialog box.
  - o If the Tree View is selected, click the connection node between the two tables to view the Joint Properties in the window on the right.



- o If the Diagram View is selected, right-click the connection between the two tables and select **Properties**.



The **Join properties** dialog box will appear.



8. From the **Join properties** dialog box, select an OUTER or INNER join and set the relation between the two selected tables from the Relation drop-down box.

If a relation is not specified here then it will be automatically detected. A relation must be specified only if

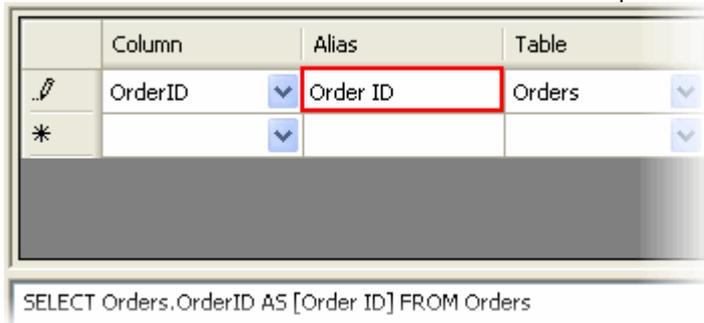
related tables have more than one relation between them, because, in this case, the automatically detected relation may not be the relation you want.

## Specifying or Changing the Table Alias

A table alias can be specified when the table is added to the definition statement. If no alias was specified or the alias needs to be changed, you can add one through the **Alias** property.

To add a table alias through the **Alias** property, complete the following steps:

1. Open the [C1DataView Definition Statement Builder](#).
2. Select a table from the **Table** drop-down list and a column from the **Column** drop-down list.
3. Enter an alias name in the **Alias** text box in the lower pane.

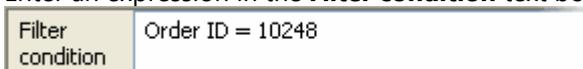


## Specifying a Filter Condition

You can specify a filter condition (an expression of the WHERE clause of a definition statement) that is used to limit number of rows represented by the view. The expression can include constants, column references, arithmetical, logical and comparison operations, and IS [NOT] NULL constructions, and subexpressions can be entered and grouped using parentheses. Column references of the expression must indicate columns from the base **DataTable(s)**, but not the columns from the SELECT list. The expression must return a logical value.

To specify a filter condition, complete the following steps:

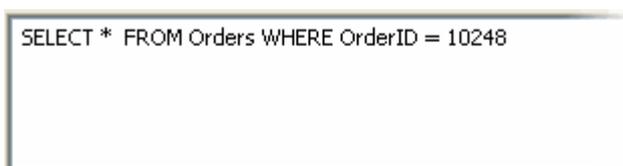
1. Open the [C1DataView Definition Statement Builder](#).
2. Enter an expression in the **Filter condition** text box.



For example, if you want to apply a filter that finds a specific *OrderID*, I can use the following expression:

```
OrderID = 10248
```

The expression is added to the definition statement.



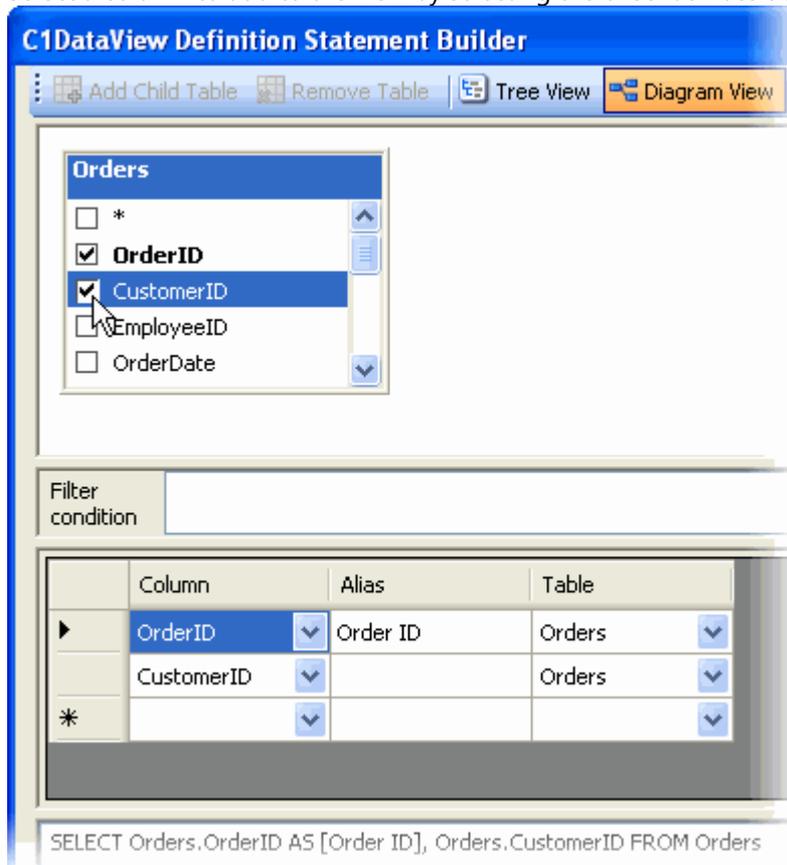
In the view, only rows containing an *OrderID* of 10248 are displayed.

## Adding Specific Columns to the View

From the **C1DataView Definition Statement Builder** you can add columns to the view manually or using the **Add columns** buttons.

### To add columns manually (in Diagram View):

1. Select a column to add to the view by selecting the checkbox beside the column item



Note that you may also use the **Column** drop-down list to select a column to add to the table

2. Enter an alias name for the column in the **Alias** text box
3. Specify a table using the **Table** drop-down list

### To add columns using the Add columns buttons (in Tree View)

1. From the **Columns** list in the right pane, select a column to add
2. Click the **Add selected columns** button . If you want to add all of the columns that are not already in the definition statement, click the **Add all columns which are not in statement** button. The specified columns appear in the left pane

## Viewing and Changing the Definition Statement

Once you have added tables and columns for the definition statement, the definition statement appears in the bottom pane. You can change the statement in this pane, if necessary, although this can change other settings in the definition statement builder.

## Adding a Child View

To add a child view to a `C1DataViewSet`, complete the following steps:

1. In the **C1ViewSetDesignerForm**, select a view in the left pane and press the **Add View** button.
2. Select a child view to add from the list and click **Add**. Select the **ellipsis** button  next to the Definition property to open the **C1DataView Definition Statement Builder**.
3. Set up a **C1DataView** definition, either simple or composite, in the **C1DataView** definition area of the dialog box and click **OK**. See [C1DataView Definitions](#) for more information on creating simple and composite views.

The child view object is created and has the Definition property value set.

4. Click **OK** to close the **C1DataView Definition Statement Builder** and again to close the **C1ViewSetDesignerForm**.

## Committing Changes

To commit changes to the server, complete the following steps:

1. Add a **Button** control to your Windows form.
2. Define some **C1DataView** objects for the `C1DataViewSet` component on the form.
3. Add the following code:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
Button1.Click
    C1DataViewSet1.Update()
End Sub
```

### To write code in C#

#### C#

```
private void button1_Click(object sender, EventArgs e)
{
    C1DataViewSet1.Update();
}
```

The Update method automatically determines the correct order in which rows should be committed to the server. It also refreshes client row columns with new values generated on the server, including the cases of server generated autoincrement columns in conjunction with master-detail relationships between tables.

## Sorting and Filtering Data

### Sorting

To sort **C1DataView** rows, complete the following steps:

1. From the **C1ViewSetDesignerForm**, select a `DataView`.

2. Enter a sort order definition in the Sort property by specifying the columns in the order they should be sorted.

 **Note:** Each column name should be separated by a comma. Ascending order is the default sort order, but you can sort in descending order by adding DESC after the column name.

## Filtering

To filter **C1DataView** rows, complete the following steps:

1. From the **C1ViewSetDesignerForm**, select a view from the ViewSet.
2. Enter a filter definition in the DataView's **C1DataView.RowFilter** property.

The syntax of a filter expression is the same as for the WHERE clause of the View Definition Language, with the single exception that the RowFilter expression references columns of **C1DataView** (C1ViewColumn objects) while the View Definition Language's WHERE expression references columns of the base DataTable(s).

## Updating a C1DataView Definition

This topic shows how to update a view's definition at run time. It assumes you have created a project with two DataGrids bound to a C1DataViewSet and two text boxes. To update the view's definition at run time, add code to the **Button\_Click** events:

1. Add the following **Button1\_Click** event code:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs) Handles Button1.Click
    C1DataViewSet1.Views(0).Definition = TextBox1.Text
End Sub
```

### To write code in C#

#### C#

```
private void button1_Click(object sender, EventArgs e)
{
    C1DataViewSet1.Views[0].Definition = textBox1.Text;
}
```

2. Add the following **Button2\_Click** event code:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Button2_Click(ByVal sender As Object, ByVal e As EventArgs) Handles Button2.Click
    C1DataViewSet1.Views("OrdDetProductView").Definition = TextBox2.Text
End Sub
```

### To write code in C#

C#

```
private void button2_Click(object sender, EventArgs e)
{
    C1DataViewSet1.Views["OrdDetProductView"].Definition = textBox2.Text;
}
```

The **Update** button enables the user to make and update changes to the view's definition statement in run time.

## Editing at run time:

Observe that the views are filled automatically.

The screenshot shows a Windows Form titled "Form1" with two data views. The first view, "OrdersView", has a text box containing the SQL query "Select \* From Orders" and an "Update Definition" button. Below the text box is a table with the following data:

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight
10248	VINET	5	8/4/1994	9/1/1994	8/16/1994	3	32.38
10249	TOMSP	6	8/5/1994	9/16/1994	8/10/1994	1	11.61
10250	HANAR	4	8/8/1994	9/5/1994	8/12/1994	2	65.83
10251	VICTE	3	8/8/1994	9/5/1994	8/15/1994	1	41.34

The second view, "OrdDetProductView", has a text box containing the SQL query "Select \* From [Order Details] join Products" and an "Update Definition" button. Below the text box is a table with the following data:

OrderID	UnitPrice	Quantity	Discount	ProductID	ProductName	SupplierID	CategoryID
10249	18.6	9	0	14	Tofu	6	7
10249	42.4	40	0	51	Manjimp Dri	24	7
*							

A "Commit changes" button is located at the bottom right of the form.

**C1DataViewSet1** fills the underlying base DataTable(s) of `nwindDataSet1` which are referenced in the views (**Orders**, **Order Details** and **Products** in this example) and builds rowsets of views based on the content of those tables. All other DataTable(s) of `nwindDataSet1` remain empty.

## Testing the Update button:

- In the **OrdersView** definition textbox, change **Orders** to **Products**.
- Now press the **Update** button and you'll see that the DataTable changes to the **Products** view and builds rowsets of views based on the content of the **Products** DataTable.

### Sample Available

For the complete sample, see the **ViewSet\_Define\_Fill\_Update** sample, which is available for download from the [ComponentOne HelpCentral Sample](#) page.

## Composite Row Editing

This topic demonstrates the behavior of composite rows after editing. It assumes you have created a project with a textbox, a button and three DataGridViews. One grid is bound to a C1DataViewSet and the other two are bound to a dataset. To begin, complete the following steps:

1. From the **C1ViewSetDesignerForm**, create new **C1DataView** with the following definition to establish a composite view with an outer join between the **Orders** and **Order Details** tables: `SELECT o.OrderID, o.CustomerID, od.* FROM Orders AS o OUTER JOIN [Order Details] AS od`
2. Connect **DataGridView1** to the **C1DataView** by setting its **DataGridView.DataSource** property to **C1DataViewSet1** and **DataGridView.DataMember** property to **DataView**.
3. Set the **DataGridView.DataMember** property for **dataGridView2** and **dataGridView3** to **Orders** and **Order Details**, respectively. Then set the **DataGrid.DataSource** property for both **DataGrid2** and **DataGrid3** to **nwindDataSet1**.

**dataGridView2** and **dataGridView3** are now bound to **Orders** and **Order Details** **DataTable(s)** from the **nwindDataSet1 DataSet**, that is, to the base tables on which the view is based.

4. Create an **Update** button to the right of the textbox. See [Updating a C1DataView Definition](#) for instructions on this.
5. Run the application and begin editing the composite rows to view their behavior.

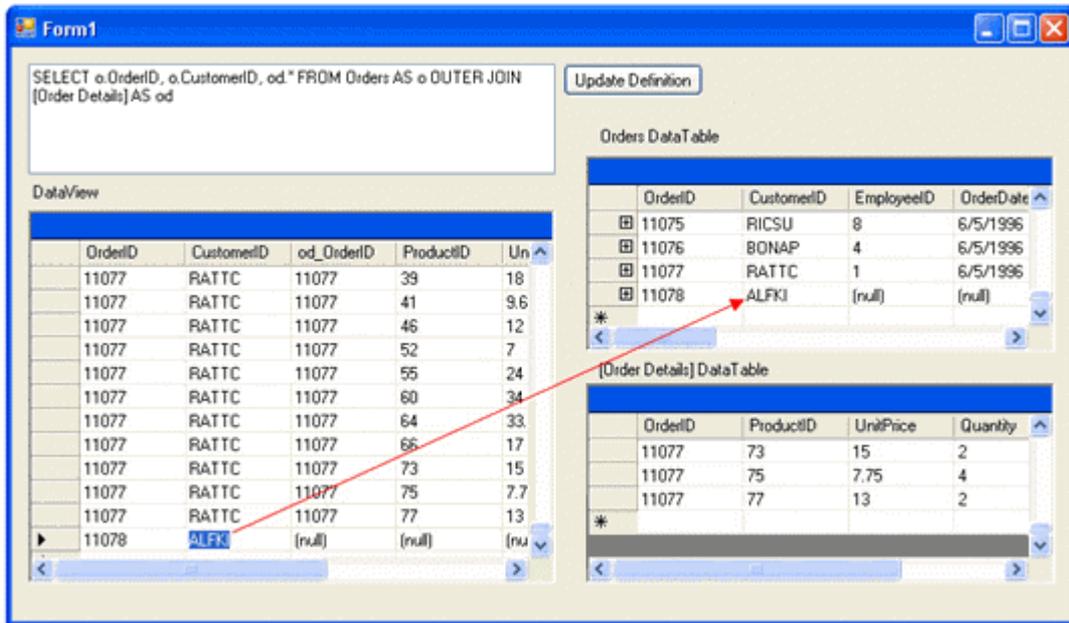
### Sample Available

For the complete sample, see the **CompositeRowEditing** sample, which is available for download from the [ComponentOne HelpCentral Sample](#) page.

## DataView Row Actions for an Outer Join

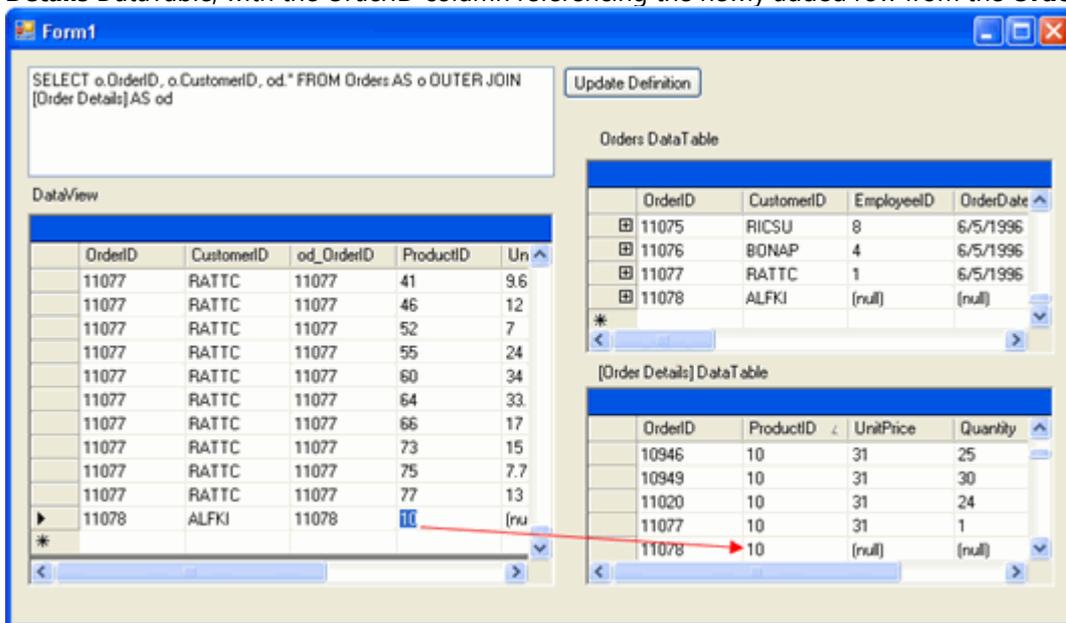
To observe **DataView** row actions for an **OUTER JOIN**, complete the following steps:

- Add a new row to the **DataGridView** (**dataGridView1**) and assign the *CustomerID* column **ALFKI**. Scroll to the bottom of the **Orders** **DataTable** (**dataGridView2**) and the **Order Details** **DataTable** (**dataGridView3**) and notice that the new row has been added to **Orders** **DataTable**, but not the **Order Details** **DataTable**. This occurs because the **OrdersDetails** **DataTable** does not have a *CustomerID* child row.



Also note that the *OrderID* column of the new row has taken a value automatically, because this column is defined as **DataColumn.AutoIncrement** in `nwindDataSet1`.

- Now select the newly added row and assign the *ProductID* to 10 (note that this column represents a column from the **Order Details** DataTable), and end editing. Observe that the new row has been added to the **Order Details** DataTable, with the *OrderID* column referencing the newly added row from the **Orders** DataTable.

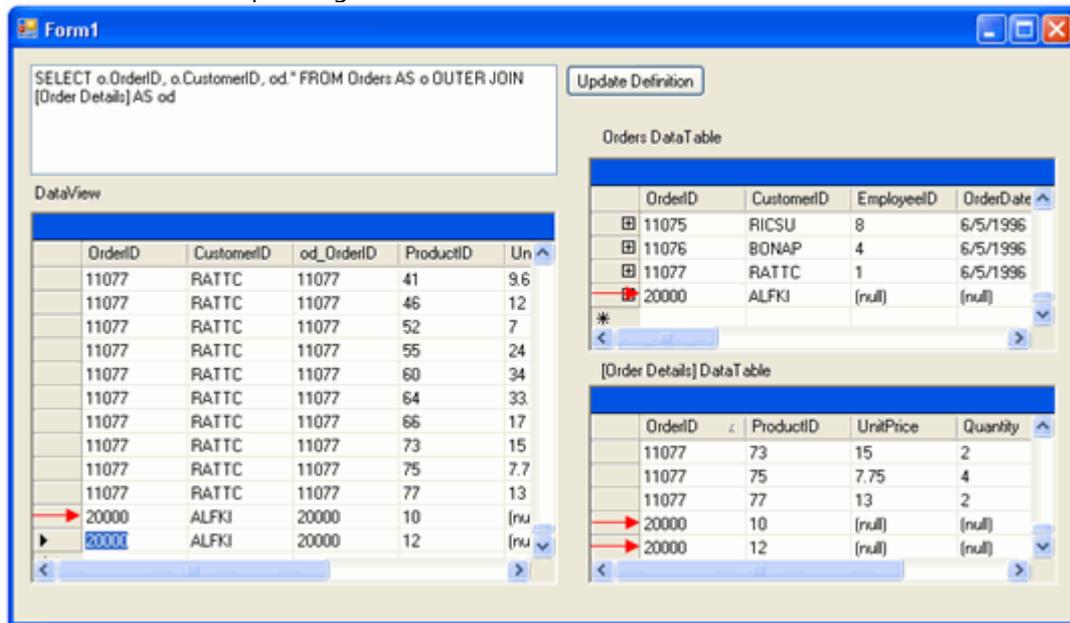


- Add another row to the `DataView`'s grid, and set its *od\_OrderId* column (foreign key of **Orders Details** DataTable referencing the **Orders** DataTable) to the same value as in the previously added row. Next, assign the *ProductID* column to 12, and end editing. Observe that nothing has changed in the **Orders** DataTable, but a new row has been added to the **Order Details** DataTable. Note that a new row has not been added to the **Orders** DataTable because the *od\_OrderId* column has been assigned with a value referencing the existing row from the **Orders** DataTable.

`img border="0" alt="" src="ImagesExt/image9_22.png" width="549" height="329" />`

- Set the view's *OrderID* column value (primary key of **Orders** DataTable) in one of the newly added rows to

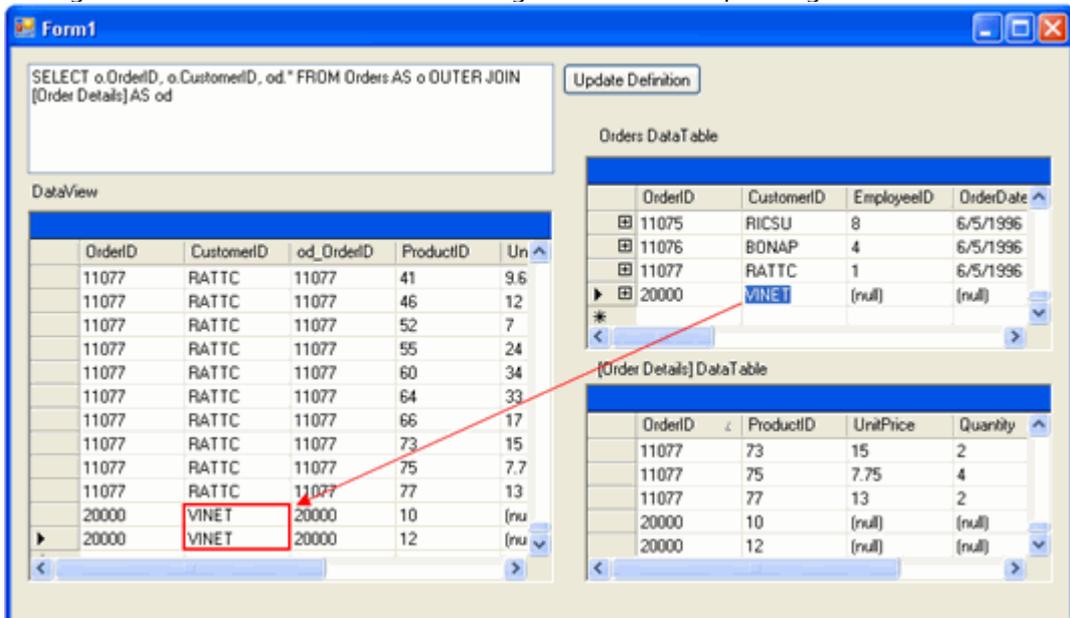
20000. Observe that this value will be changed in the other view's added row, as well as the values of *OrderID* columns in the corresponding rows of **Orders** and **Order Details** DataTables.



## Modifying the Base DataTable(s)

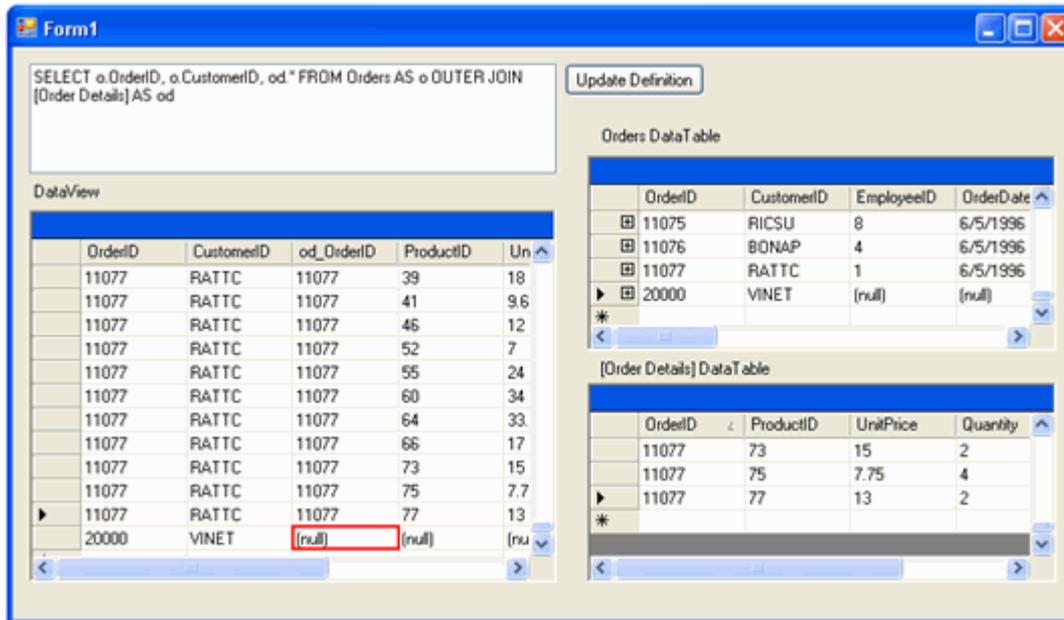
To observe DataView row actions for an OUTER JOIN, complete the following steps:

1. In the **Orders** DataTable, change the value of *CustomerID* column of the previously added row to **VINET**, end editing and observe that this value has changed in both corresponding rows of the DataView.



2. In the **Order Details** DataTable, delete one of the newly added rows and see that the corresponding composite row was deleted from DataView.

Then delete the other newly added row and observe that the corresponding row has not disappeared from DataView, but the values of columns corresponding to the **Order Details** DataTable became null. This is because of the OUTER join.



Now delete the row from the **Orders** DataTable and see that the corresponding row disappeared from DataView.

## DataView Row Actions for an Inner Join

To observe DataView row actions for an INNER JOIN, complete the following steps:

1. In the textbox containing the view definition statement, change the OUTER keyword to INNER (to define an inner join) and press the **Update Definition** button to apply the changes.

```
SELECT o.OrderID, o.CustomerID, od.* FROM Orders AS o INNER JOIN
[Order Details] AS od
```

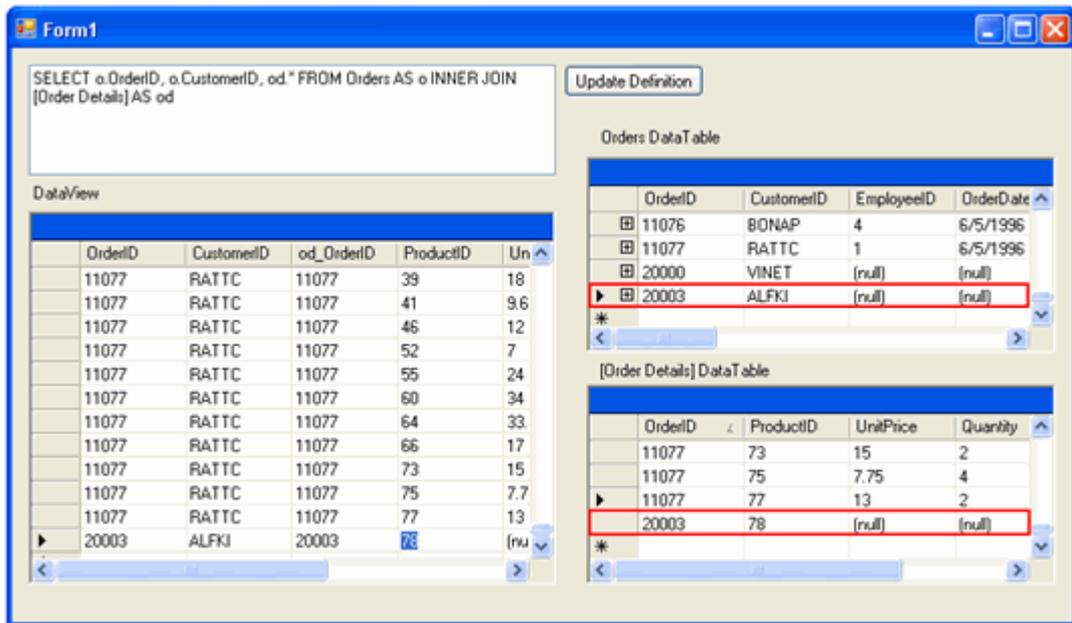
2. Perform the same operations as from [DataView Row Actions for an OUTER Join](#) and notice the following changes:
  - o When you try to add a view row without assigning some column corresponding to a column from the **Order Details** DataTable, you get a constraint violation exception concerning the empty *ProductID* column.



This occurs because the inner join composite view row can only exist if it references the rows from each base DataTable.

- o Click the **Yes** button and add a value to the *ProductID* column; notice that the view creates a new row in

the **Order Details** and **Orders** DataTables.



- When you delete the last child row from **Order Details** DataTable for the specific parent row, a corresponding view row will be removed, regardless of the existence of a parent row from the **Orders** DataTable.

## Using C1DataViewSet with an Untyped ADO.NET DataSet

This topic demonstrates how to use C1DataViewSet in conjunction with an untyped ADO.NET DataSet, where data fetching and committing are performed by means other than the C1DataViewSet. It assumes you have created a project with a textbox, three buttons and three DataGridView controls.

**Note:** C1DataViewSet only allows the user to show and edit data from untyped DataSet tables according to its **C1DataView(s)** definitions.

To begin, complete the following steps:

1. Create an untyped DataSet by adding a **DataSet** control to the form and connecting the C1DataViewSet to **dataSet1**. For more information on creating an untyped DataSet, see the Microsoft Visual Studio Documentation.
2. To connect **dataGridView1** to the C1DataViewSet, set the DataGridView.**DataSource** property to **C1DataViewSet1**.
3. Set dataGridView1's **DataGridView.DataMember** property to **DataView**.
4. From the **C1ViewSetDesignerForm**, select the DataView's Definition property and enter the following definition to establish a single view: `Select * From Orders join [Order Details]`
5. Add an **OleDbConnection** and two **OleDbDataAdapter** controls to your form.

Configure the settings to create a data connection that allows the adapter to communicate with the C1NWind.mdb database. For more information on Data Adapter configuration, see the Microsoft Visual Studio Documentation.

6. Set the OleDbConnection.**ConnectionString** property to C1NWind.mdb to create a data connection to the specific database.
7. Change the OleDbAdapter1.**Name** property to *ordersDataAdapter* and the OleDbAdapter2.**Name** property to *OrdDetDataAdapter*.

 **Sample Available**

For the complete sample, see the **UntypedDataSet** sample, which is available for download from the [ComponentOne HelpCentral Sample](#) page.

## Fetching Data from the Server at Run Time

To fetch data from the server at run time by the click of a button, complete the following steps:

1. Double-click button1 (in the sample project, the **Create Orders and Fill** button at the right of **dataGridView2**) to bring up the **Button1\_Click** event handler and replace it with the following code:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
Button1.Click
    ordersDataAdapter.Fill(dataSet1)
    Dim tb As DataTable = dataSet1.Tables("Orders")
    tb.PrimaryKey = New DataColumn() {tb.Columns("OrderID")}
    ordersDataGrid.DataSource = dataSet1
    ordersDataGrid.DataMember = "Orders"
    Button1.Enabled = False
End Sub
```

### To write code in C#

#### C#

```
private void button1_Click(object sender, EventArgs e)
{
    ordersDataAdapter.Fill(dataSet1);
    DataTable tb = dataSet1.Tables["Orders"];
    tb.PrimaryKey = new DataColumn[] { tb.Columns["OrderID"] };
    ordersDataGrid.DataSource = dataSet1;
    ordersDataGrid.DataMember = "Orders";
    button1.Enabled = false;
}
```

2. Double-click button2 (in the sample project, the **Create OrderDetails and Fill** button at the top right of **dataGridView3**) to bring up the **Button2\_Click** event handler and enter the following code:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Button2_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
Button2.Click
    ordDetDataAdapter.Fill(dataSet1)
    Dim tb As DataTable = dataSet1.Tables("Order Details")
    tb.PrimaryKey = New DataColumn() {tb.Columns("OrderID"),
tb.Columns("ProductID")}
    ordDetDataGrid.DataSource = dataSet1.Tables("Order Details")
    Button2.Enabled = False
```

```
End Sub
```

**To write code in C#**

```
C#
private void button2_Click(object sender, EventArgs e)
{
    ordDetDataAdapter.Fill(dataSet1);
    DataTable tb = dataSet1.Tables["Order Details"];
    tb.PrimaryKey = new DataColumn[] { tb.Columns["OrderID"],
    tb.Columns["ProductID"] };
    ordDetDataGrid.DataSource = dataSet1.Tables["Order Details"];
    button2.Enabled = false;
}
End Sub
```

3. Double-click button3 (in the sample project, the **Create Relation** button at the bottom right of dataGrid3) to bring up the **Button3\_Click** event handler and enter the following code:

**To write code in Visual Basic**

```
Visual Basic
Private Sub Button3_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
Button3.Click
    Dim rel As New DataRelation("Orders - Order Details",
dataSet1.Tables("Orders").Columns("OrderID"), dataSet1.Tables("Order
Details").Columns("OrderID"))
    dataSet1.Relations.Add(rel)
    Button3.Enabled = False
End Sub
```

**To write code in C#**

```
C#
private void button3_Click(object sender, EventArgs e)
{
    DataRelation rel = new DataRelation("Orders - Order Details",
dataSet1.Tables["Orders"].Columns["OrderID"],
dataSet1.Tables["Order Details"].Columns["OrderID"]);
    dataSet1.Relations.Add(rel);
    button3.Enabled = false;
}
```

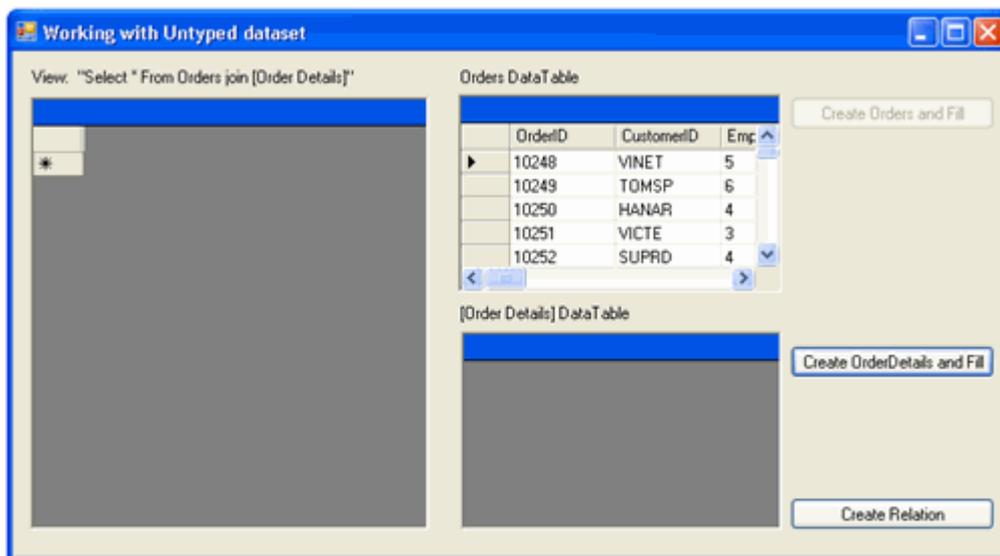
**Run the application and observe the following:**

- Notice that the view is inactive (no columns, no rows). This is because the underlying dataSet1 is empty.

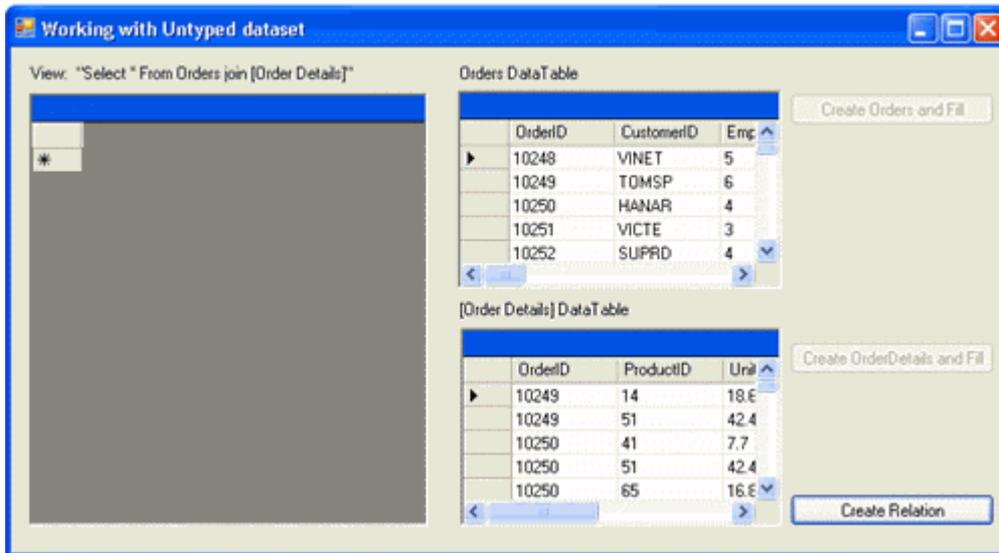


- Press the **Create Orders and Fill** button, which adds **Orders** DataTable to dataSet1, filling the view with data.

This operation is performed by means of the OleDbDataAdapter object (ordersDataAdapter) which operates against Access C1NWind.mdb database. The view is still inactive, because the **Order Details** DataTable has yet to be added in dataSet1, which is referenced in the view's definition.



- Press **Create OrderDetails and Fill** button, notice that the **Order Details** DataTable has been added to dataSet1 and filled the view with data.



After this, action view is inactive because the join specified in the view's definition supposes presence of DataRelation between joined DataTable(s).

- Finally, press **Create Relation** button, which creates DataRelation between tables. This operation makes the DataView become active, filling the view with composite rows based on data from underlying DataTable(s).

