ComponentOne

DataObjects for .NET

ComponentOne, a division of GrapeCity
201 South Highland Avenue, Third Floor
Pittsburgh, PA 15206 USA

Website: http://www.componentone.com
Sales: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Trademarks

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 $2 5 (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

Table of Contents

DataObjects for .NET Overview

DataObjects for .NET is a complete data and business objects framework that can be used in .NET applications of any range, scalability and architecture, from simple desktop applications to classic client-server applications to 3-tier distributed applications and enterprise-wide business object libraries. **DataObjects for .NET** includes two editions with a variety of components to best suits your needs: **DataObjects Enterprise for .NET** and **DataObjects Express for .NET**.

DataObjects for .NET Enterprise Edition is comprehensive while **DataObjects Express for .NET** has virtually no learning curve and, mediating the complexities of ADO.NET, makes data access and data binding in .NET applications simple. Both have extensive design-time support, particularly the more powerful **Enterprise Edition** which includes an easy-to-use Schema Designer. **DataObjects for .NET** includes many powerful features absent in standard ADO.NET including virtual mode technology and the ability to update the database immediately after the user changes a row. With data libraries, composite multi-table support, and more, **DataObjects for .NET** makes creating sophisticated, fully scalable Web-based distributed applications a matter of point-and-click.

Getting Started

Get started with the following topics:

- Key Features
- ADO.NET
- Quick Start
- 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.

Key Features

DataObjects for .NET's key features include:

- **Visual Studio Integration**
  DataObjects for .NET includes integration with Visual Studio Windows Forms features including Smart Tags and Data-binding.
- **Based on ADO.NET Technology**
  DataObjects for .NET is based on Microsoft ADO.NET technology and enhances it to empower and simplify database application development in Visual Studio. **DataObjects for .NET**' programmatic object model closely follows ADO.NET, so it will look very familiar to those who are used to the ADO.NET object model.
- **Reuse Business Logic Components**
  DataObjects for .NET Enterprise Edition uses the standard business object paradigm to allow you to develop business logic components (data libraries) and reuse them in multiple client projects. It provides clear separation of business and data logic from the presentation (GUI) layer.
- **Structured, Consistent Data**
  Expose data to users and client applications in a structured consistent way, so that all structural dependencies are maintained automatically without manual coding.
- **Multi-Table Object Support**
  Unlike other business object, data object, and data persistence frameworks, **DataObjects for .NET** fully supports multi-table objects (composite tables) automatically enforcing data relations without manual coding.
- **3-tier Web-based Application Support**
  DataObjects for .NET Enterprise Edition completely automates the task of developing distributed 3-tier Web-based applications. No special server-based code is necessary; making your application distributed becomes a simple matter of deployment configuration.
- **Virtual Mode Technology**
  With an innovative virtual mode technology, **DataObjects for .NET** allows you to use large datasets in .NET Windows Forms applications, a feature that is not supported in Visual Studio and ADO.NET without **DataObjects for .NET**.
- **Data Libraries**
  In enterprise development, **DataObjects for .NET Enterprise Edition** allows you to create a centralized and reusable repository of data schema and business logic (data libraries) used in applications throughout the enterprise.
- **Automated Database Updates**
  DataObjects for .NET completely automates database updates so there's no need for manual coding or to use ADO.NET DataAdapter or other special components. **DataObjects for .NET** can even update the database when multiple and interrelated changes have been made to multiple tables.
- **Immediate Database Updates**
  In **DataObjects for .NET** you can update the database immediately after the user changes a row, just by setting a single property. This optional feature, not supported by standard ADO.NET, is commonly used in desktop and classic client-server applications.
- **ADO.NET Data Storage**
  DataObjects for .NET stores data in an accessible ADO.NET DataSet; this enables a powerful combination of **DataObjects for .NET** and ADO.NET in the same code. Your code can work with the same data using either **DataObjects for .NET** or ADO.NET interface, whichever is most suitable for the task at hand.
- **ADO.NET Integration**
  Continue to do everything with your data that you can do with ADO.NET, including features not supported in **DataObjects for .NET** (for example, DataSet.Merge).
- **Import and Export XML Data**
  Work with XML data in **DataObjects for .NET** and import and export your data in self-describing XML to and from other tools and programs.
- **Import and Export ADO.NET Data**
  Export/import your data from/to **DataObjects for .NET** in ADO.NET DataSet format.

- **Programmatic Customization**
  DataObjects for .NET supports an extensive set of events enabling full programmatic customization.

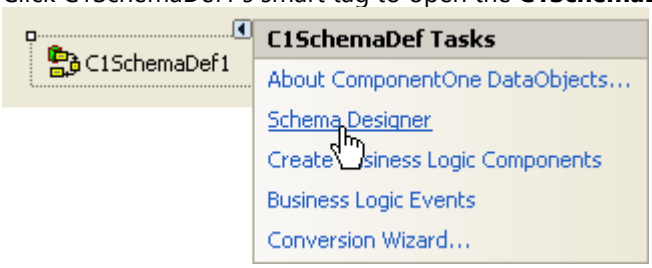
DataObjects for .NET Quick Start

This quick start tutorial shows you how to add the C1DataSet and C1SchemaDef components to your form, create a simple schema, and connect to a data source. By following the steps outlined in the quick start, you will be able create a rich data view. Note that the quick start uses **True DBGrid for WinForms** and the **C1NWind.mdb** database (standard MS Access sample database included in Visual Studio) which, unless the product was installed to another location, is installed in the **Documents** or **MyDocuments** folder at **ComponentOne Samples\Common** directory by default.

Step 1 of 4: Setting up the Form and Importing a Database Structure

In this step you'll set up the form, add **DataObjects for .NET** components, and import a database structure using the **Schema Designer**. Complete the following steps:

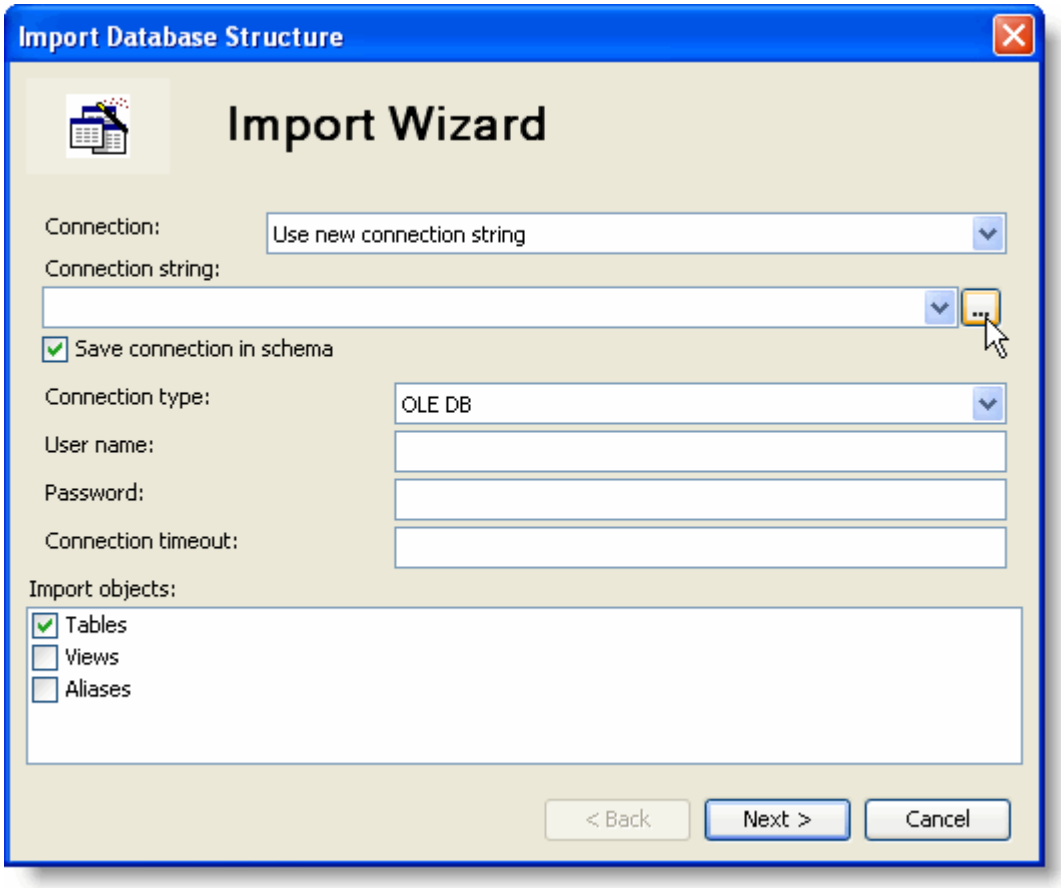
1. Create a new .NET 2.0 project.
2. Navigate to the Toolbox and add the **C1DataSet**, **C1SchemaDef**, and **C1TrueDBGrid** components to the form.
3. Click C1TrueDBGrid1's **smart tag** to open the **C1TrueDBGrid Tasks** menu and select **Dock in parent container**.
4. Click C1SchemaDef1's smart tag to open the **C1SchemaDef Tasks** menu, and select **Schema Designer**.



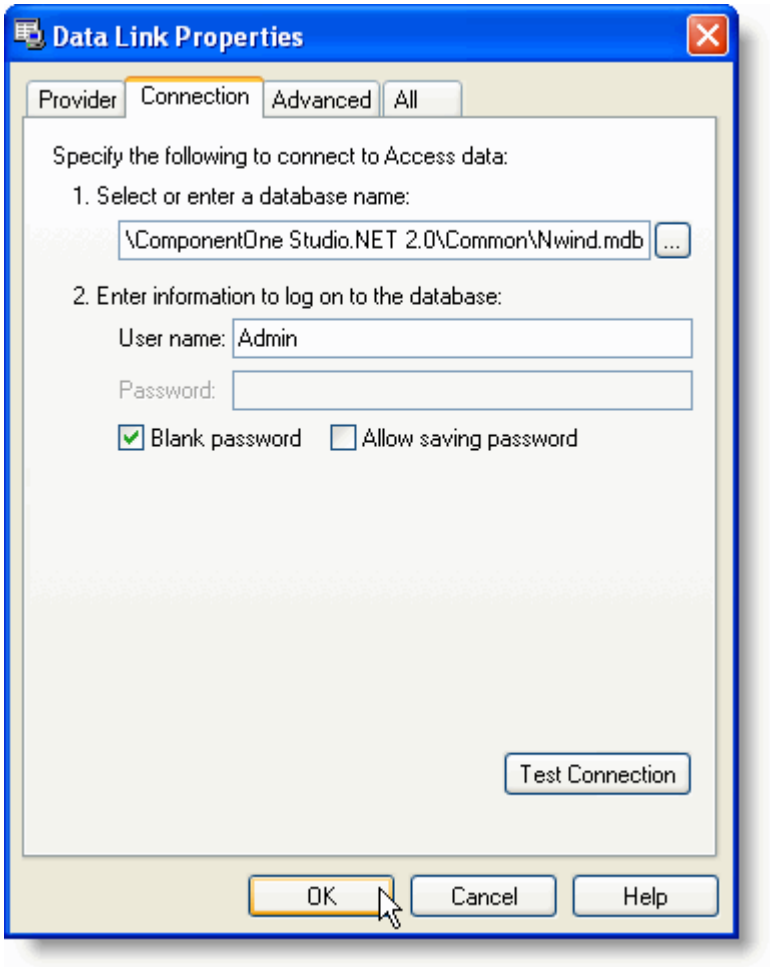
The **DataObjects Schema Designer** opens and the **Import Database Structure** Wizard appears. For more information about the **C1SchemaDef Tasks** menu, see C1SchemaDef Tasks and Context Menus.

5. Click the **ellipsis** button to the right of the connection string. The standard OLE DB **Data Link Properties** dialog box opens.

> **Note:** Select **Schema | Import** database structure in the **Schema Designer** to open the **Import Wizard** if it does not automatically appear.



6. In the **Connection** tab, click the **ellipsis** button under **Select or enter a database name** to locate a database.
7. Locate the **C1NWind.mdb** database (by default installed in **Documents\ComponentOne Samples\Common**) and click **Open**. If desired you can test the connection now by clicking the **Test Connection** button.
8. Click **OK** to close the **DataLink Properties** dialog box and import the connection string.



9. Click **Next**. In this window, you can select the tables to import into the schema.
10. To select all of the available tables, click the **>>** button to add all the tables to the schema, and click **Finish**. The **Import Wizard** creates the schema and closes.

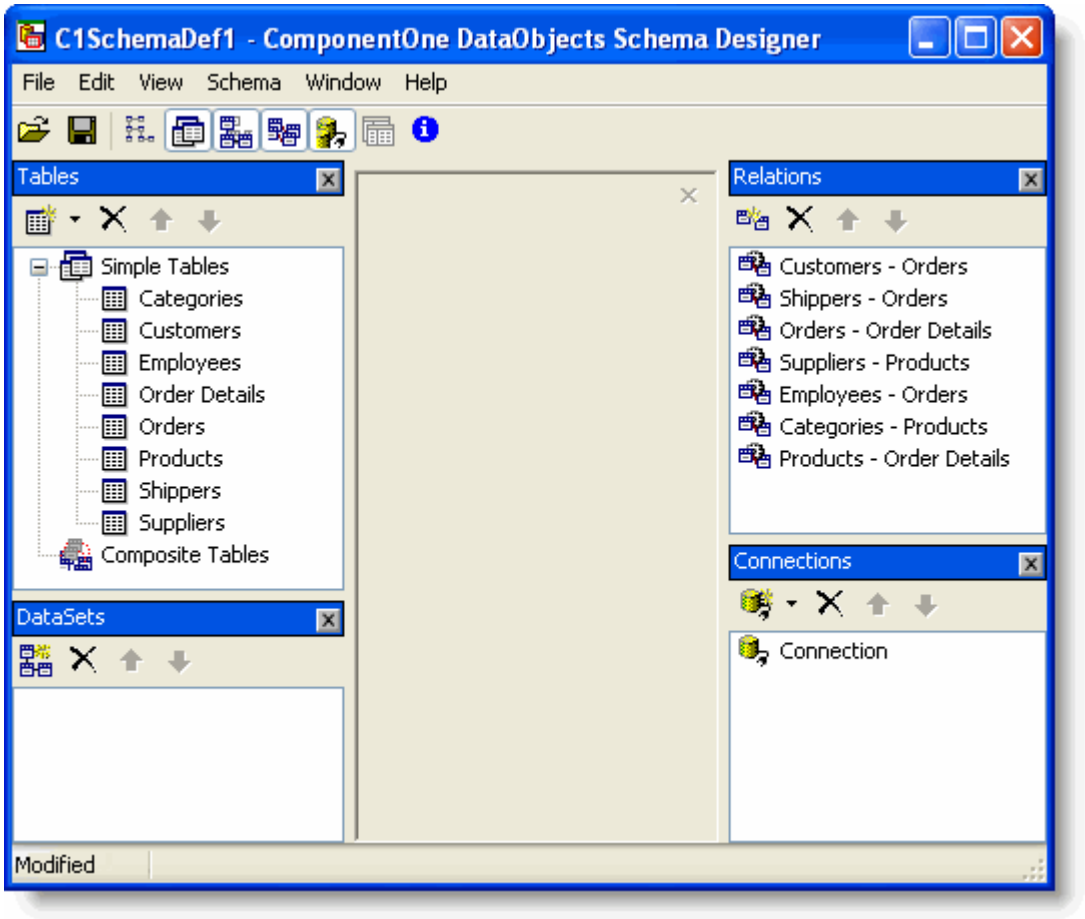
You've just completed step 1 of the quick start guide; you've set up the form, added **DataObjects for .NET** components, and imported a database structure. In the next step you'll create a simple data schema using the **Schema Designer**.

Step 2 of 4: Creating a Data Schema using the Schema Designer

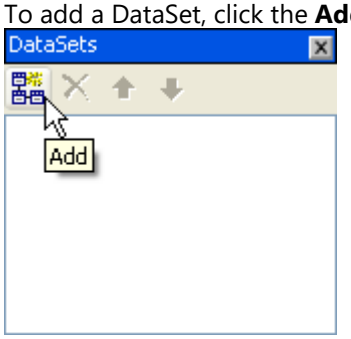
Now that you've imported a database connection, you can create a simple data schema. Schemas are the basis of **DataObjects for .NET** and allow you to manipulate the data in very useful ways. As you'll discover, creating a data schema is easy using the **DataObjects Schema Designer**. For more information about the **Schema Designer**, see Creating and Customizing the Schema Using the Schema Designer.

To create a simple data schema complete the following steps:

1. If the **DataObjects Schema Designer** is not open, click C1SchemaDef1's smart tag to open the **C1SchemaDef Tasks** menu, and select **Schema Designer**.
   Notice that the tables you added in step 1 now appear in the **Tables** window in the **Schema Designer** and that the relations between these tables appear in the **Relations** window.

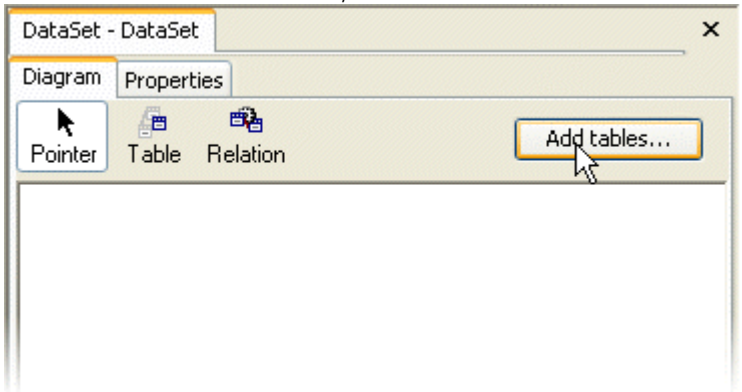


2. To add a DataSet, click the **Add** button on the **DataSets toolbar**.



A new DataSet with the default name **DataSet** will appear in the window and a **DataSet** tab will appear in the center of the **Schema Designer**.

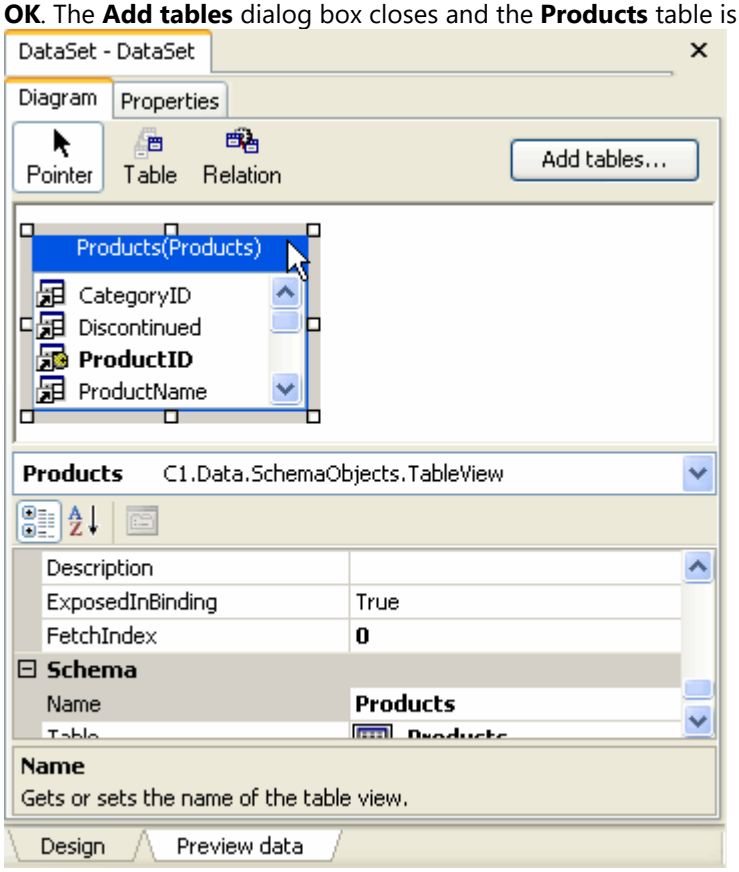
3. To add a table to the DataSet, select the **Add tables** button in the **DataSet** tab.



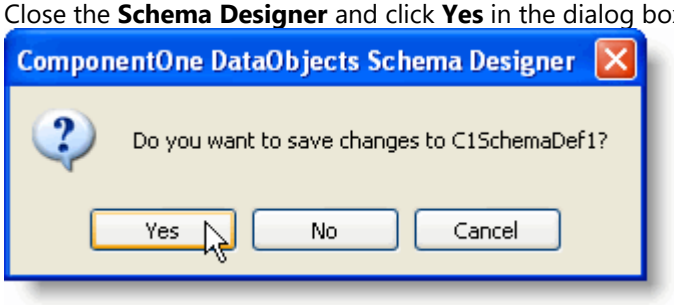
The **Add tables** dialog box opens.

4. Select **Products** from the Existing tables list, click the > button to add the table to the DataSet, and click

OK. The **Add tables** dialog box closes and the **Products** table is added to the DataSet.



5. Close the **Schema Designer** and click **Yes** in the dialog box to save the schema.

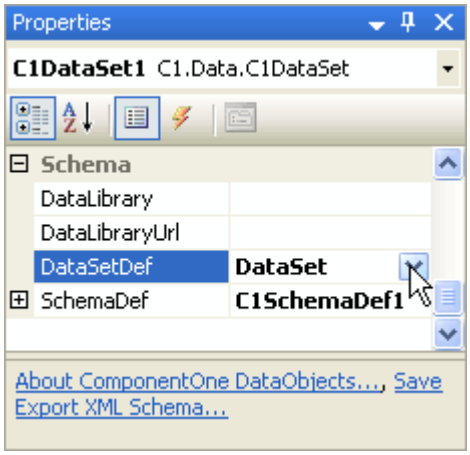


You've completed step 2 of the quick start guide and created a simple data schema. In the next step you'll bind the data source to the grid.

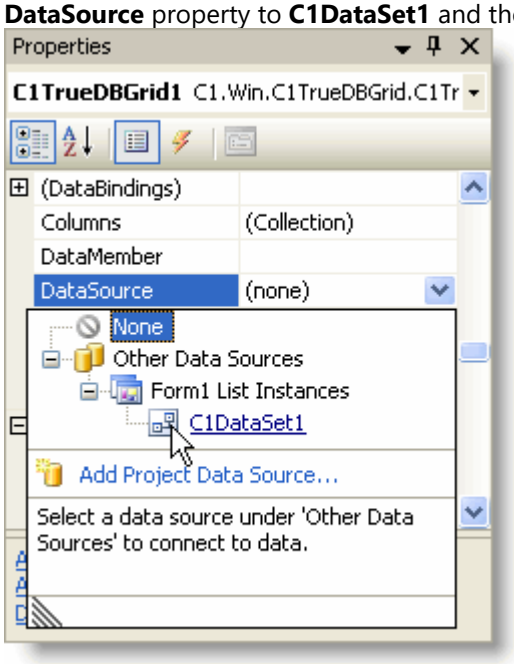
Step 3 of 4: Binding the Data Source to a Grid

Now that you've created a simple data schema, the next thing to do is to bind the **DataObjects for .NET** data source to the grid. Complete the following steps:

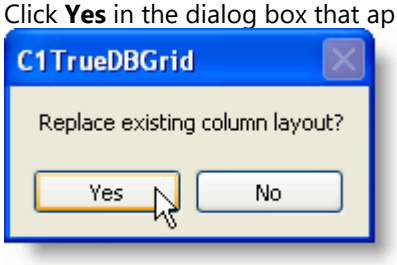
1. Select the **C1DataSet** control you added to the form and in the **C1DataSet1** Properties window, set the **SchemaDef** property to **C1SchemaDef1** and set the **DataSetDef** property to **DataSet**.



2. Select the **C1TrueDBGrid** control you added to the form and in the **C1TrueDBGrid1** Properties window set the **DataSource** property to **C1DataSet1** and the **DataMember** property to **Products**.



3. Click **Yes** in the dialog box that appears asking if the column layout should be replaced.

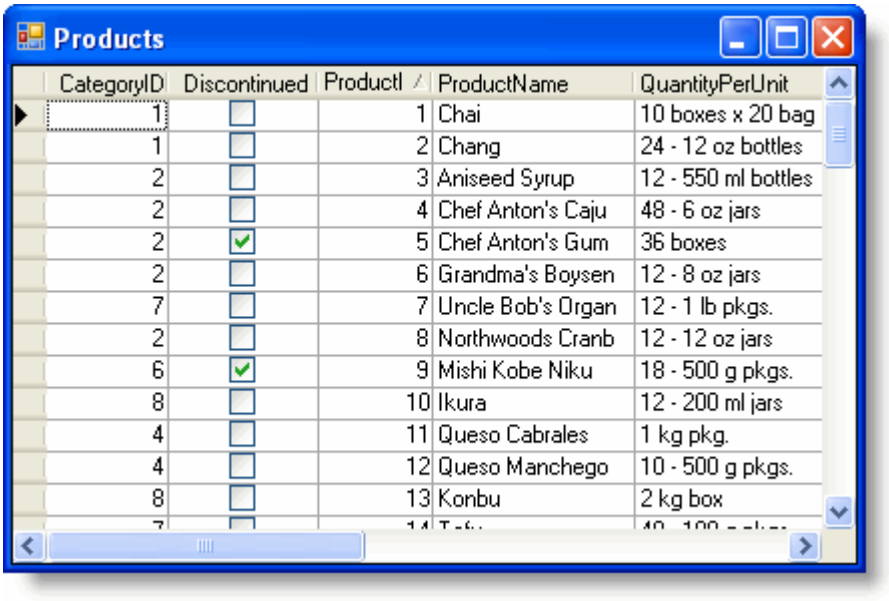


You've just bound the **DataObjects for .NET** data source to the grid and completed step 3 of the quick start guide. In the next step you'll run the program.

Step 4 of 4: Running the Program

In the previous steps you've added the controls, imported the database structure, and created a simple data schema without writing a single line of code. The only thing left to do is to run your program and see **DataObjects for .NET** in action.

Run the program and observe that the data view that you created will be displayed in the grid:



You can continue exploring **DataObjects for .NET** by returning to the **Schema Designer**, making changes, and seeing how those changes are reflected in the grid. For more information, see Creating and Customizing the Schema using the Schema Designer.

Congratulations, you've completed the **DataObjects for .NET** quick start guide and are now up and running with **DataObjects for .NET**! For detailed tutorials, see DataObjects for .NET Tutorials and DataObjects for .NET Express Tutorials.

DataObjects for .NET Top Tips

This topic lists tips and best practices that may be helpful when working with **DataObjects for .NET**. The following tips were compiled from frequently asked user questions posted in the C1DataObjects newsgroup and forum.

Tip 1: Accessing the field values of the current row

To access the field values of the current row, get the current row from the **CurrencyManager** object (which can be obtained via the **BindingContext** property) and then invoke the FromDataItem method to obtain the currently selected row's corresponding C1DataRow object. If you need typed access you can pass a C1DataRow object to the static **Obj()** method of the corresponding TableView row object from the **DataLibrary.DataObjects** assembly.

For example:

To write code in Visual Basic

VB

```
Imports C1.Data
Imports DataLibrary.DataObjects.MyDataSet

Private Sub EditButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles EditButton.Click

    Dim cm As CurrencyManager cm = _
        CType(BindingContext(C1DataTableSource1), CurrencyManager)

    If cm IsNot Nothing AndAlso cm.Count > 0 AndAlso cm.Position >= 0 Then
        Dim row As C1DataRow = C1DataRow.FromDataItem(cm.Current)

        ' untyped access
        row("UnitsInStock") = 1234

        ' typed access
        Dim product As ProductsRow = ProductsRow.Obj(row)

        product.UnitPrice += 4D
        cm.EndCurrentEdit()
        End If
End Sub
```

To write code in C#

C#

```
using C1.Data;
using DataLibrary.DataObjects.MyDataSet;

private void EditButton_Click(object sender, EventArgs e)
{
    CurrencyManager cm = (CurrencyManager)BindingContext[c1DataTableSource1];
    if (cm != null && cm.Count > 0 && cm.Position >= 0)
    {
```

```
        C1DataRow row = C1DataRow.FromDataItem(cm.Current);

        // untyped access
        row["UnitsInStock"] = 1234;

        // typed access
        ProductsRow product = ProductsRow.Obj(row);
        product.UnitPrice += 4m;

        cm.EndCurrentEdit();
    }
}
```

Tip 2: Cloning a data row

You can use the autoincremented primary key to clone a data row. Suppose you have to add a copy of the currently selected row to the same data table, for example when the user wants to add a new row based on the currently selected data row. To do so, you should create a new row, then fill it out with data from the original data row except for the primary key and the unique key fields.

For example:

To write code in Visual Basic

VB

```
Private Sub CloneRowButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles CloneRowButton.Click

    ' Get the currently selected row
    Dim CurrentRow As C1DataRow = C1DataRow.FromDataItem( _
        BindingContext(C1DataSet1, "Shippers").Current)

    ' Get data from the original row as an item array
    Dim Arr() As Object = CurrentRow.ItemArray

    ' Create a new data row
    Dim NewRow As C1DataRow = C1DataSet1.TableViews("Shippers").AddNew()

    ' Preserve autoincremented primary key
    Arr(0) = NewRow("ShipperID")

    ' Fill out the unique key fields
    Arr(1) = CompanyName_TextBox.Text

    ' Assign the item array
    NewRow.ItemArray = Arr

    ' End the edit on the new row
    NewRow.EndEdit()
End Sub
```

To write code in C#

C#

```
private void CloneRowButton_Click(object sender, EventArgs e)
{
    // Get the currently selected row
    C1DataRow currentRow = C1DataRow.FromDataItem(
        BindingContext[C1DataSet1, "Shippers"].Current);

    // Get data from the original row as an item array
    object[] arr = currentRow.ItemArray;

    // Create a new data row
    C1DataRow newRow = C1DataSet1.TableViews["Shippers"].AddNew();

    // Preserve autoincremented primary key
    arr[0] = newRow("ShipperID");

    // Fill out the unique key fields
    arr[1] = CompanyName_TextBox.Text;

    // Assign the item array
    newRow.ItemArray = arr;

   // End the edit on the new row
    newRow.EndEdit();
}
```

Tip 3: Specify a date interval in the row filter condition

You may want to specify a date constant in the client-side filter condition or how to display the set of rows between two given dates. The following code demonstrates one possible example of doing so:

To write code in Visual Basic

VB

```
Private Sub FilterButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles FilterButton.Click

    Dim date1 As DateTime = From_DateTimePicker.Value.Date
    Dim date2 As DateTime = To_DateTimePicker.Value.Date.AddDays(1)

    C1ExpressView1.RowFilter = String.Format( _
        "OrderDateTime >= #{0}-{1}-{2}# AND OrderDateTime < #{3}-{4}-{5}#",_
        date1.Year, date1.Month, date1.Day, _
        date2.Year, date2.Month, date2.Day)
End Sub
```

To write code in C#

C#

```
private void FilterButton_Click(object sender, EventArgs e)
{
    DateTime date1 = From_DateTimePicker.Value.Date;
    DateTime date2 = To_DateTimePicker.Value.Date.AddDays(1);

    C1ExpressView1.RowFilter = string.Format(
        "OrderDateTime >= #{0}-{1}-{2}# AND OrderDateTime < #{3}-{4}-{5}#",
        date1.Year, date1.Month, date1.Day,
        date2.Year, date2.Month, date2.Day);
}
```

Tip 4: Filter data on the server side if possible

When working with **DataObjects for .NET** you can filter data either on the client side (using the RowFilter property) or on the server side. Filtering on the server minimizes the amount of data passed between the server and client and improves performance.

There are two ways you can set the filter conditions:

- You can pass filters to the **C1DataSet.Fill()** method. You would also have to set the FillOnRequest property to **False** to avoid automatic filling the dataset at startup. For example:

  To write code in Visual Basic

  VB

  ```
  Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load

      Dim filters As New FilterConditions()
      filters.Add(New FilterCondition(Nothing, "Suppliers", _
          "Country = 'France'"))
      filters.Add(New FilterCondition(Nothing, "Products", _
          "SupplierID IN (SELECT SupplierID From Suppliers WHERE " + _
          "Country = 'France')"))
      C1DataSet1.Fill(filters, False)
  End Sub
  ```

  To write code in C#

  C#

  ```
  private void Form1_Load(object sender, EventArgs e)
  {
      FilterConditions filters = new FilterConditions();
      filters.Add(new FilterCondition(null, "Suppliers",
          "Country = 'France'"));
      filters.Add(new FilterCondition(null, "Products",
          "SupplierID IN (SELECT SupplierID From Suppliers WHERE " +
          "Country = 'France')"));
      C1DataSet1.Fill(filters, False);
  }
  ```

- Filters can be added in the BeforeFill event handler. For example:

To write code in Visual Basic

```
VB
Private Sub C1DataSet1_BeforeFill(ByVal sender As System.Object, _
  ByVal e As C1.Data.FillEventArgs) Handles C1DataSet1.BeforeFill

    e.Filter.Add(New FilterCondition(Nothing, "Suppliers", _
        "Country = 'France'"))
    e.Filter.Add(New FilterCondition(Nothing, "Products", _
        "SupplierID IN (SELECT SupplierID FROM Suppliers WHERE " + "Country =
'France'")))
End Sub
```

To write code in C#

```
C#
private void C1DataSet1_BeforeFill(object sender, C1.Data.FillEventArgs e)
{
    e.Filter.Add(new FilterCondition(null, "Suppliers",
        "Country = 'France'"));
    e.Filter.Add(new FilterCondition(null, "Products",
        "SupplierID IN (SELECT SupplierID FROM Suppliers WHERE " + "Country =
'France'")));
}
```

Tip 5: Share the same connection with C1DataObjects when performing a batch update

If you want to execute some SQL statements that update a group of rows you can obtain the connection object from **DataObjects for .NET**. It is also possible to start an SQL transaction using **C1.Data.SchemaObjects.Connection** and perform all data manipulations as a single logical unit.

For example:

To write code in Visual Basic

```
VB
Imports C1.Data
Imports C1.Data.SchemaObjects
Imports System.Data.OleDb

Private Sub BatchUpdateButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Button1.Click

    Dim N As Int32 = 0
    Dim c As Connection = C1DataSet1.Schema.Connections(0)
    c.Open()
    c.BeginTransaction()
    Try
        Dim oc As OleDbConnection = CType(c.DbConnection, OleDbConnection)
```

```
        Dim ot As OleDbTransaction = CType(c.DbTransaction, OleDbTransaction)

        Dim cmd As OleDbCommand

        cmd = New OleDbCommand("UPDATE Products SET " + _
            "ReorderLevel = ReorderLevel + 5 WHERE UnitPrice < 5", oc, ot)
        N += cmd.ExecuteNonQuery()
        cmd.Dispose()

        cmd = New OleDbCommand("UPDATE Products SET ReorderLevel = " + _
            "ReorderLevel - 3 WHERE ReorderLevel > 40", oc, ot)
        N += cmd.ExecuteNonQuery()
        cmd.Dispose()

        cmd = New OleDbCommand("INSERT INTO Products (ProductName, " + _
            "Discontinued, UnitPrice) VALUES ('N product', FALSE, 135)", _
            oc, ot)
        N += cmd.ExecuteNonQuery()
        cmd.Dispose()

        c.CommitTransaction()
    Catch ex As Exception
        c.RollbackTransaction()
    End Try

    ' You shouldn't close the connection here. It is already
    ' closed within the CommitTransaction() method.
    '

                c.Close()

    C1DataSet1.Fill()
    MessageBox.Show(N.ToString() + " rows affected.")
End Sub
```

To write code in C#

C#

```
using C1.Data;
using C1.Data.SchemaObjects;
using System.Data.OleDb;

private void BatchUpdateButton_Click(object sender, EventArgs e)
{
    int n = 0;
    Connection c = C1DataSet1.Schema.Connections[0];
    c.Open();
    c.BeginTransaction();
    try
    {
        OleDbConnection oc = (OleDbConnection)c.DbConnection;
```

```
        OleDbTransaction ot = (OleDbTransaction)c.DbTransaction;

        OleDbCommand cmd;

        cmd = new OleDbCommand("UPDATE Products SET " +
            "ReorderLevel = ReorderLevel + 5 WHERE UnitPrice < 5", oc, ot);
        N += cmd.ExecuteNonQuery();
        cmd.Dispose();

        cmd = new OleDbCommand("UPDATE Products SET ReorderLevel = " +
            "ReorderLevel - 3 WHERE ReorderLevel > 40", oc, ot);
        N += cmd.ExecuteNonQuery();
        cmd.Dispose();

        cmd = new OleDbCommand("INSERT INTO Products (ProductName, " +
            "Discontinued, UnitPrice) VALUES ('N product', FALSE, 135)",
            oc, ot);
        N += cmd.ExecuteNonQuery();
        cmd.Dispose();

        c.CommitTransaction();
    }
    catch (Exception ex)
    {
        c.RollbackTransaction();
    }

    // You shouldn't close the connection here. It is already
    // closed within the CommitTransaction() method.
    //
    // c.Close();

    C1DataSet1.Fill();

    MessageBox.Show(N.ToString() + " rows affected.");
}
```

Tip 6: How to change the connection string on the fly

If you choose to, you can change the connection string on the fly. To do so, the connection string for the data schema must first be specified in the app.config file.

For example, a connection string to the Northwind database might appear as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MyConnStr" value="Data Source=(local)\SQLExpress;
Initial Catalog=Northwind;Integrated Security=True" />
  </appSettings>
...
</configuration>
```

Then you would attach the following handler to the CreateSchema event:

To write code in Visual Basic

VB

```
'Add the following import statement to the top of the code:
Imports System.Configuration

' Add the C1SchemaDef_CreateSchema event handler:
Private Sub C1SchemaDef1_CreateSchema(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles C1SchemaDef1.CreateSchema
    Dim s As String = ConfigurationManager.AppSettings("MyConnStr")

    C1SchemaDef1.Schema.Connections(0).ConnectionString = s
End Sub
```

To write code in C#

C#

```
// Add the following import statement to the top of the code:
using System.Configuration;

// Add the C1SchemaDef_CreateSchema event handler:
private void C1SchemaDef1_CreateSchema(object sender, EventArgs e)
{
    string s = ConfigurationManager.AppSettings["MyConnStr"];
    C1SchemaDef1.Schema.Connections[0].ConnectionString = s;
}
```

DataObjects for .NET (Enterprise Edition)

The **DataObjects for .NET Enterprise Edition** is a comprehensive data and business objects framework that can be used in .NET applications of any range, scalability and architecture, from simple desktop applications to classic client-server applications to 3-tier distributed applications and enterprise-wide business object libraries. In fact, it is in 3-tier applications and enterprise-wide business where **DataObjects for .NET Enterprise Edition** is the most useful, the most productivity enhancing and time saving.

DataObjects for .NET and ADO.NET

DataObjects for .NET is based on Microsoft ADO.NET technology and enhances it to empower and simplify database application development. **DataObjects for .NET** has close connections with ADO.NET that programmers familiar with ADO.NET can utilize:

- **DataObjects for .NET** programmatic object model closely follows one of ADO.NET, so it will look very familiar to those who are used to the ADO.NET object model. For example, C1DataSet is similar to ADO.NET DataSet, C1DataTable to ADO.NET DataTable, and so on.
- The relation with ADO.NET goes beyond object model similarity. In fact, **DataObjects for .NET** stores data in ADO.NET DataSet and that internal storage is accessible via the StorageDataSet property. This enables a powerful combination of **DataObjects for .NET** and ADO.NET in the same code. Your code can work with the same data using either **DataObjects for .NET** or ADO.NET interface, whichever is most suitable for the task at hand. This duality is very important, because it allows you to:
    - Do everything with your data that you can do with ADO.NET, including features that are not supported in **DataObjects for .NET** (for example, DataSet.Merge).
    - Export/import your data from/to **DataObjects for .NET** in ADO.NET DataSet format.
    - Work with XML data in **DataObjects for .NET** and import/export data in self-describing XML to/from other tools and programs.

See Working with ADO.NET Dataset for details on how to use this **DataObjects for .NET**-ADO.NET bridge.

Differences between ADO.NET and DataObjects for .NET

The following list explains the differences between ADO.NET and **DataObjects for .NET**:

- **DataObjects for .NET** fully supports multi-table rowsets (composite tables) automatically enforcing data relations without manual coding. For example, changing a *CustomerID* field will automatically change the corresponding *CustomerName* field in the same row, although it is stored in a separate table.
- With an innovative virtual mode technology, **DataObjects for .NET** allows you to use large datasets in .NET Windows Forms applications, a feature that is not supported in Visual Studio and ADO.NET without **DataObjects for .NET**.
- **DataObjects for .NET** completely automates database updates. There is no need to use ADO.NET DataAdapter or other special components. Database updates are performed without manual coding. **DataObjects for .NET** can update the database even when multiple and interrelated changes have been made to multiple tables.
- By setting a single property, UpdateLeavingRow, you can make **DataObjects for .NET** update the database immediately after the user changes a row. This optional feature is commonly used in desktop and classic client-server applications. Standard ADO.NET does not support this feature.
- **DataObjects for .NET** supports an extensive set of events enabling full programmatic customization.
- **DataObjects for .NET Enterprise Edition** uses the standard business object paradigm to allow you to develop business logic components (data libraries) and reuse them in multiple client projects. It provides clear separation of business and data logic from the presentation (GUI) layer.
- **DataObjects for .NET Enterprise Edition** allows you to create a centralized and reusable repository of data schema and business logic (data libraries) used in applications throughout the enterprise.

- **DataObjects for .NET Enterprise Edition** completely automates the task of developing distributed 3-tier Web-based applications. No special server-based code is necessary, and making your application distributed becomes a simple matter of deployment configuration.

Schema Objects

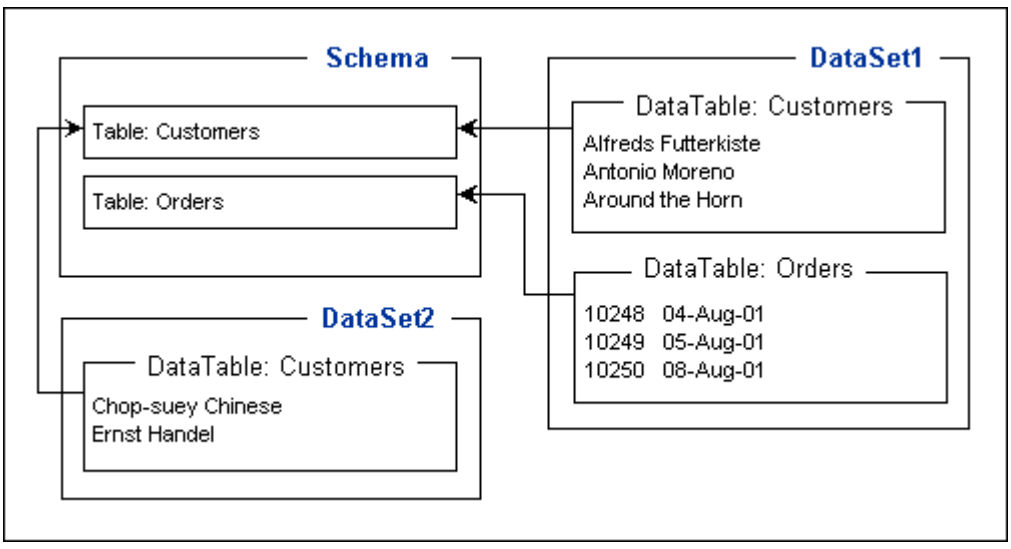
A *Schema* is the basis and starting point of **DataObjects for .NET** development. It contains data structure information, defining basic entities, such as tables and relations, with their properties. Normally, a schema is initially created by importing a database structure using the **Import Wizard** in the **Schema Designer**, and then customized in the **Schema Designer** to suit your business logic needs. A schema can also be imported from an ADO.NET schema. For additional information, see Converting Schema from Other Sources.

A schema is stored in a C1SchemaDef component, in the form of an XML string in the .resx resource file. To create or edit a schema, open the context menu of a C1SchemaDef component and select **Schema Designer** from the menu.

A **DataObjects for .NET** data library always contains a single schema, that is, a single C1SchemaDef component. An application that uses **DataObjects for .NET** directly, without a data library (direct client) is allowed to have multiple schemas, each stored in its own C1SchemaDef component.

In most cases, a schema is created at design time in the **Schema Designer**, and used at run time without modification. However, all schema objects are programmatically accessible at run time through the schema object model, with the Schema class being the root of that object model. Using this object model, it is also possible to modify the schema or even define it from scratch at run time, using the CreateSchema event. This, however, is rarely needed, so you will not need to use schema objects programmatically unless in special cases. Most of the time, you will only deal with schema objects at design time, in the **Schema Designer**.

Schema objects, classes comprising the Schema object model, are structure-only, they do not contain actual data. They are created when a C1SchemaDef component is initialized and immediately become accessible through its Schema property, before any data is fetched. Moreover, they do not represent the actual data, they only represent the structure. The actual data is fetched and managed by another component, C1DataSet. There can be multiple C1DataSet components bound to a single schema.



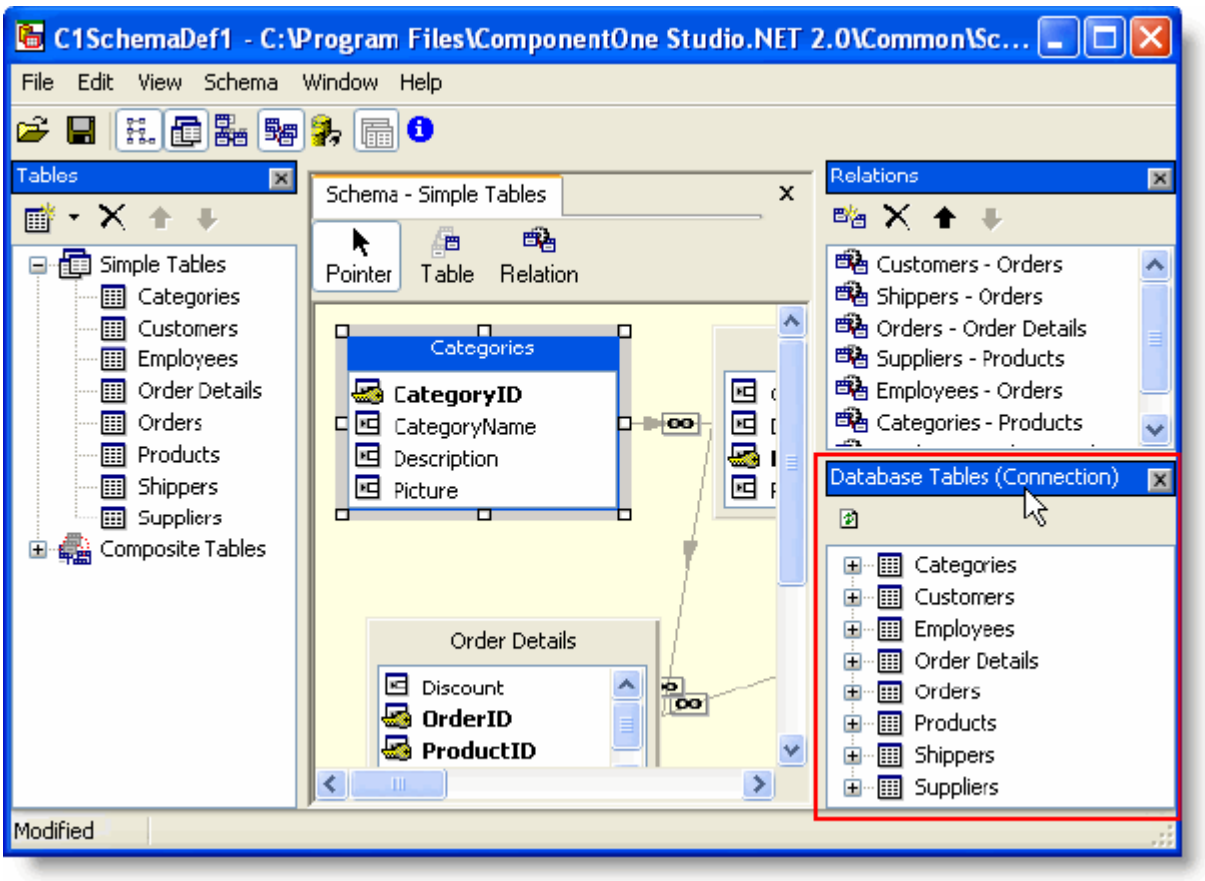
A C1DataSet component holds a collection of C1DataTable objects, each one representing a set of rows existing in this data set for a particular schema object. In fact, a C1DataSet has two collections of C1DataTable objects: Tables for data contents of Table schema objects, and TableViews for data contents of TableView schema objects. The SchemaTable property returns the schema object (either Table or TableView) defining the structure of this C1DataTable.

Importing Database Structure

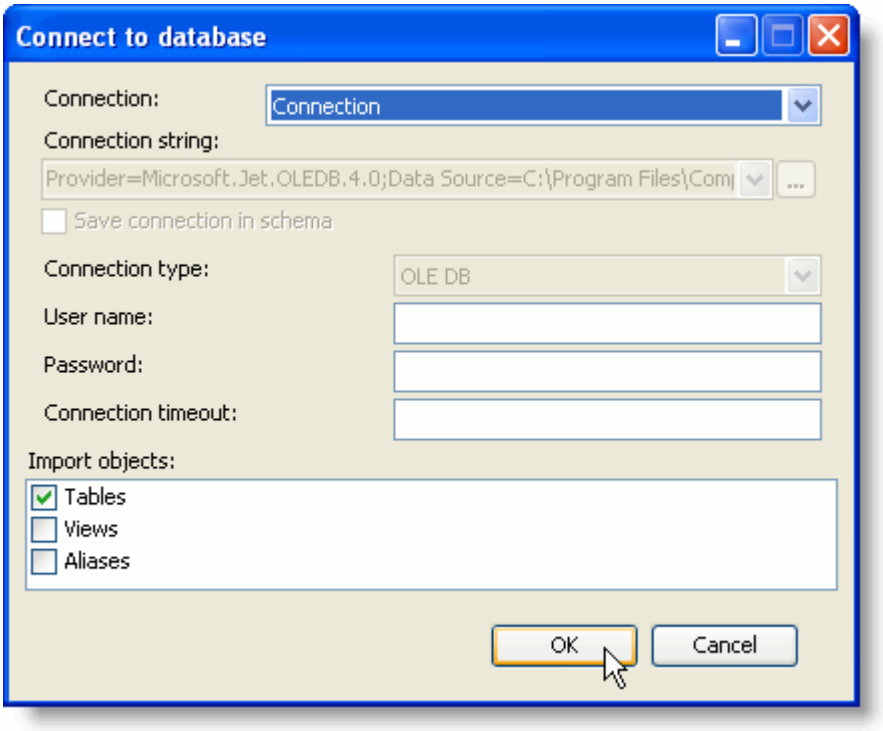
The **DataObjects Schema Designer** provides an **Import Wizard** that creates a schema based on the structure information stored in a database. It creates simple tables based on database tables, and simple relations between them based on foreign key constraints stored in the database. Tables are created with fields according to the field structure of the database tables. Field properties, such as DataType, PrimaryKey and others are set according to the database field properties. Later, you can customize table fields, change their properties, delete unnecessary fields, change the fields order, and add new fields, not necessarily based on database table fields.

Importing the database structure also creates a database connection (in the **Connections** window of the Schema Designer), that you can use later to connect to the database while editing the schema. For more information on importing a database structure, see Creating a Schema with the Import Wizard.

After you have imported the database structure, you can modify the schema, add tables (simple and composite), create relations, and so on. After import, the **Schema Designer** is connected to the database, so the **Database Tables** window is filled with the list of all database tables with their fields; this is highlighted in the image below:



To connect to the database, select **Connect to database** in the **Schema** menu, specify the connection and optional parameters in the **Connect to database** dialog box, and click **OK**.

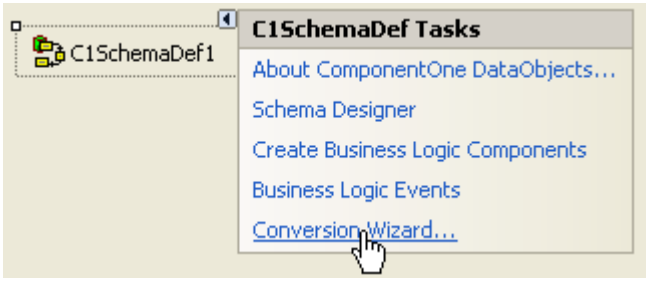


Note: It is not necessary to be connected to the database in order to work in the **Schema Designer**, although a live connection has some advantages. For example, if the designer is connected to the database, the DbTableName property of a table and the DbFieldName property of a field show the lists of available tables and fields, respectively. Also, in connected mode, it is possible to create schema tables by a drag-and-drop operation from the Schema graph window.

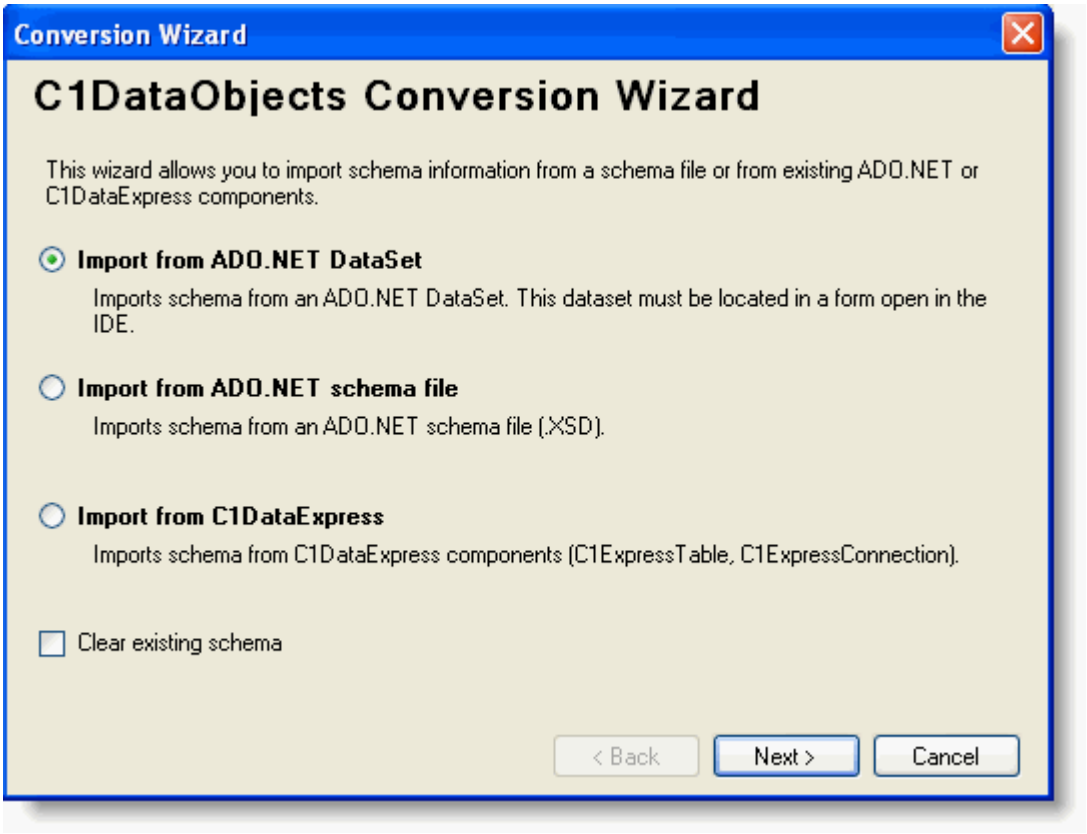
Converting Schema from Other Sources

The **C1DataObjects Conversion Wizard** allows you to import (migrate) schema information from an existing ADO.NET dataset or from an XML schema file (XSD). You can convert an existing database project from ADO.NET to **DataObjects for .NET** without recreating schema information from scratch.

To invoke the wizard, select **Conversion Wizard** in the **C1SchemaDef Tasks** menu of a C1SchemaDef component.



This opens the **Conversion Wizard** dialog box.



Here you can choose the source of the schema information to be imported. It can be an ADO.NET Dataset component located in a form or another file that is currently open in the IDE, or it can be an XSD file containing ADO.NET dataset structure in XML form. Necessary objects (tables, relations, and a data set) will be created automatically in the schema based on the imported information. Once the schema is imported, you can open the **Schema Designer** to view and edit the imported schema.

Additionally, the **C1DataObjects Conversion Wizard** can facilitate migrating a project from **DataObjects for .NET Express** to **DataObjects for .NET Enterprise**. In this case, choose a **DataObjects for .NET Express** component as the source for your schema. It can be either a standalone C1ExpressTable component or a C1ExpressConnection component with **C1ExpressTable** components attached to it.

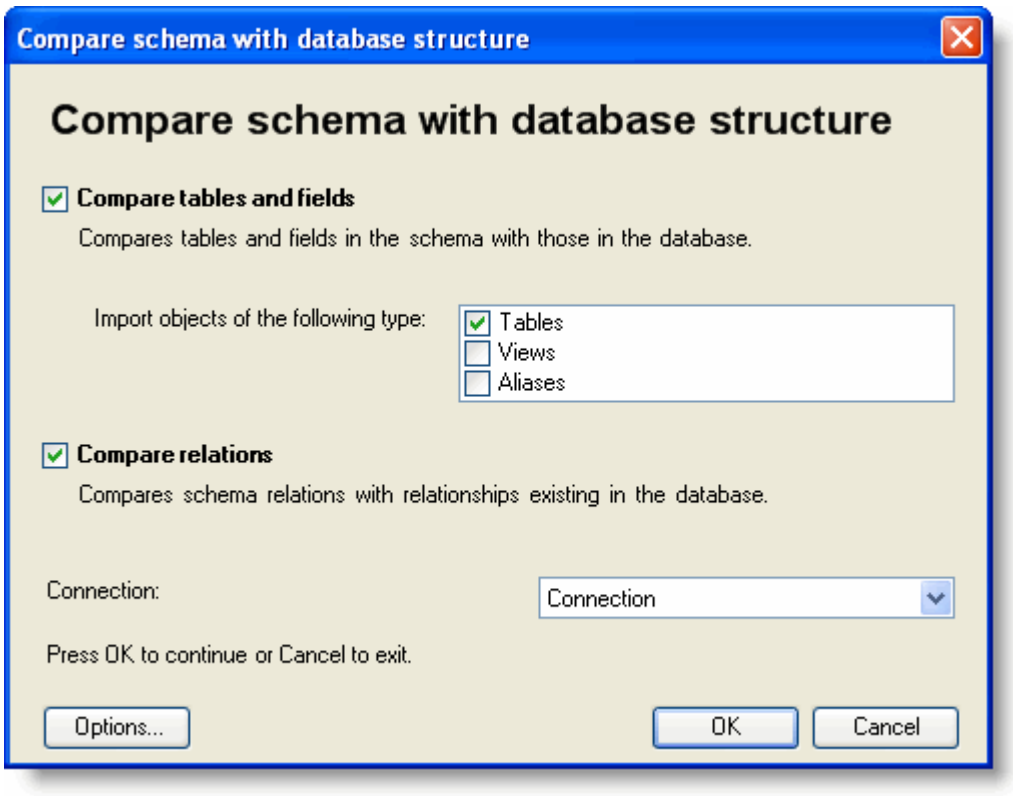
For additional information, see Importing Schema Information using the Conversion Wizard.

Database Structure Evolution and the Schema

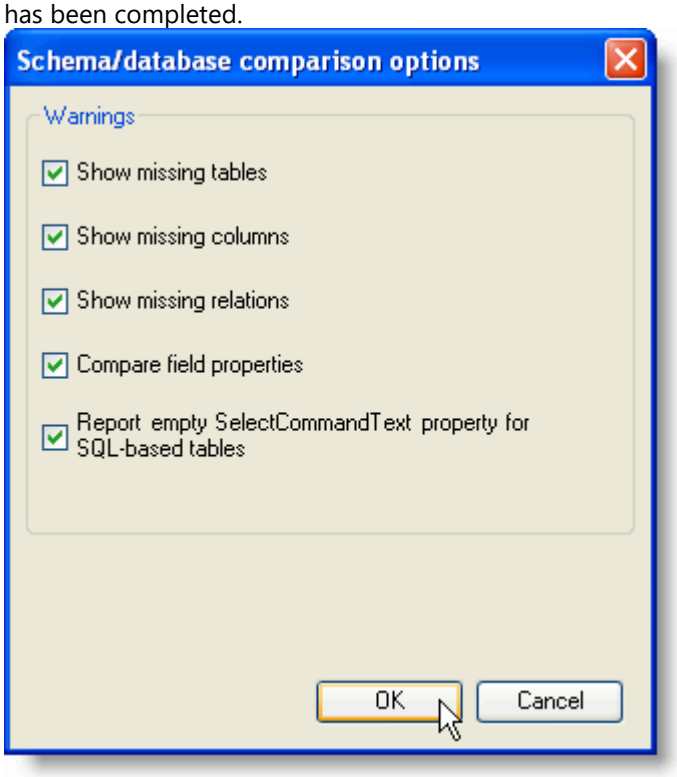
After importing database structure to a schema, the database usually changes with time as the development process progresses. As a result, the schema may be no longer in sync with the database structure. Such discrepancies cause problems at run time and they are hard to keep track of manually. Special **Schema Designer** menu items and other features can help you keep the schema in sync with the database.

The schema comparison/differences tool scans the schema and database structure and shows differences between them in the **Output** window. To use this tool, complete the following steps:

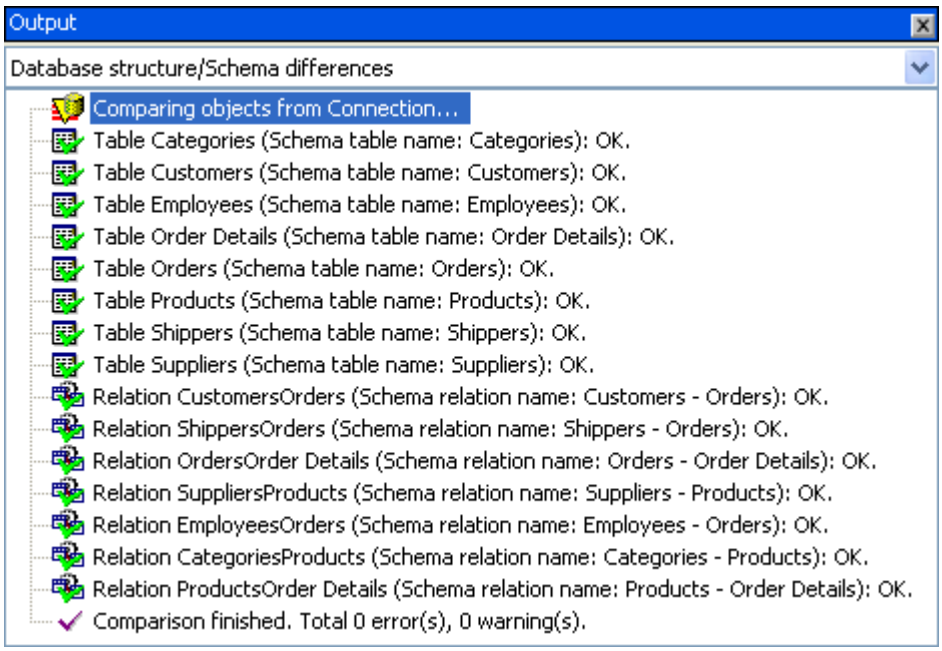
1. Select **Compare schema with database structure** from the Schema menu. The **Compare schema with database structure** window appears.



2. Select the checkbox next to **Compare tables and fields** to include tables and fields in the comparison. You can also check which types of objects to import from the database: tables, views, aliases.
3. Select the checkbox next to **Compare relations** to include relations in the comparison.
4. Select the database connection from the drop-down box next to **Connection**.
5. Click the **Options** button to select the warnings to be displayed in the Output window once the comparison has been completed.



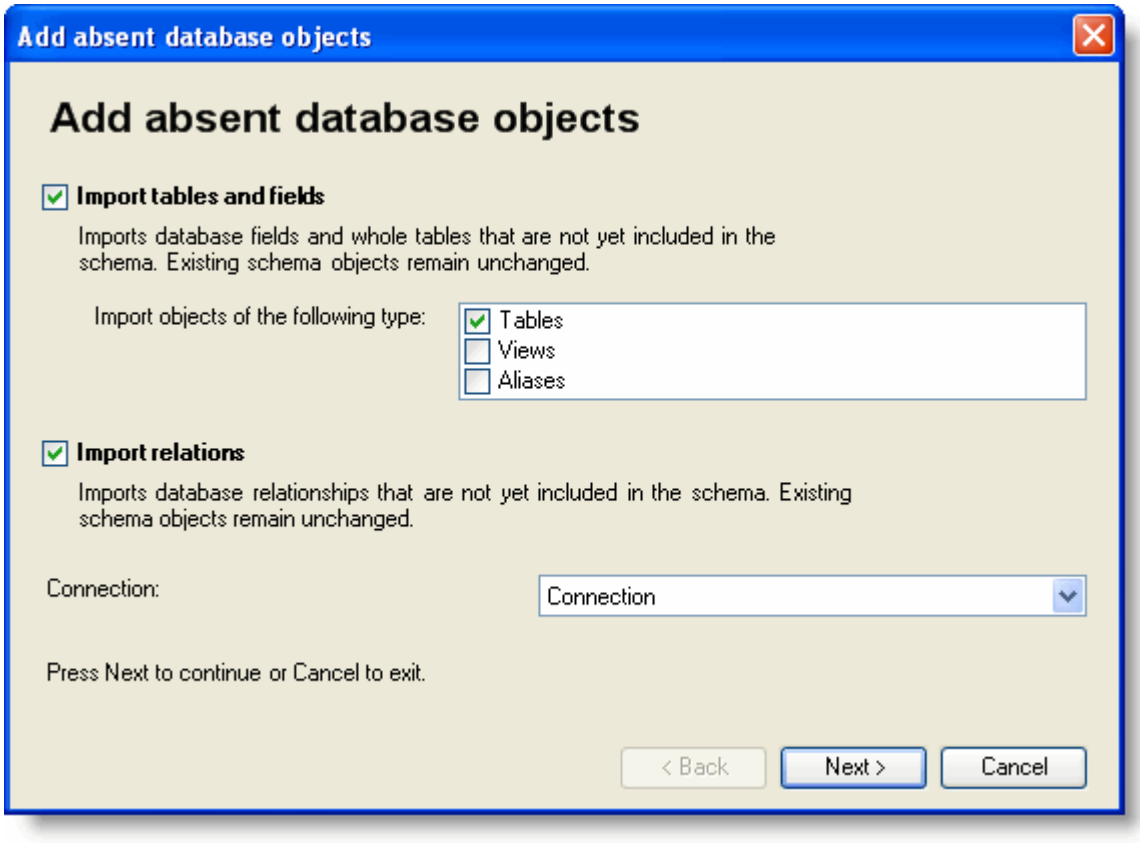
6. Click **OK** to close the **Options** window and click **OK** again in the **Compare schema with database structure** window to begin the comparison. The results appear in the **Output** window.



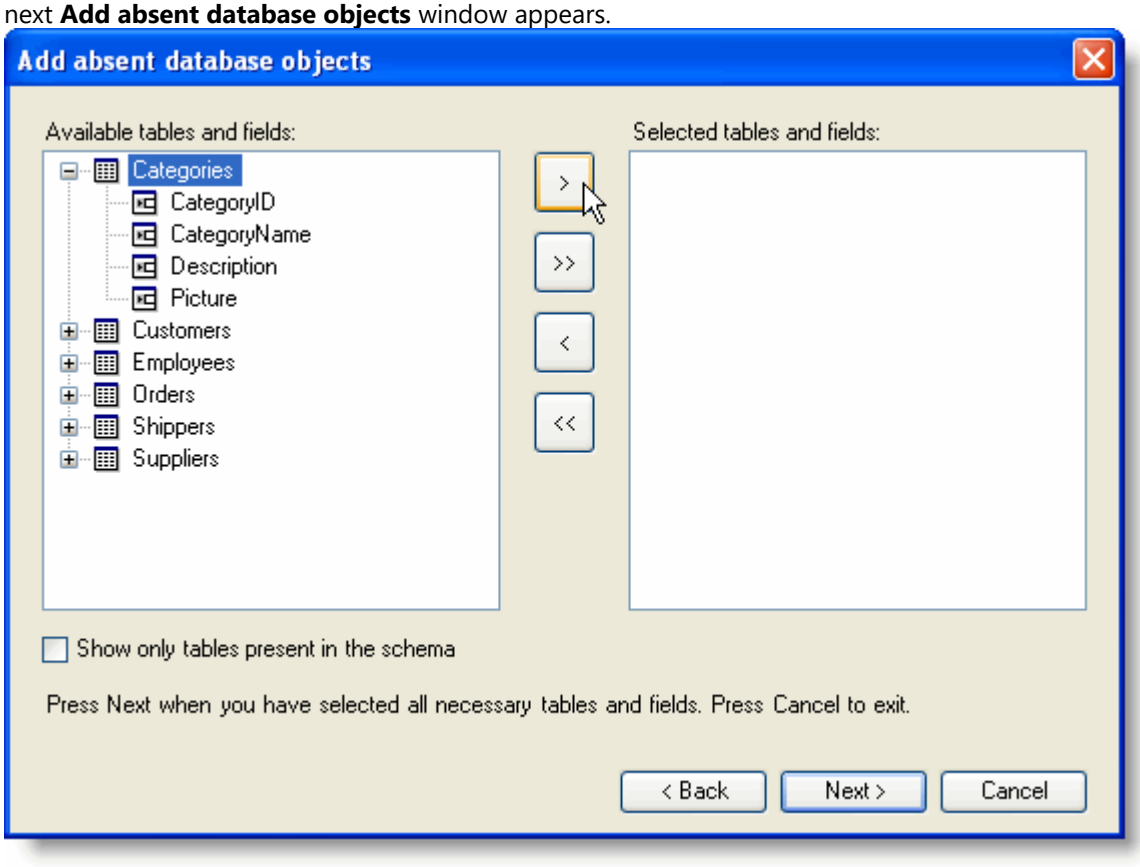
Contents of the **Output** window can be copied to the clipboard using the window's context menu. The tool shows only differences for objects and properties that can be imported from the database structure. In effect, the tool compares two schemas: one is the existing schema, and the other is the schema that would result from the import of the database structure if such import would be performed from scratch at the time of comparison. Only tables, fields and relations belonging to the selected connection are processed; therefore, unbound (calculated) fields are ignored, as are unbound tables, or tables and relations bound to other connections.

The additional import tool, **Add absent database objects** in the **Schema** menu, allows you to import additional fields or whole tables and additional relations that could be added to the database after the schema was created from the database structure. This tool does not change any properties of the objects that are already in the schema, so you can safely use it to add new tables, fields and relations as they are added to the database. To use the **Add absent database objects** tool, complete the following:

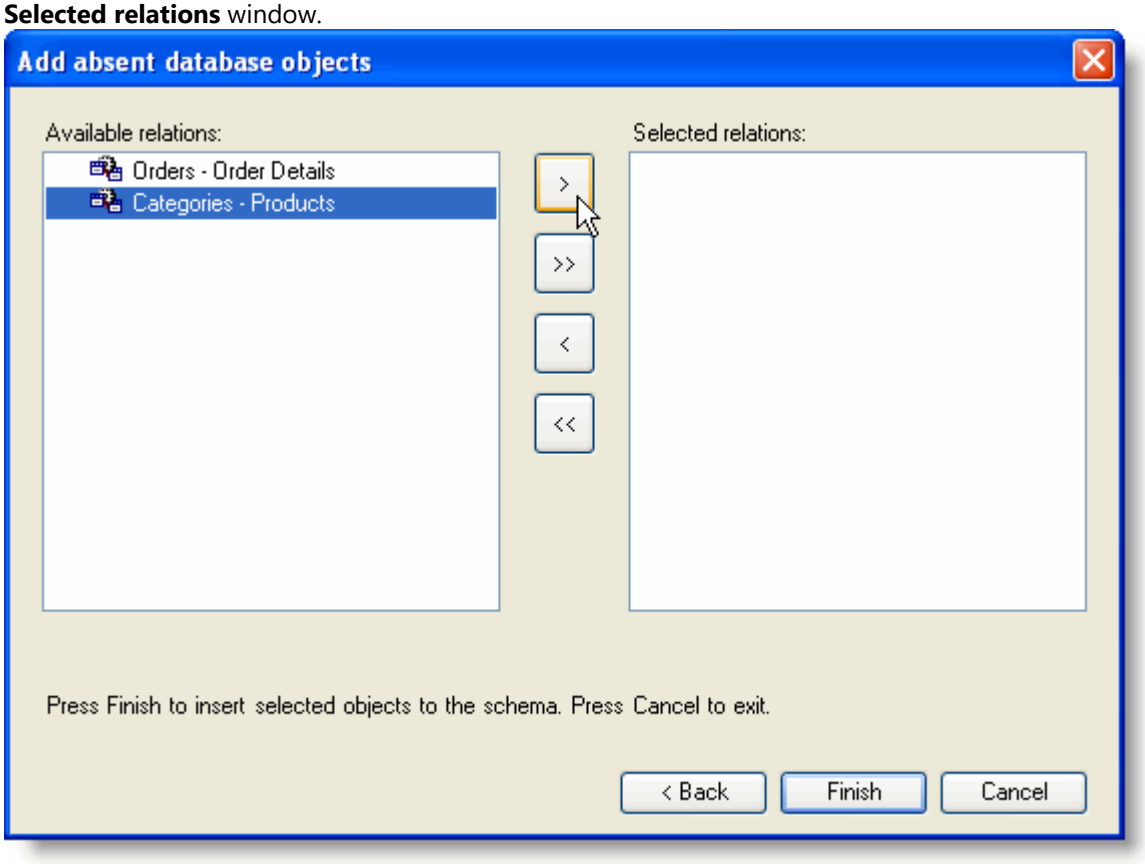
1. Select Add absent database objects from the Schema menu. The Add absent database objects window appears.



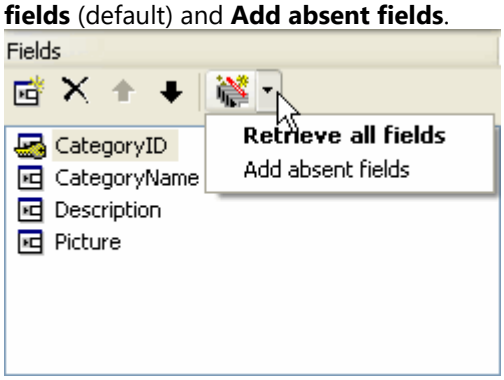
2. Select the checkbox next to **Import tables and fields** to import the ones not already in the schema. You can also check which types of objects to import from the database: tables, views, aliases.
3. Select the checkbox next to **Import relations** to include relations not already in the schema.
4. Select the database connection from the drop-down box next to **Connection** and click **Next** to continue. The next **Add absent database objects** window appears.



5. Select the desired objects from the **Available tables and fields** window and use the arrows to move them to the **Selected tables and fields** window. You can click the node next to the tables to view the list of fields in each table.
   Note: Clicking the **Show only tables present in the schema** checkbox displays only the tables already in the schema in the **Available tables and fields window**.
6. Click **Next** and use the arrows to move the desired relations from the **Available Relations** window to the **Selected relations** window.



7. Click **Finish**. The Schema Designer appears with the specified tables, views, and aliases.
   Finally, there is a **Retrieve Fields** button in the simple table editor in the toolbar at the top of the **Fields** panel. It performs two functions that can be selected in a pop-up menu at the right edge of the button: **Retrieve all fields** (default) and **Add absent fields**.

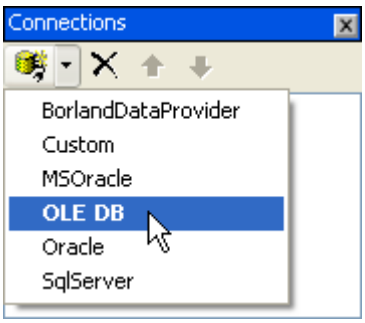


The **Retrieve all fields** action deletes all existing fields and retrieves fields from database structure. There is a confirmation dialog that warns you that this action will delete all existing fields with all their settings. The second action, **Add absent fields**, does not change the existing fields or their properties in any way. It adds the fields that exist in the database table but do not exist in the schema table. Usually those are fields that were added to the table after the schema was created.

Database Connections

A Connection object defines a connection to a database. **DataObjects for .NET** supports database access through OLE DB, native database access for SQL Server and Oracle, and also any other (custom) .NET data providers, see Native and OLE DB Database Access.

Although, in most cases, a schema contains a single database connection, it is possible to have multiple connections in a schema. To create a connection, click the **Add** button and choose a connection type or select **Add** from the context menu in the **Connections** window of the Schema Designer.



You can also add connections using **Import database structure** or **Connect to database** in the **Schema** menu (see Importing Database Structure for details).

Connections are stored in the Connections property of the Schema object. The list of available connections is shown in the **Connections** window of the Schema Designer.

Each simple table has a Connection property associating it with a certain database connection. Together with the DbTableName property, it determines the database table on which this table is based (if any, see Simple Tables for details).

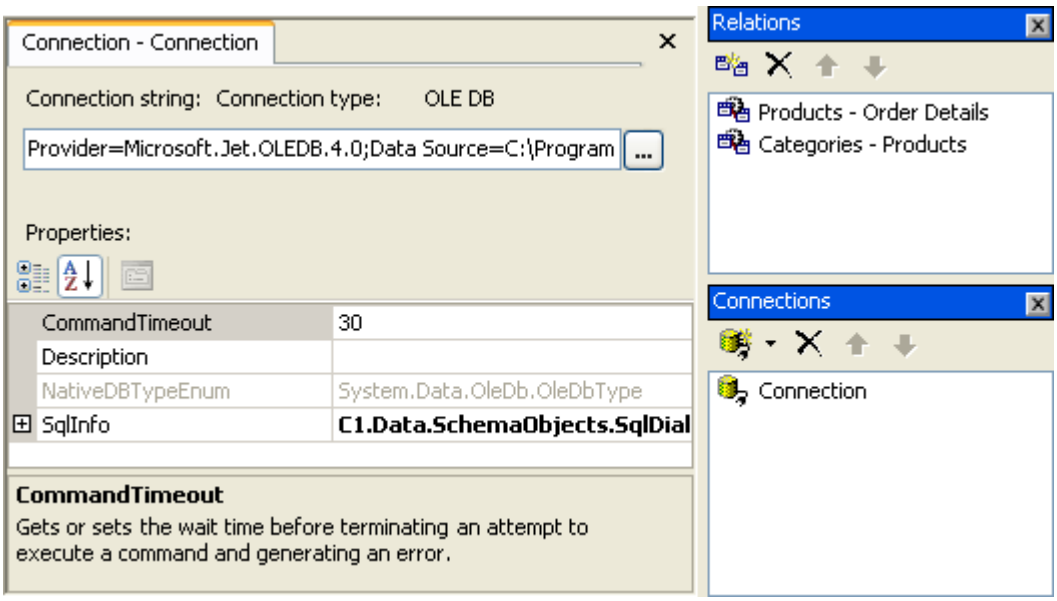
DataObjects for .NET uses Connection objects to perform database access operations, fetch and update on the server, freeing developers from the tedious task of writing database access code. Database access is performed automatically by **DataObjects for .NET**, and it can be customized with user code in appropriate business logic events, see How the Data is Fetched and Updating the Database. A Connection object also serves as transaction context. Performing database updates, **DataObjects for .NET** opens a separate transaction for each connection involved in the operation. In the case of multiple connections, these transactions are managed together in a consistent way, They are either committed or rolled back together, so the whole update operation is performed consistently, as a single, possibly heterogeneous (spanning multiple databases) transaction.

Having multiple connections in a schema allows you to easily switch between different physical databases storing the same logical data, without changing anything else. This is possible both at design and at run time. All that is needed is to change the Connection property in DbTable objects. At run time, it can be done either in the CreateSchema event or using the DynamicConnections run-time property.

Sometimes Connection objects must be changed at run time, to set a specific database location, user ID, password, connection timeout, or other attributes of the ConnectionString. The ConnectionString contains all necessary information to make a database connection. Changing it at run time allows you to adjust that information dynamically. Changing ConnectionString and other connection properties can be done at run time using one of the two methods: in the CreateSchema event or using the DynamicConnections property. In the CreateSchema event you can change any part of the schema, including connections, but that is done only once for the whole application when the data library is initialized. If you need to change ConnectionString or other connection properties several times or on a per-data set basis, use the DynamicConnections property of C1DataSet.

You may also need to change the ConnectionString at run time for security reasons, because you do not want to expose User ID and Password information to all clients of the data library. A frequently used technique is to clear the ConnectionString in the **Schema Designer** before you deploy the data library and use the DynamicConnections property to specify the ConnectionString at run time.

To edit a connection, double-click the connection object in the **Connections** window to open the **Connection Editor**.



Here you can edit the ConnectionString, either manually or in the standard **Data Link Properties** dialog box. Also in the **Connection Editor**, you can specify SQL syntax rules, properties of the SqlDialectInfo object. These properties determine various specifics of SQL syntax relevant to **DataObjects for .NET** that can vary between different databases. Normally, these properties are set automatically to their appropriate values when you specify ConnectionType or ConnectionString, but, occasionally, they may need manual adjustment, for example, when using a third-party OLE DB provider or a custom .NET data provider.

Connections and Transactions at Run Time

In addition to changing the ConnectionString at run time, you can also set the database connection (and/or transaction) to a pre-created IDbConnection (IDbTransaction) object. Fetching data and updating the database (in **Fill** and **Update** methods), **DataObjects for .NET** creates a database connection and transaction for each schema **Connection** object, if its DbConnection has not been set. If you need to use a pre-created database connection, set the DbConnection property. To prevent **DataObjects for .NET** from creating its own connection, it must be set:

- for **Fill**: in or before the BeforeFetch event
- for **Update**: in or before the BeforeUpdate event

If you set the DbConnection to your own IDbConnection object, **DataObjects for .NET** will not close that database connection automatically – you are responsible for closing it. That is usually done in AfterFetch and AfterUpdate events.

By default, **DataObjects for .NET** opens a database transaction for each database connection before update.

If you want to use your own transaction object, set the DbTransaction property in or before the BeforeUpdate event.

If you set DbTransaction to your own IDbTransaction object, **DataObjects for .NET** will not close (commit or roll back) that transaction automatically. You are responsible for committing it in case of success or rolling it back in case of failure, which is usually done in the AfterUpdate event.

Supporting Distributed (COM+) Transactions

Distributed (COM+) transactions can span multiple databases and other transactional resources. They are supported in .NET declaratively, using classes marked with a special [Transaction] attribute.

If you need to support distributed transactions, you will have to override the default **DataObjects for .NET** behavior: opening and committing an ADO.NET database transaction to update the database.

Using ADO.NET (so called "manual") transactions is incompatible with distributed transaction support.

To disable ADO.NET transactions, override the virtual Update method in the RemoteDataService-derived class of your data library. That method has a beginTransaction parameter set to **True** by default. Set BeginTransaction to False and call a method in an object of a class with appropriate TransactionAttribute. In that method, perform the Update, for example, calling the RemoteDataService-derived object's Base.Update() or calling a business method of that object.

Native and OLE DB Database Access

C1DataObjects supports several options for database connectivity, for example:

- You can use OLE DB to connect to any database having an OLE DB provider, or you can use native .NET data providers for SQL Server or Oracle, or you can use any other (custom) .NET data provider.
- For native SQL Server access, System.Data.SqlClient classes are used. For native Oracle access, there are two options: Oracle and MSOracle. The Oracle option uses the Oracle Data Provider for .NET (namespace Oracle.DataAccess; you must install Oracle.DataAccess.dll). The MSOracle option uses Microsoft .NET Framework Data Provider for Oracle available as an MSDN download (namespace System.Data.OracleClient; you must install System.Data.OracleClient.dll).
- For other (custom) .NET data providers, you need to install the data provider and specify necessary properties in the CustomProviderInfo property.

The options in the ConnectionTypeEnum enumeration are: OleDb, SqlServer, Oracle, MSOracle, and Custom. The corresponding classes, all derived from C1.SchemaObjects.Connection are: C1OleDbConnection, C1SqlServerConnection, C1OracleConnection, C1MSOracleConnection, and C1CustomConnection. Creating a Connection object in the **Schema Designer**, you can select one of the five ConnectionTypeEnum options in a combo box. In **DataObjects for .NET Express**, the database access type is specified in the ConnectionType or ConnectionType properties. The latter is used with stand-alone C1ExpressTable components that are not attached to a C1ExpressConnection component.

At run time, you can retrieve the native IDbConnection and IDbTransaction objects from the DbConnection and DbTransaction properties of a C1.SchemaObjects.Connection object. DbConnection contains a non-null object only while the database connection is open. DbTransaction contains a non-null object only while the database connection is open and is performing a transaction while updating the database. You can cast the values of DbConnection, DbTransaction and event arguments representing IDbCommand to the actual type of the native database access objects. For example, for a SQL Sever connection, you can cast DbConnection to SqlConnection, or RowUpdateEventArgs.**UpdateCommand** to SqlCommand.

A Field object in the schema has a NativeDbType property. This property is used for database update, to specify the type of command parameters containing field values that are written to the database. Sometimes it is essential to specify the type for database update more exactly than it can be inferred from the .NET type of the field (from **Field.DataType**). For example, DataType = String can be further specialized to be a Unicode or ASCII string. When you create a schema importing it from database structure, NativeDbType is automatically set to the actual type of the database field. In those rare cases when you need to change it, you can set the NativeDbType property to a specific native type you need, or you can use NativeDbType = Any (-1) to indicate that inferring parameter type from **Field.DataType** is good enough (or not needed at all, as, for example, in calculated fields). The type of the NativeDbType is Integer. It contains the numeric value corresponding to one of the values of the enumerated type describing all possible data types for the database access provider. These enumerations are:

- for **OleDb**: System.Data.OleDb.OleDbType
- for **SqlServer**: System.Data.SqlDbType
- for **Oracle**: Oracle.DataAccess.Client.OracleDbType
- for **MSOracle**: System.Data.OracleClient.OracleType
- for other, custom .NET data providers: the enumeration type must be specified in the NativeTypeTypeName

property

Note that using native database access does not automatically mean your application will have better performance comparing to using OLE DB. Native providers are better optimized, avoiding unnecessary overhead imposed by OLE DB middleware. However, it depends on the relative weight this overhead has in the overall performance of your application. For a very simple application that just uses IDbReader to read database values and does not fill any data set, the performance boost can be very noticeable. But for filling a C1DataSet, the overhead from OLE DB is nearly negligible, so eliminating it will hardly matter. However, using native database access in **DataObjects for .NET** can be important. When you write code, you may need a native provider to improve performance or to perform operations that are not supported by the OLE DB provider for .NET (such as ref cursors in Oracle, and so on). When you do it in your code in conjunction with **DataObjects for .NET**, you will need **DataObjects for .NET** to give you native database connections, not an OLE DB connection.

Using Other (Custom) .NET Data Providers

In addition to the built-in providers (OleDb, SqlServer, Oracle, MSOracle) you can use any other .NET data provider by selecting the **Custom** option in the connection type combo box. For example, the **CustomDataProvider** sample in the **ComponentOne Samples** directory uses Microsoft ODBC .NET Data Provider.

Setting ConnectionType = Custom (selecting **Custom** when you create a new Connection), you indicate that you will use a .NET data provider **DataObjects for .NET** does not know about. There are two differences in using a custom (generic) data provider comparing with using one of the four providers included in the ConnectionType list:

- You cannot create a whole schema by importing database structure. But you can manually define tables one-by-one and retrieve their field from the database structure, and, of course, you can manually create relations and data sets.
- You need to specify necessary information about the provider, such as its assembly name, several type names, and so on, in CustomProviderInfo, telling **DataObjects for .NET** how to use the provider.

Note: For an example of using a custom .NET data provider, see the **CustomDataProvider** sample, which is available installed with the **Studio for WinForms** samples.

Simple Tables

Simple tables are the basic, elementary data objects in a schema. We distinguish between simple and composite tables, although both are Table objects and for the most part can be used interchangeably. Composite tables combine multiple simple tables in a single Table object. A composite table row contains fields from different tables, see Composite Tables for details.

The Table class is abstract, actual Table objects are either simple tables (DbTable) or composite (CompositeTable).

A simple table usually corresponds to a database table (or view), that is, if it has a non-empty DbTableName. In this case, we say that the simple table is based on the database table. This is enough to satisfy most application needs, because in addition to this, the composite table mechanism provides the ability to combine simple tables to form complex objects. However, there is also an option to use SQL-based and unbound tables that are not based on a single database table. A SQL-based table is based on a SQL statement or stored procedure, either by setting the SelectCommandText property or by setting the SQL statement in code. An unbound table is filled with data entirely by code. See Bound, SQL-Based and Unbound Tables for details.

At run time, each simple table has data in a collection of rows. However, this collection of rows is not usually exposed to the user directly, but through another schema object, a TableView. One simple table can participate in many table views, presenting the table's data to the user in different structure arrangements. This is one of the most powerful features of **DataObjects for .NET**: structured data consistently maintained without manual coding. This also means that a simple table does not necessarily include all rows of the underlying database table. The process of filling simple tables with rows (fetching rows) is initiated by TableView objects, each table view making its contribution. See the

details in Structured Data Storage: Tables and Table Views.

Each table, both simple and composite, has the **Fields** collection. Table fields define the structure of a table row. Field properties affect the way the field data is stored and manipulated, and how modifications are sent back to the database. A table's **Fields** collection is not necessarily in one-to-one correspondence with the database fields, the fields of the database table on which the table is based. The fields can have different order, some fields can be deleted, and some fields can be added that are not based on a database field (unbound fields). See Table Fields for more details.

Creating and Modifying Tables

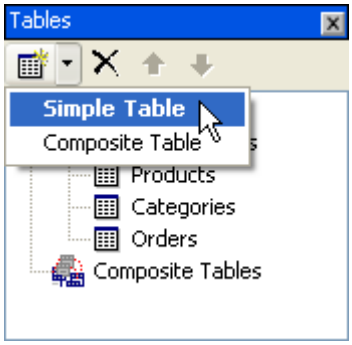
Simple tables are usually created during database structure import (see Importing Database Structure). Any new tables can be added to a schema later. If necessary, there can be more than one table based on a single database table, although that is not the usual practice. Also, you can create tables that are not based on a database table, SQL-based and unbound tables. See Bound, SQL-Based and Unbound Tables for more information.

To add a new simple table to an existing schema:

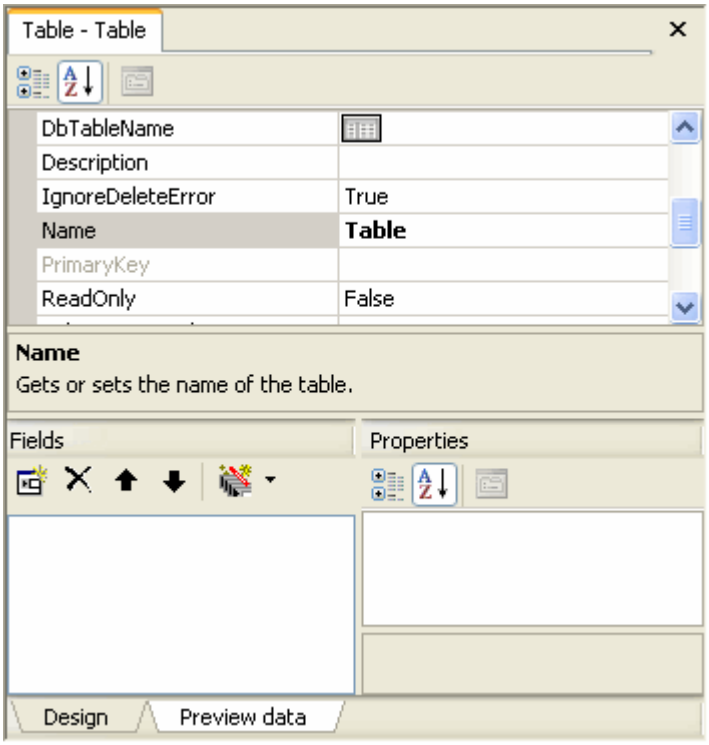
- Using a drag-and–drop operation, place a table from the **Database Tables** window on the **Tables** window creating a bound table based on the database table.

OR

- Press the **Add** button or click the drop-down arrow next to the **Add** button and select **Simple Table** from the context menu in the **Tables** window.



When a new table is added, the **Table Editor** appears:



The **Table Editor** has two panes: the upper pane shows table properties while the lower pane shows the properties of table fields.

Set the Connection and DbTableName properties of the newly created table in the **Table Editor**. For a SQL-based table, instead of setting DbTableName property, set the DataMode property to **SqlBased** and set the SelectCommandText property.

When you set the DbTableName or SelectCommandText property, the **Schema Designer** retrieves table fields from the database. You can also retrieve fields later, using the **Retrieve Fields** button on the **Fields** panel of the **Table Editor**.

To modify properties and fields of an existing table, double-click the table node in the **Tables** window or right-click the table node and select **Open** from the context menu. The **Table Editor** appears.

Using Table Properties

The Connection and DbTableName properties (or SelectCommandText for a SQL-based table) define the database identity of a table. If both properties are set to a value, the table is considered bound to a database table and all database operations are performed automatically by **DataObjects for .NET**.

SQL-based tables can be managed automatically by **DataObjects for .NET** if you create DataAdapter components for them. There are also advanced options for SQL-based and unbound tables, where database access is performed with custom SQL statements or entirely from code; for more information see Bound, SQL-Based and Unbound Tables.

The PrimaryKey read-only property returns the table primary key fields whose PrimaryKey property is set to **True**. Primary key values must be unique in table rows; an attempt to set a duplicate primary key generates an exception. Normally, every table must have a primary key, that is, at least one field with PrimaryKey property set to **True**. Although tables without primary key are allowed (such tables usually have an empty DbTableName (see Bound, SQL-Based and Unbound Tables), they have to be read-only, and they cannot be used in composite tables.

The ReadOnly, AllowAddNew and AllowDelete properties control table updatability.

ConstraintsFieldLevel and ConstraintsRecordLevel properties are collections of table constraints, ConstraintInfo objects. Constraints are expressions (see DataObjects for .NET Expressions for more information). ConstraintsRecordLevel contains record level constraints that are evaluated when the user finishes editing a row,

before the C1TableLogic BeforeEndEdit event. If one of the constraints is not satisfied, an exception is thrown. The exception message is determined by **ErrorDescription.Constraints** with the Condition expression (if non-empty) evaluating to **False**, are skipped and not tested. ConstraintsFieldLevel contains field level constraints, those that are evaluated on every field change. Usually, a field level constraint belongs to a certain field, not to the table as a whole, in which case it resides in Constraints. ConstraintsFieldLevel should contain only field level constraints that cannot be associated with a particular field. Constraints in ConstraintsFieldLevel are evaluated each time any table field changes, whereas constraints in Constraints are evaluated on the owner field change.

The IgnoreDeleteError, UpdateLocateMode, and UpdateRefreshMode properties control the process of updating table rows, committing changes to the database (see Updating the Database for more information).

Table Business Logic Events

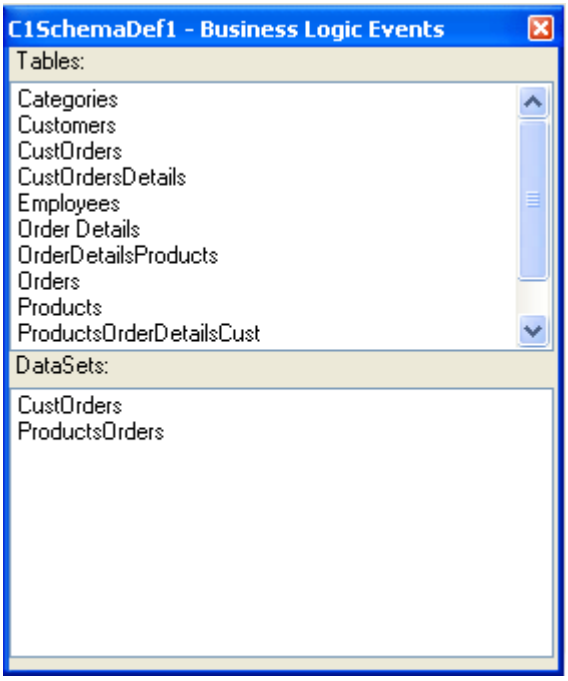
DataObjects for .NET allows developers to specify business logic in code on table level. This is the main power feature of the **DataObjects for .NET** framework: specifying business logic where it belongs, on a data object (table) level and its automatic propagation and enforcement wherever that object is used.

DataObjects for .NET executes table-level business logic code wherever the table is involved: in a composite table including this table and in a table view based on this table.

You can also specify business logic on data set level. Such code will be executed only in a particular data set. Using dataset-level business logic allows the user to enforce rules that are specific to a certain data set. For more information, see Data Sets.

Developers write table-level business logic code in the events of a special C1TableLogic component. Only one C1TableLogic component is allowed for each table. To create C1TableLogic components, right-click the C1SchemaDef component and select **CreateBusiness Logic Components** from its context menu. The C1TableLogic components are automatically added to your form. You can also add them manually from the Toolbox by double-clicking the C1TableLogic component and setting their SchemaComponent and Table properties.

Once you have added the C1TableLogic components, you can attach event code by either selecting the components on the designer surface, or by using the **Business Logic Events** tool window. Right-click the C1SchemaDef component and select **Business Logic Events** from the context menu to open the tool window:



The **Business Logic Events** tool window shows the list of all tables and data sets. When you select a table in the tool

window, the table's business logic events appear in the Properties window (in Visual C#, when the Events radio button is selected in the Properties window; in Visual Basic use the Method Name combo box in the code editor).

DataObjects for .NET also allows you to associate business logic code with table views instead of tables. This is done to specify data set-specific (table view-specific) logic, rules that must be enforced in the context of a particular data set and table view, but do not always apply to the underlying table. In this case, a C1DataSetLogic component is used, for more information see Table View Business Logic Events.

The following is a brief list of business logic events available in **DataObjects for .NET** (note that the Before and After prefixes pertaining to most events have been omitted and the prefix is only included if an event occurs only Before or only After):

Event	Description
AddNew	Fired when a new (empty) row is added.
AfterChanges	Fired when all changes initiated by a field change are done and handled by the business logic code, see the **FieldChange** event.
BeginEdit	Fired when the user starts editing a row (data-bound controls start editing a row immediately after they position on it, even though no changes have been made yet).
CancelEdit	Fired when the user cancels editing a row reverting the changes made to it.
Delete	Fired when a row is deleted.
EndAddNew	Fired when a newly added row becomes a regular row in the rowset. When a row is added, it is added empty, its primary key is unknown. A row with unknown primary key is in special transitory state, it is not a regular rowset row. Only after its primary key is set it becomes a regular (added) row, which is signaled by this event.
EndEdit	Fired when the user finishes editing a row (data-bound controls finish editing a row when they leave that row, even if no changes have been made).
FieldChange	Fired when a field value is set. Inside this event, your code can set other fields triggering recursive **FieldChange** events, **DataObjects for .NET** handles this situation correctly. Only after all changes are done and handled, the AfterChanges event is triggered.
FirstChange	Fired when a first change is made to the row (a field value changed) after **BeginEdit**.
UpdateRow	This event is not fired in a client application, unless it is a direct client, that is a 2-tier application updating the database directly from the client, see Application Configurations. In a 3-tier deployment, it is fired only on the server, when a modified row is committed to the database. See also, Updating the Database.

Table Fields

Table fields determine the structure of a table row. In a bound table (see Bound, SQL-Based and Unbound Tables), fields usually correspond to database fields of the database table. However, they do not have to be in exact one-to-one correspondence with database fields. Fields order can be changed, some fields can be deleted from the table, and new fields can be added. If necessary, multiple fields can represent a single database field. The database field represented by a table field is determined by the DbFieldName property. This property can be empty, which means that the field is not based on (not bound to) a database field. Such fields are called *unbound*, or *calculated*.

A field has a name that must be unique in the table. A field can be renamed by renaming the field node in the Fields list. The name is used to identify the field as a property of the business object associated with the table, see Using Typed Data Objects in Business Logic. It is also used as the name of the column exposed to the users, and as the default display name (caption) of the column.

The DataType property determines the field .NET type, and NativeDbType – its native database or OLE DB type. Both DataType and NativeDbType are set by Import Wizard when a table is first created in database structure import, see Importing Database Structure. DataType is the field's most important characteristic, it affects all **DataObjects for .NET** functions. Normally, **Field.DataType** should not be changed after importing structure from the database. Adding a new field to a table, you must set its DataType property. NativeDbType is used only for updating values in the database, and can be set to **Any (-1)**, in which case its value is effectively ignored. When in doubt, use **Any (-1)** as the default value for NativeDbType.

The following properties determine the field's additional storage characteristics:

Property	Description
AllowDbNull	Gets or sets a value indicating whether null or empty string values are allowed in this field. If it is set to **False**, an attempt to assign null or empty string value to this field generates an exception.
AutoIncrement	Gets or sets a value indicating whether the field automatically receives an incremented value for a new row added to the table.
AutoIncrementSeed	Gets or sets the starting value for a field with AutoIncrement not **None**.
AutoIncrementStep	Gets or sets the increment a field with AutoIncrement not *None*.
MaxLength	Gets or sets the maximum length of a string field, in characters. If the length is unlimited, the value is **0** (default).
Precision	For numeric fields (NativeDbType is **Numeric**, **Decimal**, or **DbTimeStamp**), this property sets or gets the maximum number of digits representing values.
Scale	For numeric fields (NativeDbType is **Numeric**, **Decimal**, or **DbTimeStamp**), this property sets or gets the scale of numeric values, that is, how many digits to the right of the decimal point are used to represent values.
Unique	Gets or sets a value indicating whether the values of this field in each row must be unique. If it is set to **True**, an attempt to assign a duplicate value to this field generates an exception.

The PrimaryKey property determines whether the field belongs to the table's primary key. The table's primary key, the sequence of field names constituting the primary key, is returned by the PrimaryKey read-only property. Each modifiable table must have primary key, that is, at least one field where the PrimaryKey property is set to **True**. Primary key values must be unique in table rows; an attempt to set a duplicate primary key generates an exception.

Each field can have one or more calculation expressions associated with it in the Calculations property, a collection of FieldCalculationInfo objects. Usually, calculation expressions are used in *calculated* (unbound) fields, although they can be useful in bound fields as well. A field is called *unbound* if its DbFieldName property is empty. **DataObjects for .NET** automatically computes calculated field values and automatically refreshes them when any of the arguments of its calculation expression(s) changes. Calculation expressions can contain fields of the same table as well as of its parent and child tables with respect to relations. Child fields are used with aggregation functions to perform grouping and aggregation. See DataObjects for .NET Expressions for a description of **DataObjects for .NET** expression language.

Each calculation (FieldCalculationInfo) contains Expression, which is the expression used to obtain field values, and two additional properties: Condition and FireEvent. Condition is an optional Boolean expression determining the calculation's applicability. If Condition evaluates to **False**, the calculation expression is not evaluated. If a field has multiple calculations, **DataObjects for .NET** applies the first with the Condition expression evaluating to **True**. If none is applicable, the field value is left unchanged. FireEvent is a Boolean property set to **False** by default. If it is set to **True**, setting the value from calculation expression triggers the same sequence of events (BeforeFieldChange, AfterFieldChange, AfterChanges) as if the value has been modified by the end user.

Field calculations are useful for bound fields as well as for calculated fields. In this case, they are usually qualified by Condition expressions. Depending on the conditions, a field's value can be derived from a value stored in the database, or it can be calculated. For instance, if field A is non-empty, field B always returns the same value as A, but if A is empty, B can be set independently of A.

The following properties control field value modifications:

Property	Description
ReadOnly	Gets or sets a value indicating whether the field value can be changed by the end user or from event code. If set to **True**, an attempt to change the field throws an exception.
ReadOnlyUnlessNew	Gets or sets a value indicating whether the field value can be changed after the row has been added to the table (after **EndAddNew** event). If set to **True**, an attempt to change the field throws an exception unless it is done in a newly added row, before the **EndAddNew** event.
Constraints	Returns the collection of field-level constraints, ConstraintInfo objects. Field-level constraints are evaluated (tested) when the value of the field changes. For a change to be successful, all constraint expressions (Expression) must evaluate to **True**. If one of the constraints is not satisfied, an exception is thrown. The exception message is determined by ErrorDescription. Constraints with Condition expression (if non-empty) evaluating to **False**, are skipped, not tested. See also, DataObjects for .NET Expressions for a description of **DataObjects for .NET** expression language.
DefaultValue	Gets or sets the default value, in string representation, for the field in a newly created row.

The following properties control field behavior in updating the database (they only have effect for bound fields, that is, if DbFieldName is not empty):

Property	Description
DataSourceReadOnly	Gets or sets the value indicating whether the field value in the database can be changed. If this property is set to **True**, the field value will not be set in the database update operation (as with UpdateSet = *Never*) and it cannot be modified unless it is done in a newly added row, before the **EndAddNew** event (as with ReadOnlyUnlessNew = **True**).
UpdateIgnore	Gets or sets the value indicating whether the field value is sent to the database (to the server) for update. If this property is set to **True**, the field value is not sent to the server for update, even if it has been modified by the user. The default is **False**.
UpdateLocate	Gets or sets the value indicating whether the field value is used to locate the database record that is going to be updated. If it is set to **False**, the field is not used for locating the database record, regardless of the value of UpdateLocateMode. If it is set to **True** (default), this is determined by the value of the UpdateLocateMode property. See Updating the Database for details on locating database record for update.
UpdateRefresh	Gets or sets the value indicating whether the field value is refreshed, retrieved from the database after updating the database record. If it is set to **False**, the field value is not refreshed, regardless of the value of UpdateRefreshMode. If it is set to

	True (default), this is determined by the value of the UpdateRefreshMode property.
UpdateSet	Gets or sets the value indicating whether the field value is set in the database record. If it is set to **Always**, the value in the database is always modified, set to the current field value. If it is set to **Never**, the value in the database is always left unchanged. If it is set to **IfChanged**(default), the value in the database is set to the current field value if the current field value is different from the original field value as it was last fetched from the database (the original value can be retrieved using Item (field, DataRowVersionEnum.Original)).

DataObjects for .NET can perform automatic conversion from DbNull to empty string when returning string field values and back from empty string to DbNull when saving string field values. To enable this automatic conversion, set ConvertNullToEmpty/ConvertEmptyToNull for individual fields, or set FieldDefaults:DataSetDef in a data set in the schema (in **DataObjects for .NET Express**, FieldDefaults) to enable it by default for a whole data set.

Structured Data Storage: Tables and Table Views

One of the main advantages of **DataObjects for .NET** over other data frameworks is its ability to represent structured data. For example, suppose we have tables *Customers*, *Orders* and *Employees* (as in the standard Northwind MS Access sample database), and we have defined a composite table *CustOrders* (see Tutorial 1: Creating a Data Schema) that combines all three tables in a single table according to the following diagram:

(The *Orders – Employees* many-to-one relation is defined by the join condition Orders.EmployeeID = Employees.EmployeeID, where EmployeeID is the primary key of the Employees table.)

Suppose the user navigates to a *CustOrders* row (containing fields from all three tables), and changes the value of Orders.EmployeeID. Then **DataObjects for .NET**, knowing the structure behind the composite table, changes the employee name (fields Employees.FirstName and Employees.LastName) and other employee attributes according to the changed EmployeeID value.

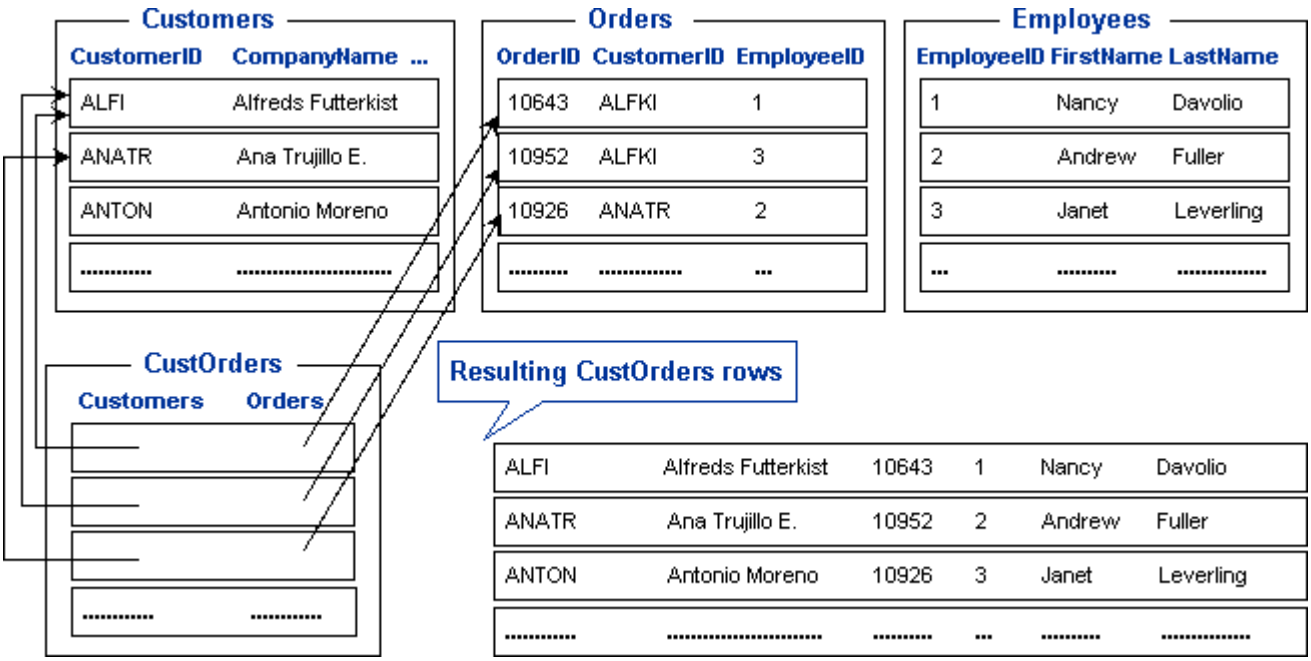
Or the user can change the value of Employees.FirstName or another Employees field in the simple *Employees* table. In this case we expect that all CustOrders rows including this Employees rows will show the modified value in the corresponding field.

Or let's take another example: two table views (TableView) showing the data of a single table (in different order, with different filter conditions, different fields, and so on). Here also, once a table row is modified in one table view, the corresponding row will be automatically changed in another.

In other words, **DataObjects for .NET** always maintains and enforces the structure, relations between tables, composite tables and table views, exactly as you need it to be maintained. What mechanism is responsible for this, how **DataObjects for .NET** does this? The answer to this question lies in the way **DataObjects for .NET** stores data. **DataObjects for .NET** storage is organized according to the structure and role of schema objects, enabling **DataObjects for .NET** to maintain the correct structure at all times.

The data is stored on the simple table level (except unbound fields). Each table has its own cache, rowset. Table views do not store actual data, instead they store pointers to table rows. In the same way, composite table rows do not store actual data, they also store pointers to simple table rows. This storage structure serves a dual purpose: to eliminate data redundancy and thus conserve memory, and to ensure that the structure is preserved when the user modifies data. Composite table rows and table view rows pointing to the same simple table row share the same data, since the actual data is stored in one place, in a simple table row. When the shared data is modified through one of the composite table rows or through one of the table view rows, all other rows pointing to the same simple table row are notified of the change and display the modified data.

DataObjects for .NET storage is illustrated in the following figure showing how it stores the simple table rows of tables *Customers*, *Orders*, *Details*, and the rows of the *CustOrders* composite table:



In this figure, you can see that pointers to simple table rows exist only for two tables: *Customers* and *Orders*, there are no pointers to *Employee* rows. This is because the Employees table is connected to its parent in the CustOrders diagram, the *Orders* table, with a many-to-one relation. In this case, the Employees row is uniquely defined by the Orders row, therefore we do not need a separate pointer for it.

Now you can see how **DataObjects for .NET** ensures that the correct data structure is maintained at all times regardless of where, in which table view, a modification is made: Any data modification made in a table view ends up in a simple table row. If the table view represents a simple table, this row is found using the single pointer that this table view row contains. For more information, see Table Views. If the table view represents a composite table, the table view uses one of its pointers to simple table rows associated with one-to-many relations, and, if necessary, foreign key values, such as Orders.EmployeeID in the above example, to resolve many-to-one relations. Once the simple table row is found, it is modified. Then **DataObjects for .NET** notifies all table views (both simple and composite), whose rows point to the modified row, that the row has changed. When, in response to this notification, a client requests the values of those table view rows, they return the changed value, because they always return the current value of the simple table row, using their row pointers.

There is one exception to this storage scheme: *unbound* fields (also known as *calculated*) in a table view. These are table view fields that do not correspond to any table field, they were specifically added to the table view by the developer. For more information, see Table View Fields. A table view can have unbound fields, and a composite table can have unbound fields of its own. Unbound composite table fields are those that do not correspond to any field of constituent simple tables, they were added to the composite table by the developer. For more information, see Composite Table Fields. Unbound field values do not belong to simple tables, so they cannot be stored in simple table rows. **DataObjects for .NET** stores unbound field values in table view rows where they are defined.

How the Data is Fetched

Filling Data Sets with data (executing Fill), **DataObjects for .NET** performs fetch for every table view in that data set. Simple tables themselves are not fetched, because all simple table rows may not be needed. **DataObjects for .NET** exposes table views to the user, not simple tables, so it fetches only those rows that occur in the table views. You must take this fact into account when designing the data set structure. Sometimes, it is necessary to ensure that all rows of a certain simple table are present in the dataset. For example, if you want the end user to be able to select an

employee in an Orders row (see the example in Structured Data Storage: Tables and Table Views), you must have all Employees rows available in the data set. To ensure that all simple table rows are fetched, make sure that the data set includes a table view based on that simple table, add this table view to the data set if it is not already present, and do not specify a filter condition for this table view calling the Fill method.

Sometimes it is necessary to control the order in which table views are fetched. Specifically, it can be necessary when custom code is used in fetching and some parameters in this code depend on the fact that some other table view has already been fetched. To specify fetch order, set FetchIndex (usually in the **Schema Designer**) orFetchIndex (only in the BeforeFetch event).

How the Data is Modified

Structured storage also allows **DataObjects for .NET** to solve another very important problem that other data frameworks fall short of solving in full: the problem of updateable views. Virtually all database applications, especially their GUI front ends, encounter this problem in one way or another. In ADO.NET, as in most previous SQL-based frameworks, it is solved by placing the responsibility for updating a multi-table data view (rowset) on the developer. Unless a rowset (ADO.NET DataTable) is based on a single database table (which is rarely the case in GUI applications), developers must write custom code to send modifications to the database. This is almost taken for granted, as if there is no other solution to this problem causing a lot of tedious manual coding. In fact, the solution exists and has been known for a long time to previous generations of application builder tools. Unfortunately, it has been abandoned if not forgotten. **DataObjects for .NET** restores the data structure paradigm to its rightful place and with it comes the solution to view updatability.

As we already know, imposing structure on rowsets (composite tables) originating from multiple database tables (as opposed to structureless DataTable of ADO.NET) ensures data consistency and synchronization that would otherwise require manual coding. The same structure helps to solve the update problem. In fact, using **DataObjects for .NET**, you will rarely write any code related to database updates at all. In most cases, database updates will be performed automatically by **DataObjects for .NET** in the exact way they are required by the data model. When your application sends updates to the database (Update), **DataObjects for .NET** collects all modified simple table rows and sends them to the server for update (commit).

Note that simple table modifications are sufficient for the update process. We do not need to be concerned with where those modifications came from. The user might use a simple table view or a composite table view to modify a row of a simple table, it does not matter. All that matters is that a simple table row has been modified (or inserted, or deleted). Before the update, **DataObjects for .NET** has already resolved the problem that other tools must resolve during Update: using the data structure, it has already determined which rows of which simple tables have been modified by the end user changes. Therefore, all that remains is to send those simple table modifications to the database, applying each of these modifications to the corresponding database table. Only simple table rows are involved in this process. Complex objects, such as composite tables and table views are not involved in update.

Consider the *Customer-Orders-Employees* example of Structured Data Storage: Tables and Table Views, the user could modify some rows of the *CustOrders* composite table (using a grid control showing the combined customer-order-employee information), and also add and delete some rows. Since *CustOrders* is a composite table, it means that some Customers rows could have been modified, as well as some Orders and some Employees rows. Additionally, other modifications to these simple tables can be made in the same session via other table views. All these modifications are basically modified rows of the simple tables. Therefore, **DataObjects for .NET** already knows what database table rows must be modified, added or deleted to commit this transaction. It goes to the server and does just that, applies the modifications to the database tables *Customers*, *Orders* and *Employees*.

For details on the process of committing changes to the database, including possible customization of the update process with your business logic code, see Updating the Database.

Note that in addition to the **batch update mode**, standard in distributed applications, where database updates are performed only when explicitly requested by a special method call (when the user presses a button, for example), **DataObjects for .NET** also supports the classic **automatic update mode**. This mode, familiar to developers of classic client-server and desktop applications, ensures that all changes made in the current row are committed to the database when the end user leaves this row, moves to another row in a table. This mode is not supported in ADO.NET

and standard Windows Forms data binding. It is correct that this mode is unsuitable for distributed applications. However, many existing designs depend on it, and it is very well suited for desktop applications. **DataObjects for .NET** adds yet another enhancement to the .NET data framework by implementing the automatic update mode. By default, all **DataObjects for .NET** data sources work in batch update mode. But if you use **C1DataTableSource** as your data source, and set its **UpdateLeavingRow** property to **True**, it will perform update automatically when the end user leaves a row after modifying it.

How to Access Table Data

Data Binding

As described in Schema Objects, fetched data is contained in a C1DataSet object. By using a C1DataSet component as your data source, you can bind data-aware components such as a grid or a text box to the fetched data. The C1DataSet component exposes table view rowsets. Data-aware controls can bind to any of these rowsets. A table view represents a table, but there can be multiple table views representing the same table. Also, table views can represent composite tables as well as simple tables. C1DataSet also supports master-detail hierarchy in data binding. See How to Access Table View Data for details.

DataObjects for .NET also allows you to bind directly to a simple table rowset, bypassing table views. This can be done at run time by setting the DataSource property of a data bound control to a C1DataTable object representing a table rowset. To obtain this C1DataTable object, use the **Tables** collection property of the C1DataSet component.

This can also be done, both at run time and at design time, by using the C1DataView component. It can represent either a table view, if its TableViewName property is set, or a simple table, if its TableName property is set.

Programmatic Access

To access a table rowset (both for simple and composite tables), obtain a C1DataTable object from a C1DataSet component (using the Tables collection property), and use the **Rows** collection containing C1DataRow objects representing table rows. To perform actions on table rows or table data as a whole, use the appropriate methods and properties of the C1DataTable and C1DataRow classes. Their object model is similar to that of ADO.NET.

DataObjects for .NET also allows typed access to table and table view rows. For each table and table view, **DataObjects for .NET** generates a class representing its row. For example, **ProductsRow** is an object (business object, data object) where each field has a corresponding property (**ProductsRow.UnitPrice**, **ProductsRow.UnitsInStock**, and so on) and each relation has a corresponding method (**Products.GetOrder_DetailsRows**, and so on) allowing you to obtain child rows and the parent row. Using these classes, you can write your business logic code in a convenient, type-safe way, and benefit from Visual Studio code completion features giving you the lists of properties and methods to choose from. See Using Typed Data Objects in Business Logic, for details about these *data object classes*.

To use a data object class in your code, call the static **Obj** method implemented in every data object class. It obtains a business object given a C1DataRow object as its argument. For example, the following code obtains a ProductsRow business object from a C1DataRow:

To write code in Visual Basic

VB

```
Dim dataRow As C1.Data.C1DataRow

Dim product As DataLibrary.DataObjects.DataSet.ProductsRow
product = DataLibrary.DataObjecs.DataSet.ProductsRow.Obj(dataRow)
```

To write code in C#

```
C#
C1.Data.C1DataRow dataRow;

DataLibrary.DataObjects.DataSet.ProductsRow product;
product = DataLibrary.DataObjects.DataSet.ProductsRow.Obj(dataRow)
```

Sometimes it may be necessary to obtain C1DataRow objects from the rows (items) received by bound controls from **DataObjects for .NET**. For example, you may need to access the current row obtained from **CurrencyManager.Current** as a C1DataRow. Row objects (also called data items) used in data binding are not C1DataRow objects, but each of them refers to a single C1DataRow object, and you can use a static FromDataItem method to obtain that C1DataRow object.

Bound, SQL-Based, and Unbound Tables

The most common case is a *bound table* (DataMode = **Bound**) where a table has non-empty Connection and DbTableName properties. In this case, a table is based on a database table, so all database access operations can be performed by **DataObjects for .NET** automatically. There is no need to write custom code fetching and updating the table data (although it is possible to customize the default behavior in code).

In more complicated cases, there may be a need for more flexibility. For example, you may need to use a complex, non-standard SQL statement to fetch table data, or to use a stored procedure. Normally, for a bound table, the SQL statements fetching and updating data are generated by **DataObjects for .NET**. If that is not adequate, you can use a *SQL-based table* (DataMode = **SqlBased**). A SQL-based table has non-empty Connection property and empty DbTableName property. Instead of DbTableName it uses the SelectCommandText property where you can specify a SQL statement or a stored procedure depending on the SelectCommandType property setting.

To define a SQL-based table, set the DataMode property to **SqlBased**, set the Connection property, leave the DbTableName property empty, and specify the SelectCommandType and SelectCommandText properties.

Using a SQL-based table can be almost as easy as using a bound table, if you create a DataAdapter component for it. A DataAdapter component can be created in a C1TableLogic component associated with the table by selecting **Create DataAdapter** from its context menu. The DataAdapter component will then perform both fetch and update without custom code (but you can customize the default fetch and update behavior in event code if needed).

> **Note:** For an example of using a SQL-based table with DataAdapter, see the **SQLBasedTablesEasy** sample which is installed with the **Studio for WinForms** samples.

Although SQL-based tables are almost as easy to use as bound tables, bound tables are always preferable when you have a choice. Bound tables are more intimately related to database tables, so **DataObjects for .NET** can support features that are impossible for SQL-based tables; for example:

- Only bound tables can be combined to form a composite table. SQL-based tables cannot be used in a composite table.
- Only bound tables can be used in virtual mode (with large data sets), SQL-based tables cannot.
- FilterCondition in Fill does not work with SQL-based tables without additional coding, because **DataObjects for .NET** does not generate SQL statements in this case, so it cannot automatically modify the SQL statement by adding the condition to its WHERE clause. You need to do it in code, in the AfterGenerateSql event.

If you want even more control over fetch and update process, you can use a SQL-based table without DataAdapter component and even without setting the SelectCommandText property. In this case you perform fetch and update by generating SQL statement(s) in code:

- To fetch data for an SQL-based table, write code in the BeforeGenerateSql event setting the **Sql** event argument to the SQL statement fetching data and set the **Status** argument of the event to **Skip**. Note that since it resides in a C1DataSetLogic component, this code is attached to table views based on this table, rather

than to the table itself. This is in keeping with the general rule that data is fetched by table views, not by tables; for more information, see How the Data is Fetched.
- If the table is updatable, write code in the C1TableLogic BeforeUpdateRow event. In that code, you can specify update, insert and delete commands, or perform the update in any desirable custom fashion.

> **Note:** For an example of using SQL-based tables without DataAdapter, see the **SQLBasedTables** sample which is installed with the **Studio for WinForms** samples.

You can also combine these options, using these custom SQL statements created in code with a SQL-based table with DataAdapter, or, for that matter, event with a bound table. For example, you can fetch data without custom code, using DbTableName or SelectCommandText properties, and update with custom code.

If you need even more flexibility than provided by SQL-based tables, for example, when working against a non-SQL data source, you can use an *unbound table*. An unbound table is a table with both DbTableName and Connection properties empty (set the DataMode property to *Unbound*). Fetching and updating data for an unbound table is done in code, for example:

- To fetch data for an unbound table, write code in the BeforeFetch event. In that code, create a reader object implementing the System.Data.IDataReader interface (either use a standard .NET Framework implementation such as OleDbReader or SqlDataReader, or, for custom data, create an UnboundDataReader object), fill the reader with data, assign it to the Reader event argument and set the **Status** argument of the event to **Skip**. Note that, as in SQL-based tables, this code is attached to table views based on this table, rather than to the table itself, according to the general rule that data is fetched by table views, not by tables. For more information, see How the Data is Fetched.
- If the table is updatable, write code in the C1TableLogic BeforeUpdateRow event. In that code, perform the necessary update programmatically and set the **Status** event argument to **SkipCurrentRow**, indicating that the update has been done.

> **Note:** For an example of how to use unbound tables, see the **UnboundTables** sample, which is installed with the **Studio for WinForms** samples.

Simple Relations

A simple relation (class SimpleRelation) is an object establishing a parent-child relationship between two simple tables. The concept of relationship is common in database modeling. It is reflected in a database structure, for example, by foreign key relationships where the primary key of one table (parent) corresponds to a foreign key of another table (child).

A simple relation relates the two tables by creating a correspondence between a field (or several fields) of the parent table (parent fields) and a field (or several fields, the same number of fields as in parent) of the child table (child fields). For example, in a *Customers – Orders* relation, the *Customers* table is the child and the *Orders* table is the parent. They are related by the equality

```
Customers.CustomerID = Orders.CustomerID
```

where **Customers.CustomerID** is the parent field and **Orders.CustomerID** is the child field.

For each parent (*Customers*) row there can be multiple child (*Orders*) rows with the same value of the **CustomerID** field. For each child (*Orders*) row there is only one parent (*Customers*) row with the same value of the CustomerID field. This is why this is a **one-to-many** relation (one parent row to many child rows), which is represented by the following notation:

```
Customers →(1-∞) Orders
```

We can also define an *Orders – Customers* relation where *Orders* is the parent and *Customers* is the child. This will be a **many-to-one** relation:

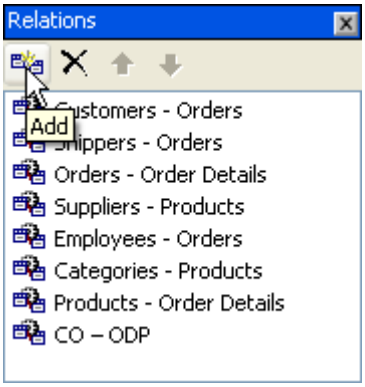
```
Orders →(∞-1) Customers
```

In a many to one relation, there is only one child row for each parent row, and there can be many parent rows for a child row.

Creating and Modifying a Simple Relation

When you import database structure to a schema, **C1DataObjects Import Wizard** automatically creates simple relations for you based on foreign key information in the database. After the initial step of importing the database structure, you can delete redundant relations, create new relations and modify the existing relations.

To create a new simple relation, press the **Add** button or select **Add** from the context menu in the **Relations** window, which will create an empty relation and open it for editing in a new **Relation Editor** page.



The **Relation Editor** has two panes. The top pane contains the relation properties, the bottom pane contains Join Conditions defining the relation (such as Customers.CustomerID = Orders.OrderID).

Set the relation's Name either by setting the Name property or by renaming the newly added node in the **Relations** window.

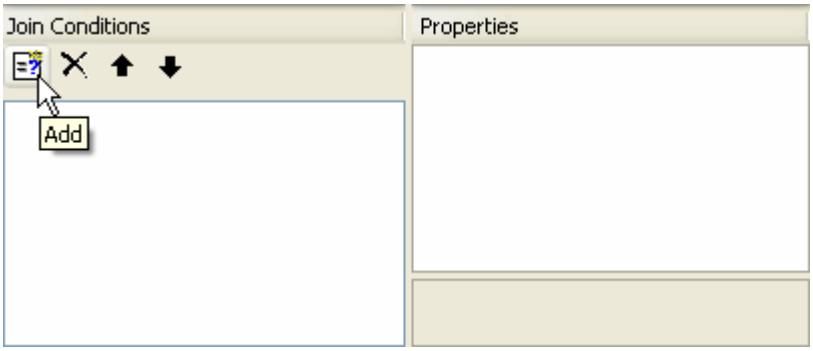
Set the Parent property by selecting a simple table from the list of all simple tables available in the schema. Set the Child property by selecting a simple table from the list of all simple tables. Note that if you select a composite table for either Parent or Child, the relation will automatically become composite relations, which will be apparent by the elimination of the bottom panel, Join Conditions, which are not applicable to composite relations.

When creating a simple relation, it is your responsibility to assign the correct value to the Cardinality property, that determines whether this relation is one-to-one or one-to-many. **DataObjects for .NET** relies on and makes extensive use of relation cardinality, and it cannot derive its value from other relation properties, unless for relations it creates while importing structure from the database.

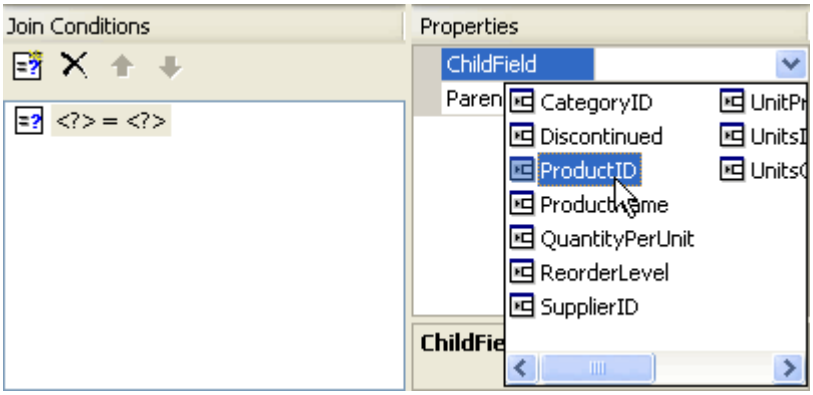
The recommended practice is to define all your simple relations as one-to-many (but make sure they are really one-to-many, **DataObjects for .NET** cannot check this). For example, if you need a relation between *Customers* and *Orders*, you may be uncertain as to which table to choose as parent and which as child. Both choices are possible, only Cardinality value depends on the choice. We recommend to make the choice that leaves Cardinality at one-to-many. So, in our example, *Customers* is the Parent, and *Orders* is the Child.

If you have a *Customers – Orders* relation, there is no need to define the inverse relation, *Orders – Customers*. In your code, you can always navigate in both directions, from parent to child and from child to parent. When you use this relation in a diagram (in Composite Tables or in Data Sets) to specify view relations, you can use it in direct or inverse direction, as needed.

The last step in defining a simple relation is specifying its join conditions, equalities connecting parent fields with child fields. For each parent/child field you need one join condition. To add a join condition, press the **Add** button or select **Add** from the context menu in the **Join Conditions** pane.



Then, in the properties grid on the right, select the ParentField and ChildField for this join condition.



Simple Relation Properties

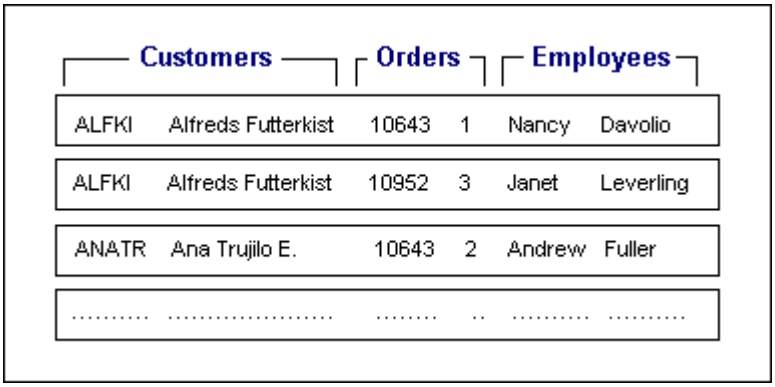
The following properties can be used when creating a simple relation between two simple tables:

Property	Description
EnforceConstraints	If this property is set to **True** (default), then this relation will not allow child records without parent. For example, the Customers – Orders relation will not allow the user to set Orders.CustomersID to a value that does not exist in the Customers table.
DeleteCascadeRule	This property specifies what happens to child rows when their parent row is deleted. **None** means that child rows are left unchanged. As a result of deleting the parent row, they become orphan rows, without parent. The **Cascade** value means that child rows are deleted. Set this property to **Cascade** when child rows belong to their parent, should not exist without it. There are also two less frequently used values: **SetNull** meaning that child rows remain with related field(s) set to null, and **SetDefault** meaning that child rows remain with related field(s) set to its default value.
UpdateCascadeRule	This property determines what happens to child fields when parent field values are changed. If it is set to **Cascade** (default), the child fields are changed correspondingly. If it is set to None, the child fields are left unchanged. If it is set to

	SetNull, the child fields are set to null. If it is set to **SetDefault**, the child fields are set to their default values.
UpdateCascadeServer	Cascade update, equivalent to what happens if UpdateCascadeRule is set to **Cascade**, can also be performed by the database itself. If this is the case, **DataObjects for .NET** must know that and adjust the process of updating the database. If this is not done, cascade update occurring in the database will conflict with cascade update already performed by **DataObjects for .NET** (if UpdateCascadeRule=**Cascade**). To prevent this conflict, if your database performs cascade updates for this relation, set the UpdateCascadeServer property to **True**. Note that there is no corresponding property for cascade delete, although cascade delete can also be performed by the database. If both a simple relation and the database perform cascade delete, make sure the IgnoreDeleteError property is set to **True** (default value). That will prevent conflicts between cascade deletes in **DataObjects for .NET** and in the database. The only possible conflict is that **DataObjects for .NET** tries to delete database records in the child table already deleted by the database as a result of deleting their parent record. If IgnoreDeleteError is set to **True**, this error (trying to delete already deleted record) will be ignored.

Composite Tables

A *composite table* is several simple tables bundled in one. A composite table row represents a row of each constituent simple table, as shown in the following figure:

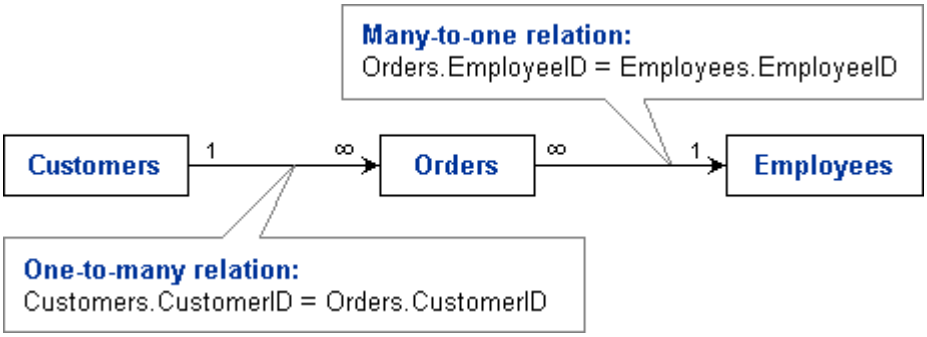


Composite table rows do not actually store the data shown on the figure, data belonging to the simple tables; they store pointers to simple table rows instead. See Structured Data Storage: Tables and Table Views for explanation on how data is stored in simple and composite tables.

Composite table support is one of the main power features of **DataObjects for .NET**. It allows you to define business objects with data stored in multiple database tables, objects whose structure is determined by the inherent business domain logic, not by the physical database structure that is often far from the logical structure, due to normalization and other database-specific techniques. Composite tables bridge the gap between physical database structure and logical structure that is needed in business logic and user interface design.

Composite tables represent business objects just as simple tables do. Developers can associate business logic code with composite tables, as with simple tables, using the C1TableLogic component events, see Table Business Logic Events. Each composite table has a class associated with it, with properties corresponding to composite table fields, see Using Typed Data Objects in Business Logic.

A composite table is defined by a diagram with simple tables as nodes connected with arc – relations. For example, the following is a diagram of the composite table shown in the figure above:



or, in brief notation:

`Customers` →(1-∞) `Orders` →(∞-1) `Employees`

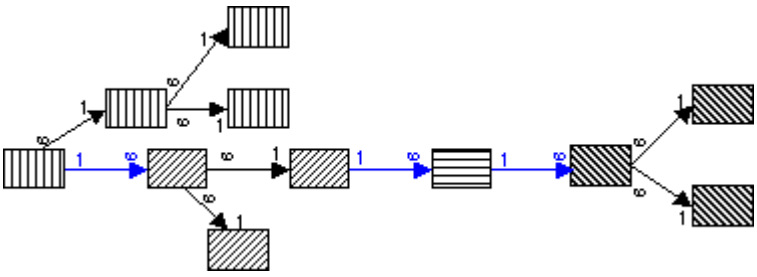
> **Note:** In practice, outer joins are not supported in **DataObjects for .NET**. One-to-many relations always produce inner joins. This is because every composite table row must contain non-null values for all its primary key fields, and that requires that for every one-to-many relation at least one child row exists for every parent row.

Composite Table Diagram

Not every diagram defines a valid composite table. First, it must contain only bound tables, unbound and SQL-based tables are not allowed, see Bound, SQL-Based and Unbound Tables. The diagram must also satisfy the following conditions ensuring that it represents a plain table, a rowset:

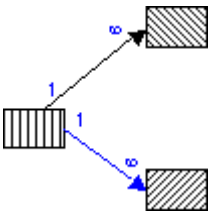
- It must be a tree, that is, a connected graph without loops: every two nodes are connected with an arc path, and there are no cyclic paths in the graph.
- One-to-many arcs must not branch. In formal terms, this means that for each group of nodes that are connected with many-to-one arcs to each other (or for a single node if it does not have adjacent many-to-one arcs), there can be only one (or none) outgoing one-to-many arc.

For example, the following diagram represents a valid composite table (one-to-many relations highlighted blue; groups of nodes interconnected many-to-one filled with the same pattern):

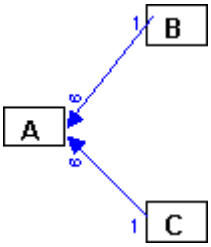


The following three diagrams do not represent valid composite tables:

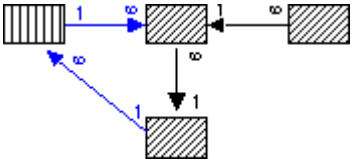
In the above image the diagram is not connected and consists of two independent parts.



The above image illustrates a one-to-many relations branching.



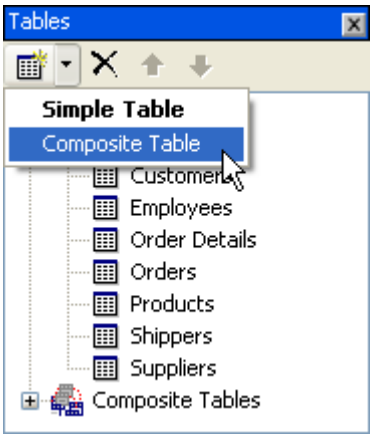
In the above image the diagram is not connected; there is no path from B to C and vice versa. To make this diagram a correct composite table, invert one of the relations.



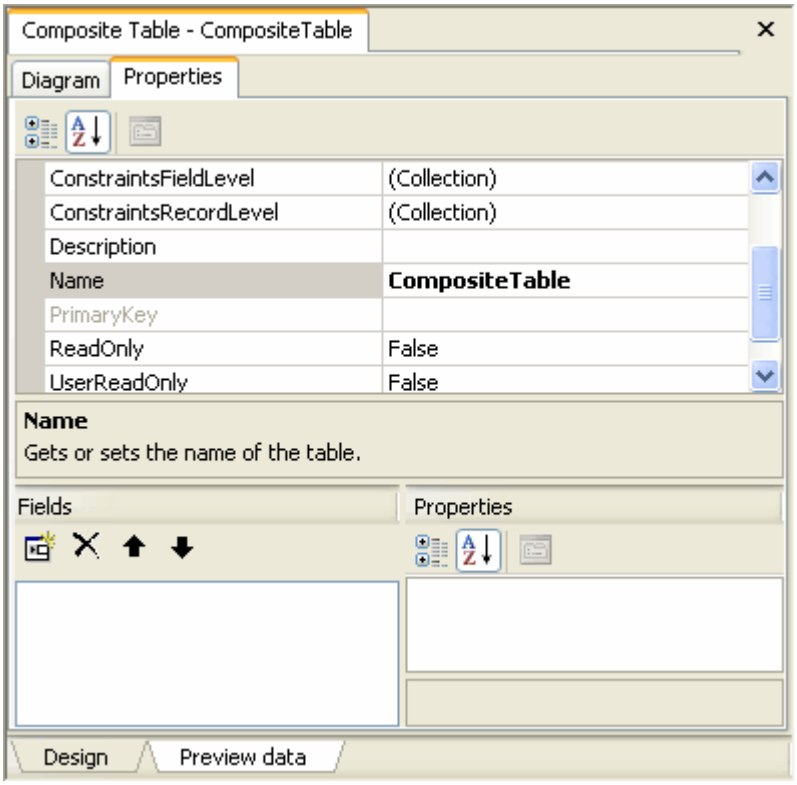
In the above image the diagram is not connected and contains a loop.

Creating and Modifying Composite Table Diagrams

To create a composite table, press the **Add** button or click the drop-down arrow next to the **Add** button and select**Composite Table** from the context menu in the **Tables** window.



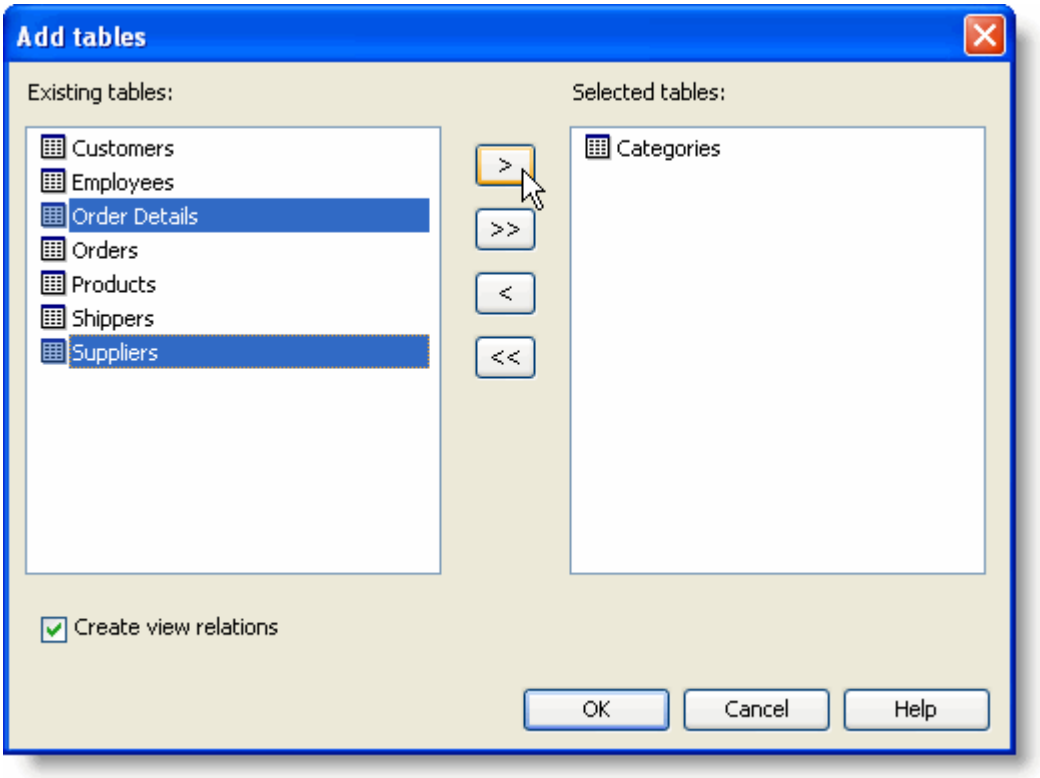
The empty new composite table will open as a new **Composite Table Editor** page. The **Composite Table Editor** has two tabs: **Diagram** and **Properties**.



The **Diagram** tab contains the diagram (graph) of constituent simple tables connected with relations. It also allows you to set properties of the graph's nodes, *table views* (CompositeDefView objects), and of the graph's arcs, *view relations* (CompositeDefRelation objects). The **Properties** tab consists of two panes: the upper pane showing the composite table properties and the lower pane showing the composite table fields and their properties.

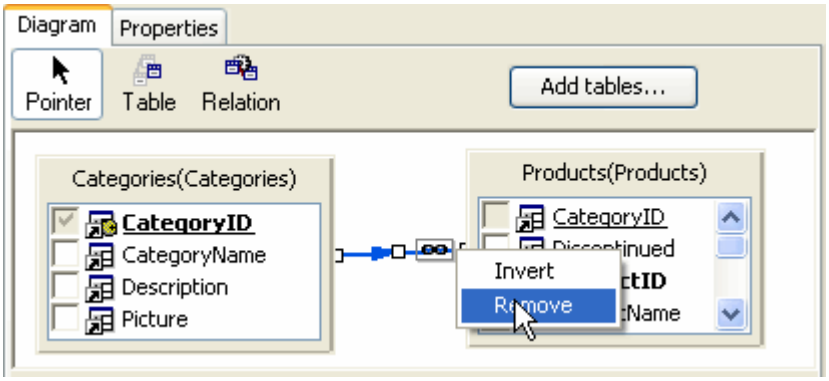
Immediately after creating a new composite table, it should be renamed by setting its **Name** property in the **Properties** tab or by renaming the composite table node in the **Tables** window.

Setting up a composite table starts with adding some simple tables to its diagram. Tables can be added to the diagram by clicking the **Add tables** button in the Diagram of the **Composite Table Editor**. The **Add tables** dialog box appears.

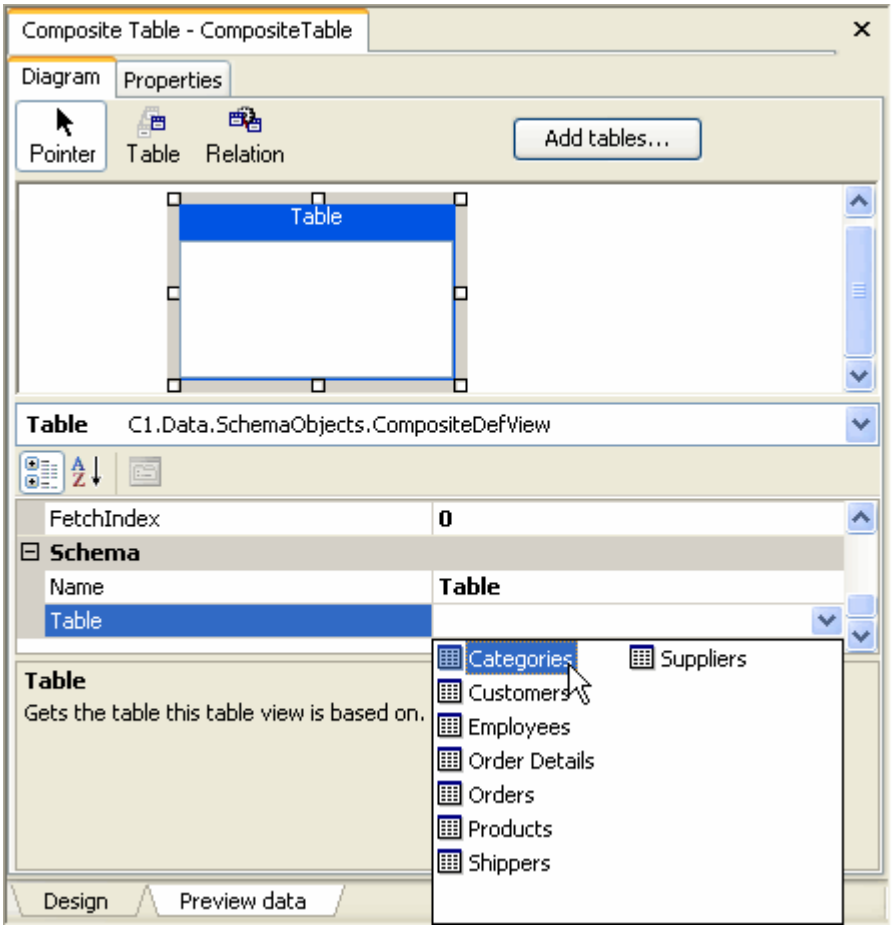


Select tables from the list of **Existing tables** and use the arrows to move them to the list of **Selected tables**.

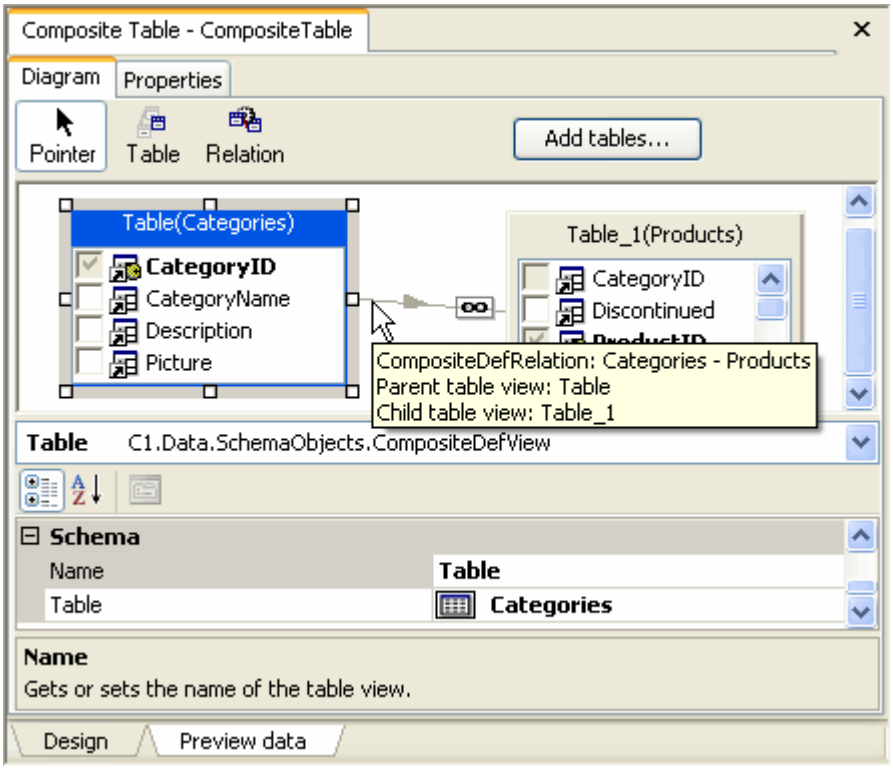
Table can also be added by dragging and dropping them from the **Tables** window onto the Diagram in the **Composite Table Editor**. When you add a table view to the diagram, relevant relations connecting it to other table views in the diagram are also added, unless suppressed by unchecking the "Create view relations" check box in the **Add tables** dialog box or by unchecking the "Automatically create relations adding tables to diagrams" check box in the **Options** dialog box (select **Options**in the **Schema** menu). Redundant relations can then be deleted by selecting a relation arrow in the diagram and pressing the Delkey or by right-clicking it and selecting **Remove** from the context menu.



You can also add tables to the diagram using the **Table** button on the top bar of the **Diagram** tab, but then you will have to set the Table property manually, selecting the newly created table view (a CompositeDefView object) and setting its Table property in the property grid below the diagram.



It is also possible to add relations to the diagram manually. Press the **Relation** button on the top bar of the **Diagram** window and draw an arrow connecting parent and child tables using a drag-and-drop operation from parent to child. **The Schema Designer** will try to find an appropriate relation connecting these two tables (inverting the relation, if necessary). When the new view relation is created, select it and look at popup box or the value of the Relation property in the property grid below the diagram. If this property is empty, it means that the **Schema Designer** could not find an appropriate relation.



The objects shown on the diagram are CompositeDefView and CompositeDefRelation objects.

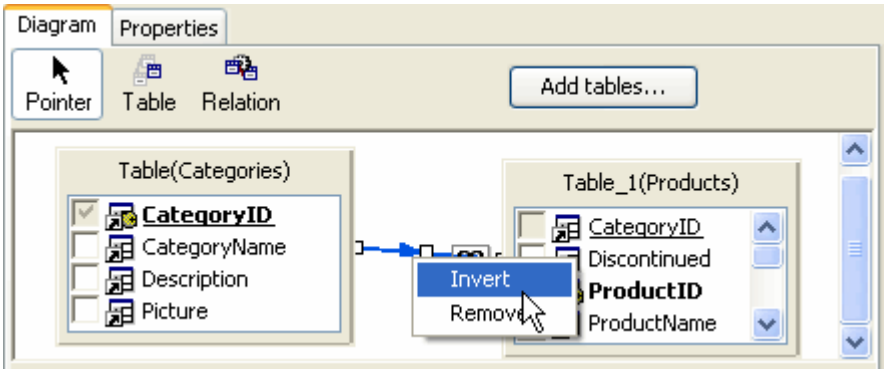
A table view (CompositeDefView object) represents a table, determined by its Table property, as a part of a composite table, as well as properties of its own. CompositeDefView properties in a composite table diagram can be used to override simple table properties. For example, you can set **ReadOnly** to **True**. This will make this table a read-only part of the composite table, although the simple table itself may be modifiable. The following CompositeDefView properties override Table properties: AllowAddNew, AllowDelete,and ReadOnly.

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. Although it is not common, this is sometimes useful to represent relations of a table with itself, what is called "self-joins" in SQL, as in Employee1.SupervisorID = Employee2.EmployeeID, where *Employee1* and *Employee2* are both aliases for the same table *Employee*. This feature is fully supported in composite tables, with a single restriction that if this table duplication occurs on the last level (the last table that is a child in a one-to-many relation is duplicated somewhere else in the same diagram), then adding new records to the composite table is not allowed, since it would lead to abnormal results.

A CompositeDefRelation object represents a relation, determined by its Relation property, as a part of a composite table diagram, an arc in the graph. The CompositeDefRelation class is derived from ViewRelation. Only simple relations can be used in a composite table diagram, its CompositeDefRelation objects cannot be based on composite relations.

Note: General ViewRelation objects can be based on any relation. For more information, see View Relations.

A view relation, as an arc in a composite table diagram, can have direction opposite to the direction of the relation on which it is based. For example, having a one-to-many relation *Customers – Orders*, we can create a many-to-one view relation *Orders – Customers*, based on *Customers – Orders*, but in inverse order (direction). So, if the Relation object *Customers – Orders* has Parent = Customers, Child = Orders, the view relation, if inverted, will have Parent = Orders, Child = Customers. This inversion does not change the original Relation object. The usual practice is to define all simple relations, Relation objects shown in the **Relations** window as one-to-many, and then apply them in diagrams, create view relations based on them, in direct (one-to-many) or inverse (many-to-one) order. To invert a view relation, right-click the relation in the diagram and select **Invert** from the context menu.

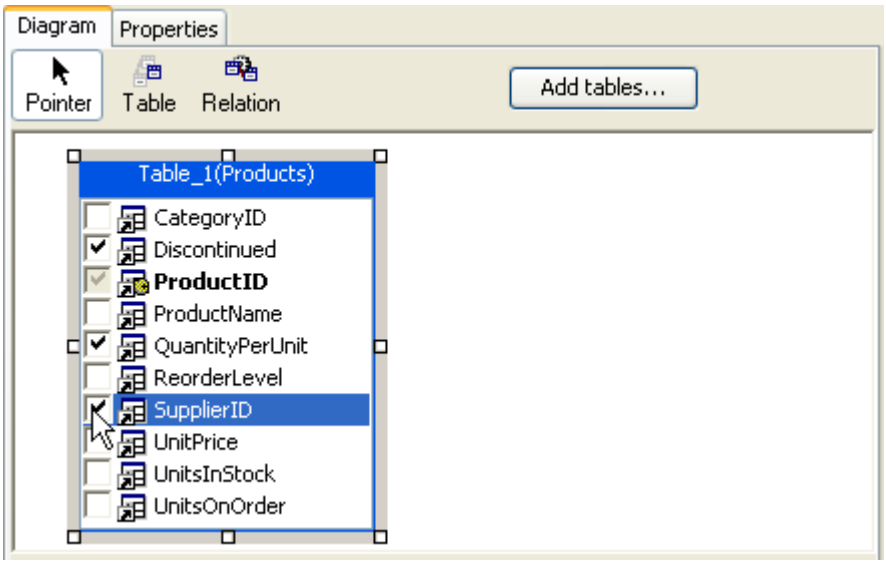


A CompositeDefRelation object has a number of properties in addition to the properties Relation, Parent and Child that determine the relation it represents and in which direction, direct or inverse, it is applied. In addition to that, it only has a read-only Cardinality property, returning *OneToMany* or *ManyToOne* according to the relation's Cardinality and the direction (if inverse, Cardinality is inverted), and the OuterJoinInManyToOne property affecting the process of generating SQL for data fetch. For more information, see How Composite Table Data is Fetched, Stored and Updated.

Composite Table Fields

After creating a composite table diagram and selecting simple tables and connecting them with relations, you must create composite table fields. Composite table fields define the structure of a composite table row. A composite table field is based either on a constituent simple table field, or on a calculated (unbound) field.

To define a composite table field based on a simple table field, use the **Diagram** tab of the **Composite Table Editor**. Every field of every simple table in the diagram has a check box. Check this check box to include it in the composite table field collection.

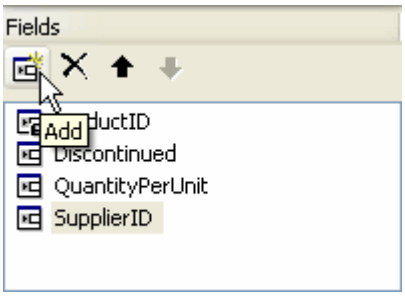


Checking the check box creates a CompositeTableField object referring to the simple table field via the TableViewField property. Primary key fields are always checked, they must be present in the composite table to form its primary key, so you cannot uncheck them. Child relation fields (fields that appear on the child side of a relation connecting simple tables) are always unchecked, they are not allowed to be included in the composite table because they have no independent meaning, and their values are always the same as the values of the corresponding parent keys.

The composite table field collection is shown in the **Properties** tab of the **Composite Table Editor**, in its bottom panel (its top panel shows the CompositeTable properties).

In addition to fields based on simple table fields, a composite table can contain unbound (calculated) fields. A

composite table field is unbound if its TableViewField property value is empty. Unbound fields are usually calculated either by Calculations (expressions) or in code. Unbound field values are stored in composite table rows, see Structured Data Storage: Tables and Table Views. To create an unbound field, click the **Add** button or select **Add** from the context menu in the **Fields** list.



A composite table field has a name that must be unique in the composite table. It has the same function as the name of a simple table field; for more information, see Table Fields. A field can be renamed by renaming the field node in the **Fields** list. Using this list and its button bar, you can add, delete and reorder composite table fields.

CompositeTableField properties are the same as the properties of a simple table field, although some properties are not applicable in composite table, and thus are hidden. For an unbound (calculated) composite table field, its properties have exactly the same meaning as simple table field properties, see Table Fields. For a bound composite table field (one based on a simple table field), its properties have some special inheritance-like features:

- Some properties inherit their value from the corresponding property of the simple table field (determined by the value of the TableViewField property) and that value cannot be changed unless it is changed in the original simple table field. Such properties are: DataType, NativeDbType, AutoIncrement, AutoIncrementSeed, AutoIncrementStep, DefaultValue, MaxLength, Precision, Scale, and Unique.
- Some properties can be changed, but their effective value at run time is determined by the conjunction of their value and the value of the property of the base simple table field. Such properties are: ReadOnly, ReadOnlyUnlessNew, and AllowDbNull.
- The remaining properties can be specified in a composite table field independently of their value in the base simple table field. Such properties are: CalculationCondition, CalculationExpression, Calculations, and Constraints.

Every modifiable table in **DataObjects for .NET** must have a primary key, and composite tables are no exception. A composite table's PrimaryKey is composed of bound composite table fields that belong to the primary key of their simple table (have PrimaryKey = **True**), except child relation keys (fields that appear on the child side of a relation connecting simple tables). This is the rule determining the value of the PrimaryKey property of a composite table field and the value of CompositeTable.PrimaryKey.

Composite Table Properties and Business Logic Events

A CompositeTable object has the same properties as the Table object (see Table Properties), except some that are not applicable to composite tables and thus hidden. There is also a AllowMultipleConnections property specific to CompositeTable (see How Composite Table Data is Fetched, Stored and Updated).

Business logic code can be associated with a composite table as well as with a simple table, in the form of event handlers of a C1TableLogic component (see Table Business Logic Events).

How Composite Table Data is Fetched, Stored, and Updated

As with simple tables, fetching data for composite tables originates from table views based on the composite table, see How the Data is Fetched. The resulting composite table rowset is the union of rowsets fetched by table views based on this composite table.

> **Note:** In practice, outer joins are not supported in **DataObjects for .NET**. One-to-many relations always produce inner joins. This is because every composite table row must contain non-null values for all its primary key fields, and that requires that for every one-to-many relation at least one child row exists for every parent row.

Fetching composite table data, **DataObjects for .NET** generates a SQL statement joining all involved simple tables with SQL joins. This is only possible if all constituent simple tables are bound tables sharing the same Connection. For more information, see Bound, SQL-Based and Unbound Tables. By default, this is a necessary condition for a composite table; it will throw an exception if this condition is not satisfied. However, by setting a special property AllowMultipleConnections to **True** you can override this restriction. Then **DataObjects for .NET** will apply a nested loop algorithm to fetch the composite table data from different database connections and/or from unbound data sources. The result will be the same as if all the data were stored in the same database, with the same connection, but performance may be much slower than with a single SQL statement.

The process of generating the SQL statement fetching data is automatic, but it can be affected by a few properties:

- OuterJoinInManyToOne (default: **True**): By default, many-to-one relations in a composite table diagram generate outer joins in the SQL statement. This ensures the standard interpretation of many-to-one links, where the child table contains one row corresponding to the parent row, or none. If there is no corresponding row, the child fields receive null values. However, some databases do not implement outer joins, or implement them inadequately, with various limitations. In this case, and when outer joins are undesirable for other reasons, you can set this property to **False**. **DataObjects for .NET** will use inner joins for this view relation in the composite table diagram. Note that one-to-many relations always produce inner joins in the generated SQL statement. This is because every composite table row must contain non-null values for all its primary key fields, and that requires that for every one-to-many relation at least one child row exists for every parent row.
- Alias (default: empty string): A generated SQL statement must use aliases for constituent simple tables, because a table can occur more than once in a composite table diagram, so its name may be not unique in the statement, in which case it needs aliasing. For more information, see Creating and Modifying Composite Table Diagrams. By default, view table names are used as aliases, but if you want to change the default alias, you can set it for a table view in the Alias property. This is rarely needed; this property is added only for completeness, since all other names in the generated SQL statement can be controlled by developers using DbTableName and DbFieldName.

If you want to see the generated SQL statement, set up a C1DataSetLogic component and attach a handler to its AfterGenerateSql event. The generated SQL statement is passed as an argument to that event.

Once the data is fetched, it is stored as described in Structured Data Storage: Tables and Table Views. This structured storage ensures that data is always kept in sync, with the schema structure preserved on all modifications, whether modifications are made from a simple table or from a composite table including that simple table.

When updating data, committing changes to the database does not involve composite tables. As described in How the Data is Modified, all changes end up in simple tables, so it is this simple table data that is sent to the server to update the database.

How to Access Composite Table Data

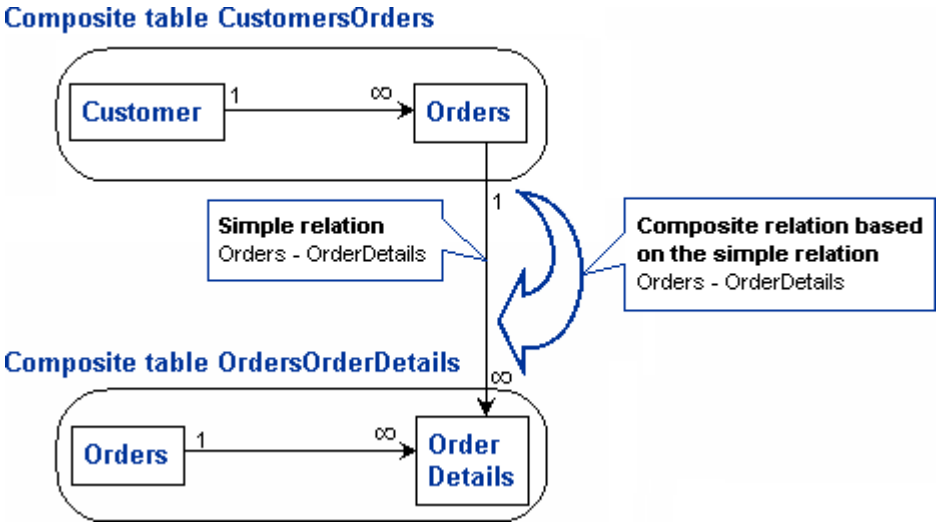
Composite tables do not differ from simple tables in the way their data is accessed. See How to Access Table Data and How to Access Table View Data for information on how to use data in data bound controls and programmatically from code.

Composite Relations

A composite relation (class CompositeRelation) defines a parent-child relationship between two tables one of which is a composite table. See Simple Relations about parent-child relationships between tables. If both parent and child are

simple tables, we use a simple relation to connect them. If at least one of them, or both, are composite tables, we use a composite relation.

A composite relation is based on a simple relation connecting one of the parent's constituent tables to one of the child's constituent tables, as in the following figure:



More exactly, it connects one of the parent's table views to one of the child's table views. The distinction between simple tables and table views must be made because a composite table can have multiple table views (CompositeDefView objects) based on the same table in its diagram. In this case, we need to know which of the table views actually participates in the relation.

Creating a composite relation starts in exactly the same manner as creating a simple relation. When you set either the relation's Parent or Child property to a composite table, the relation will become a composite relation.

When you set both Parent and Child of a composite relation, the **Schema Designer** looks for a simple relation connecting a parent's table view with a child's table view. If one is found, and if it is the only one possible simple relation, the SimpleRelation property is automatically set to this simple relation. If more than one simple relation is possible, the SimpleRelation property will not be set automatically, you will need to open the property combo box and select one of the applicable simple relations.

Once you select a SimpleRelation (or it is set automatically), the **Schema Designer** sets the properties ParentTableView and ChildTableView to the parent's and child's table views connected by the simple relation. This, however, is done only if the choice of such table views is unambiguous, that is, if both parent and child composite tables have only one table view based on the corresponding table. If the choice is non-trivial, that is, there is more than one table view to choose from, the properties ParentTableView/ ChildTableView are not set automatically; you will need to choose a table view in the property combo box.

Data Sets

A C1DataSet is a self-contained data and code unit filled with data according to the schema. It is the data as it is exposed to the user, which can be used both by data bound controls and programmatically. Any **DataObjects for .NET** client works with a C1DataSet. A data set also serves as a transaction context; modifications made in separate data sets belong to separate transactions. Modifications are not visible across data sets.

The structure of a data set is defined in the schema as a DataSetDef object. A schema can contain multiple data set definitions.

To fill a C1DataSet with data, you can either explicitly call Fill from code or set FillOnRequest property to **True** (the default value), making a data set fetch data at startup, when the data is requested.

When filling a data set with data, you must be aware that unless you specify filter conditions in code before calling Fill

or in BeforeFill event, each table view will fetch all existing table data unrestricted. To restrict fetch, you must specify filter conditions for each table view separately. Restricting one table view does not automatically restrict any other.

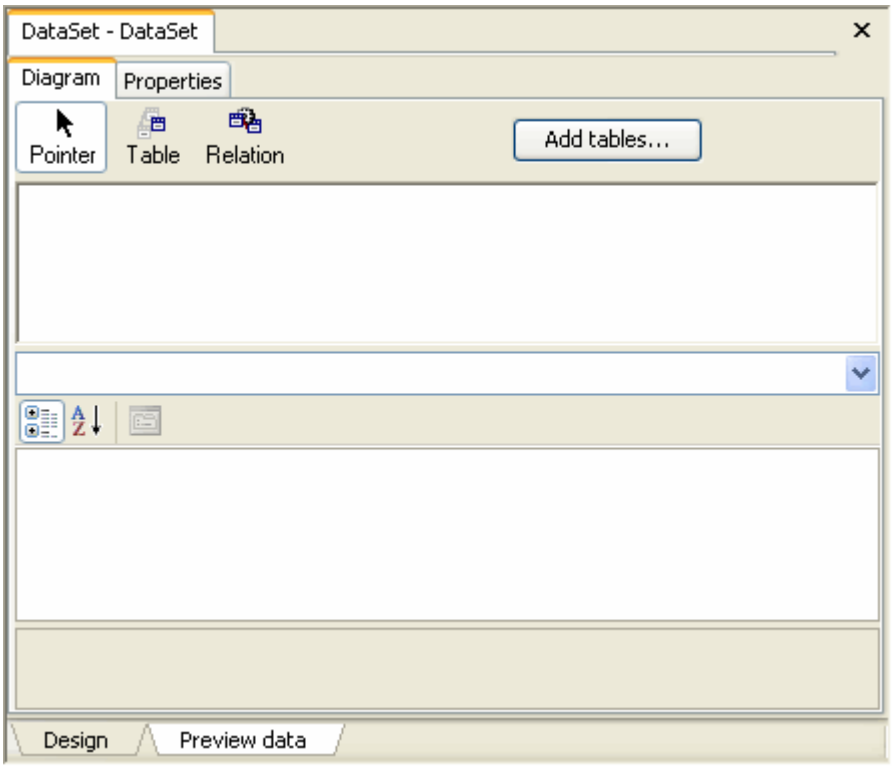
All data fetched for a C1DataSet object are retrieved in one call to the server, a Fill call, unless in virtual mode where data retrieval occurs in limited chunks of data, upon request, see Tutorial 4: Virtual Mode: Dealing with Large Datasets. The user application can then make modifications to the data in a C1DataSet locally, and send modifications to the data source (usually, a database) calling Update, or automatically, using a C1DataTableSource with UpdateLeavingRow property set to **True**.

A data set is a collection of table views (TableView objects) optionally connected with view relations (ViewRelation objects). Each table view represents a certain table, either simple or complex. There can be multiple table views representing a single table in a data set. It is most common for a table view to represent a table without changing anything, without modifying business logic and properties, and with table view fields exactly corresponding to the table fields. In this case, creating a data set definition in the **Schema Designer** is a simple matter of selecting tables that need to be represented by the data set (use the **Add Tables** button). However, in more advanced cases you can use the ability of a table view to override and modify the table's business logic and properties, and customize table view fields, see Table Views.

The set of table views you include in a data set determines what tables are filled with data in this data set. It is the developer's responsibility that all tables that are used by business logic code of a data set are present in the data set, otherwise an exception may occur when the code tries to access tables that have not been fetched. See How the Data is Fetched for details.

Creating and Modifying a Data Set Definition

To create a data set, open the **Schema Designer** and select **View | DataSets**. In the **DataSets** window, press the **Add** button  or select **Add** from the context menu. The empty new data set will open as a new **DataSet Editor** page.

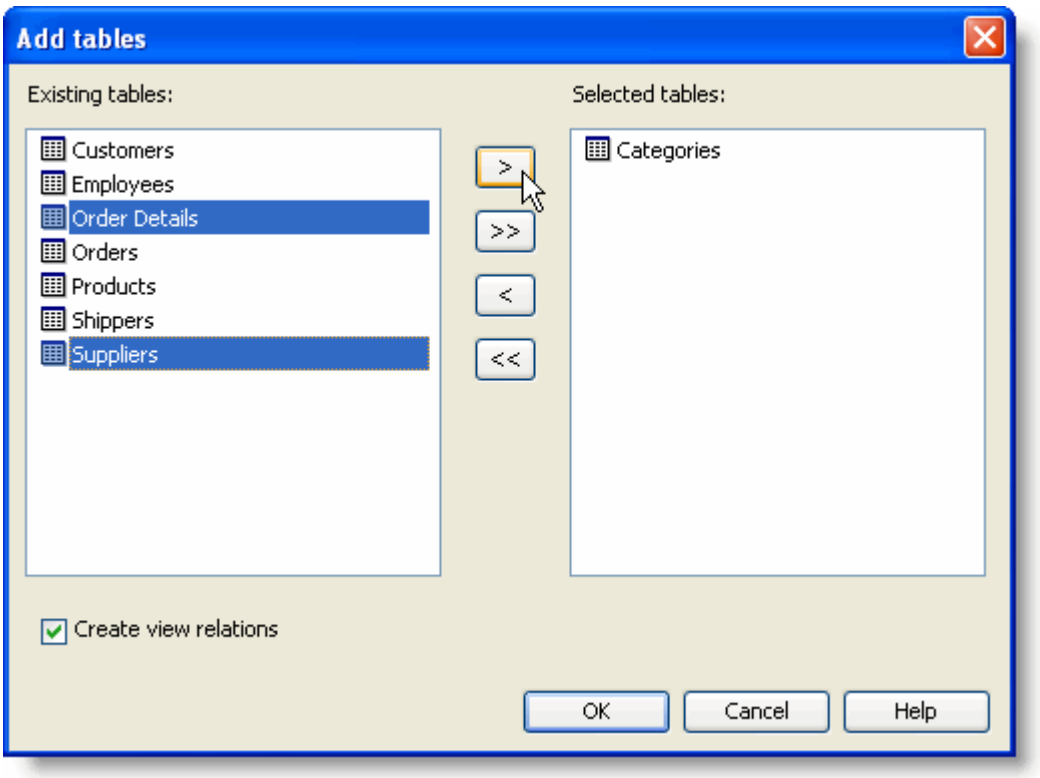


The **DataSet Editor Diagram** looks similar to the **Diagram** tab of the **Composite Table Editor** – it contains a diagram (graph) of table views connected with relations. It also allows you to set the properties of the graph's nodes: TableView and ViewRelation objects. Unlike the **Composite Table Editor**, it also allows you to set the properties of

table view fields (TableViewField objects).

Immediately after creating a new data set, you can rename it by renaming the DataSet node in the **DataSets** window.

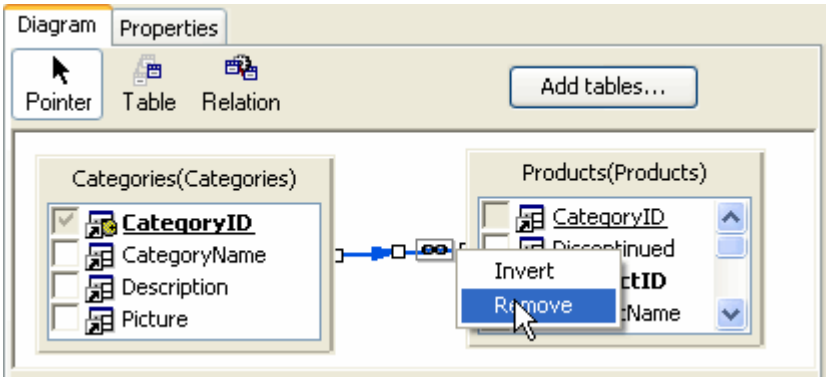
The first action when setting up a data set is creating table views and adding them to the diagram. Table views can be added to the diagram by clicking the **Add tables** button in the **DataSet Editor Diagram**. The **Add tables** dialog box appears.



Select tables from the list of **Existing tables** and use the arrows to move them to the list of **Selected tables**.

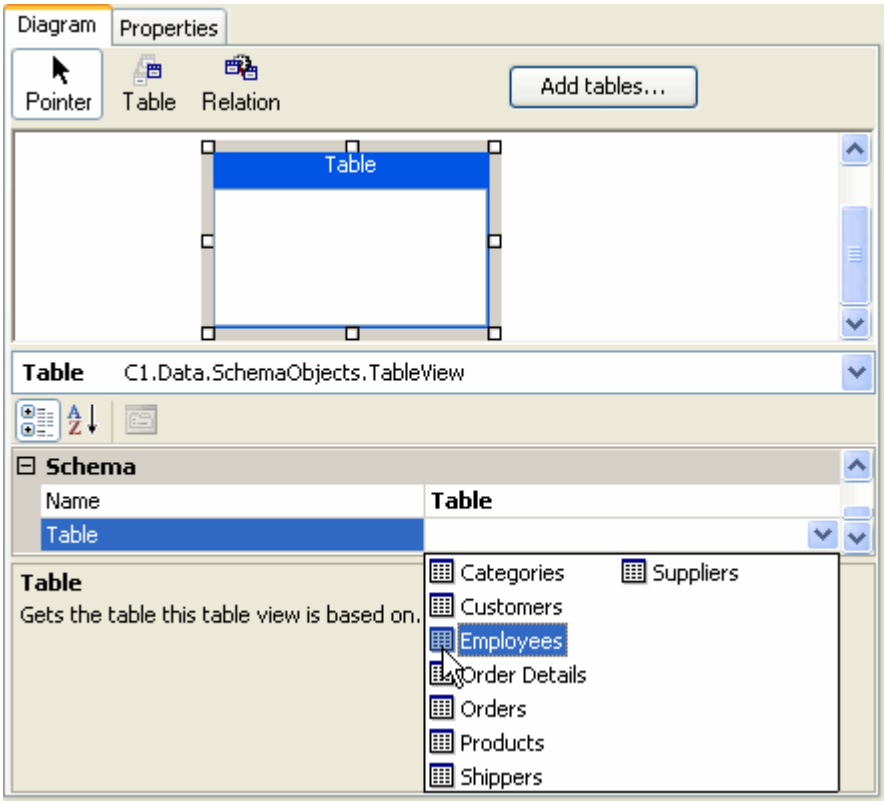
Table views can also be added by dragging and dropping them from the **Tables** window onto the Diagram of the DataSet Editor.

When you add a table view to the diagram, relevant relations connecting it to other table views in the diagram are also added, unless suppressed by unchecking the "Create view relations" check box in the **Add tables** dialog box or by unchecking the "Automatically create relations adding tables to diagrams" check box in the **Options** dialog box (select **Options**in the **Schema** menu). Redundant relations can then be deleted by selecting a relation arrow in the diagram and pressing the Delkey or by right-clicking the relationship and selecting **Remove** from the context menu.

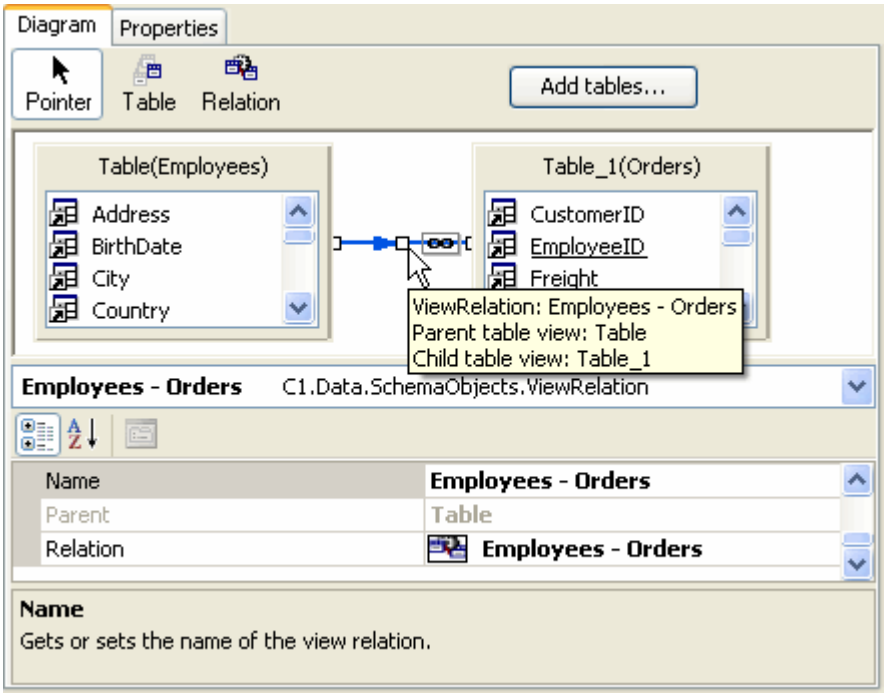


You can also add tables using the **Table** button on the top bar of the Diagram. Simply click the **Table** button, then

click the diagram, and the table is added. In this case, you will have to set the Table property manually, selecting the newly created table view and setting its Table property in the property grid below the diagram.



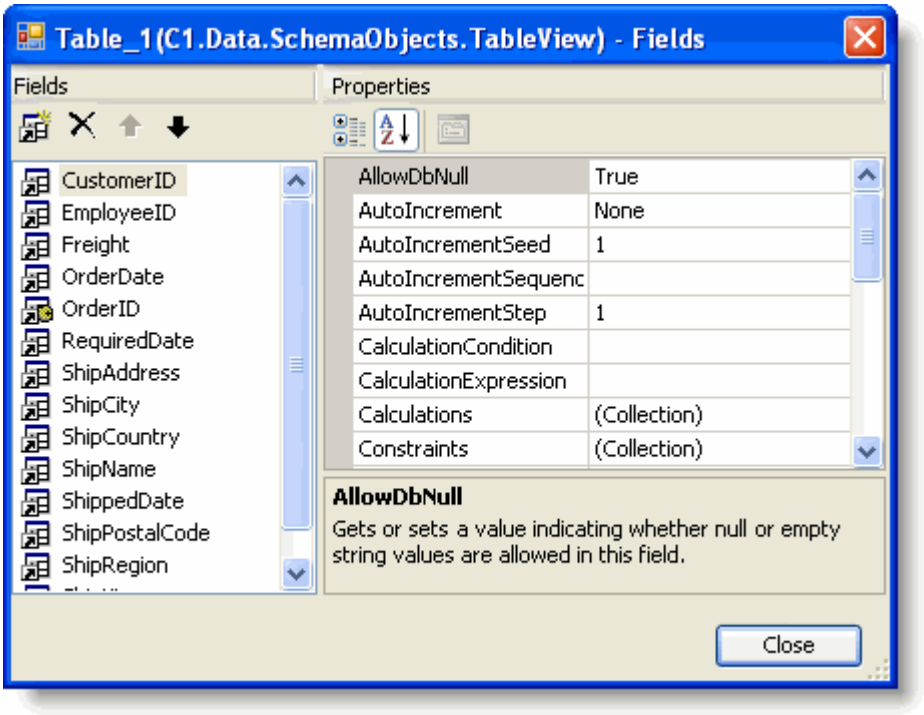
It is also possible to add relations to the diagram manually. Click the **Relation** button on the top bar of the Diagram and draw an arrow connecting parent and child tables using drag-and-drop from parent to child. The **Schema Designer** will try to find an appropriate relation connecting these two tables (inverting the relation, if necessary). When the new view relation is created, select it and look at popup box or the value of the Relation property in the property grid below the diagram. If this property is empty, it means that the **Schema Designer** could not find an appropriate relation.



The objects shown on the diagram are TableView and ViewRelation objects. See Table Views and View Relations for detailed descriptions. To set properties of any object on the diagram, including table views, table view fields and view relations, select the object on the diagram and browse/set its properties in the property grid below the diagram. To select a view relation, select the arrow representing the view relation. If it represents multiple view relations, you will need to choose one of them from a context menu that pops up when you select the arrow. To select a table view field, select an item in the list of fields shown inside a table view.

By default, when you add a table view, its **Fields** collection is filled with all fields of the corresponding table. More exactly, since TableViewField and TableField are different objects, it means that the table view contains a TableViewField representing each TableField of the corresponding table.

If you need to add a table view field (for example, a calculated, unbound field, see Table View Fields), or delete some table view fields or re-arrange their order, right-click the table view and select **Fields** from the context menu. In the **Fields** dialog box you can also set properties of table view fields, although that can be done in the diagram too, in the property grid below the diagram.



If you want to restore the collection of table view fields to its initial state, as if the table view has just been added to the diagram, right-click the table view and select **Retrieve Fields** from the context menu. This will create TableViewField objects anew, one table view field for each table field.

Table Views

A TableView object represents a table in a data set. It can represent a simple or a composite table. The table is determined by its Table property.

TableView also has properties of its own. Some TableView properties can be used to override table properties. For example, you can set ReadOnly to **True**, which will make the table view read-only, although the table itself may be modifiable. The following TableView properties override Table properties: AllowAddNew, AllowDelete, ReadOnly.

Other TableView properties work independently of or in addition to Table properties:

The PrimaryKey property is read-only. It shows the field name(s) constituting the primary key of the table on which the table view is based. Multiple field names are separated with commas. A table view's primary key is the same as the corresponding table's primary key. The only purpose of having a special PrimaryKey property in TableView is that table view field names can differ from table field names, so the string value may not be the same. Note that primary key table fields are always represented by table view fields, such table view fields are not allowed to be deleted. This insures that each table view has proper primary key.

The FillSort property controls the order in which data rows are sorted after fetch. The order can also be specified at run time, calling Fill method, by specifying FillSort in a FilterCondition. By default, fetched data is sorted by primary key. If a different sort is required, set the FillSort property to the sort field name(s). To specify sort order (ascending/descending), add "ASC" (ascending) or "DESC" (descending) after the field name. If no order is specified, the order is "ASC" (ascending). Multiple field names are separated with commas.

Example: "CustomerID DESC, OrderID".

ConstraintsFieldLevel and ConstraintsRecordLevel are collections of table view constraints, ConstraintInfo objects. Constraints are expressions, see DataObjects for .NET Expressions. These constraints are tested in addition to table constraints when data is modified in this table view, see Table Properties. If you have a constraint that must always be enforced for a given table, must be satisfied in all table views representing that table, put it in the table's constraint collection. If, on the other hand, you have a constraint that is specific to a particular table view, then it belongs to the

table view's constraint collection.

ConstraintsRecordLevel contains record level constraints that are evaluated when the user finishes editing a row, before the C1DataSetLogic BeforeEndEdit event. If one of the constraints is not satisfied, an exception is thrown. The exception message is determined by ErrorDescription. Constraints with Condition expression (if non-empty) evaluating to **False**, are skipped, and not tested.

ConstraintsFieldLevel contains field level constraints that are evaluated on every field change made in this table view. Usually, a field level constraint belongs to a certain field, not to the table view as a whole, in which case it resides in Constraints. ConstraintsFieldLevel should contain only field level constraints that cannot be associated with a particular field. Constraints in ConstraintsFieldLevel are evaluated each time any table view field changes, whereas constraints in Constraints are evaluated on the owner field change.

The AutoEndAddNew property determines behavior of newly added rows. If it is set to **True** (default), setting the row's primary key triggers adding the new row to the table. If it is set to **False**, the newly added row is considered temporary, not belonging to the table, until the **EndAddNew** method is explicitly called.

The DataAccessMode property determines whether the table view functions in virtual mode, and specifies which of the three possible virtual modes is used. **DataObjects for .NET** virtual mode allows you to use large datasets in .NET Windows Forms applications, a feature that is not supported in Visual Studio and ADO.NET without **DataObjects for .NET**. In virtual mode, data retrieval occurs in limited chunks of data, upon request, instead of pre-fetching all data at once. See Tutorial 4 for more details.

The following options are available for the DataAccessMode:

- The default DataAccessMode value, **Static** means that virtual mode is not used.
- Data access mode **Virtual** means that data is fetched in chunks (called *segments*) of limited size (the size of each segment is determined by the VirtualSegmentSize property, default: 400), and the number of segments cached at the client at any given time is limited (this number is determined by the VirtualSegmentCount property, default: 4). Mode ***Virtual*** works for rowsets of virtually unlimited size, for example, in Tutorial 4 we demonstrate how it can be used to display 2.7 million rows in a grid.
- Data access mode **VirtualUnlimited** is similar to **Virtual** in that data is fetched in segments, but the number of segments in the cache is unlimited, the VirtualSegmentCount property value is ignored. Once a segment is brought into cache, it remains there. This setting is appropriate when you want to enhance performance by eliminating redundant roundtrips to the server, but it should not be used with very big rowsets if the user is expected to fetch many segments into memory.
- In **VirtualAutomatic** mode, data is fetched in segments, as in the previous two modes, and the number of segments in the cache is unlimited as in **VirtualUnlimited** mode, and, in addition to that, fetch is continually performed in background mode, asynchronously, until all data is fetched. This mode is appropriate for large rowsets that are big enough to make it undesirable to fetch all data at startup time, but not too big, so they still fit in client memory. This mode is additionally qualified by a Boolean VirtualConsolidateRows property. If it is set to **True** (default), then **DataObjects for .NET** will rebuild the rowset when fetch is complete. While fetch is incomplete, Rows contains only rows of the current segment. Once fetch is complete, Rows contains all fetched rows, the whole rowset. In the other two virtual modes, Rows always contains only rows of the current segment.

The following properties are performance tuning settings for virtual mode, must be changed only if default values are unsatisfactory: VirtualAsyncThreshold, VirtualConsolidateRows, VirtualSegmentCount, VirtualSegmentSize, and VirtualSyncThreshold; see Tutorial 4 for more information.

Table View Fields

Table view fields (TableViewField objects) define the structure of a table view row. A table view field is either based on a table field (a field of the table on which this **TableView** is based, according to its **Table** property), or it is a calculated (unbound) field.

By default, table view fields are in one-to-one correspondence with table fields. To customize the table view **Fields**

collection, right-click a table view and select Fields from the context menu that appears; this will open the **Fields** dialog box. In the **Fields** dialog box you can add new fields, including unbound (calculated) fields, you can also delete fields and re-arrange their order (note that primary key fields can be deleted; primary key table view fields are always present to ensure that the table view has proper primary key). The **Fields** dialog box also allows you to set properties for table view fields. If you want to restore the collection of table view fields to its initial state, select the table view, right-click a table view and select **Retrieve Fields** from the context menu.

In addition to fields based on table fields, a table view can contain unbound (calculated) fields. A table view field is unbound if its TableField property value is empty. Unbound fields are usually calculated either by Calculations (expressions) or in code. Unbound field values are stored in table view rows; for more information, see Structured Data Storage: Tables and Table Views. To create an unbound field, click the **Add** button or select **Add** from the context menu in the **Fields** list.

A table view field has a name that must be unique in the table view. It has the same function as the table field name (for more information, see Table Fields). A field can be renamed by renaming the field node in the **Fields** list.

TableViewField properties are the same as the properties of a simple table field (see Table Fields), except some field properties do not apply to table view fields; for example, properties controlling database update, such as UpdateIgnore, UpdateSet, and so on do not apply to table view fields and are hidden. Table views do not participate in updating the database (see How the Data is Modified). An unbound (calculated) table view field's properties have exactly the same meaning as table field properties (see Table Fields). A bound table view field's (one based on a table field) properties have some special inheritance-like features:

- Some properties inherit their value from the corresponding property of the table field (determined by the value of the TableField property) and that value cannot be changed unless it is changed in the original simple table field. Such properties are: DataType, NativeDbType, AutoIncrement, AutoIncrementSeed, AutoIncrementStep, DefaultValue, MaxLength, Precision, PrimaryKey, Scale, and Unique.
- Some properties can be changed, but their effective value at run time is determined by the conjunction of their value and the value of the property of the base table field. Such properties are: ReadOnly, ReadOnlyUnlessNew, and AllowDbNull.
- The remaining properties can be specified in a table view field independently of their value in the base table field. Such properties are: CalculationCondition, CalculationExpression, Calculations, and Constraints.

Table View Business Logic Events

You can associate business logic code with table views using the C1DataSetLogic component. Use this component when you need data set-specific (in other words, table view-specific) logic rules that must be enforced in the context of a particular data set and table view, but do not always apply to the underlying table.

Only one C1DataSetLogic component is allowed for each data set. To create C1DataSetLogic components, use the **Create Business Logic Components** menu item of the C1SchemaDef component, or add it manually from the Toolbox and set its SchemaComponent and DataSetDef properties. Then you can attach event code either selecting the components on the designer surface, or using the **Business Logic Events** tool window (select **Business Logic Events** from the C1SchemaDef context menu to open the tool window). The **Business Logic Events** tool window shows the list of all tables and data sets. When you select a data set in the tool window, the data set's business logic events appear in the Properties window (in Visual C#, when the Events radio button is selected in the Properties window; in Visual Basic use the Method Name combo box in the code editor).

See Table Business Logic Events for the list of event handlers that can be used to specify business logic.

View Relations

View relations (ViewRelation objects) are depicted in a data set diagram with arrows connecting nodes - table views. View relations are optional; it is possible to have a data set without them. View relations play the following roles:

- They define the master-detail (parent-child) hierarchy between table views. When you use a data set with view

relations as the DataSource for data bound controls, you can bind one grid to a master (parent) table view and another grid to a detail (child) table view. Then the detail grid will show the child rows related to the parent row that is current in the master grid at the moment. See also, How to Access Table View Data.

- They allow programmatic access to parent and child rows of a table view row, using methods GetParentRow and GetChildRows. Note that even without view relations; it is possible to navigate from parent to child rows and vice versa between tables, for table rows, using the same C1DataRow methods GetParentRow /GetChildRows and relations between tables. For more information, see Simple Relations and Composite Relations.

A view relation between table views is usually based on the relation (Relation object) between the two tables represented by those two table views (except for *custom* view relations, see Custom View Relations). This relation can be simple or composite (see Simple Relations and Composite Relations). It is determined by the ViewRelation.Relation property. A view relation, as an arc in the data set diagram, can have direction opposite to the direction of the table relation on which it is based. For example, having a one-to-many relation *Customers – Orders*, we can create a many-to-one view relation *Orders – Customers*, based on *Customers – Orders*, but in inverse order (direction). So, if the Relation object *Customers – Orders* has Parent = Customers, Child = Orders, the view relation, if inverted, will have Parent = Orders, Child = Customers. This inversion does not change the original Relation object. Usual practice is to define all simple relations, Relation objects shown in the **Relations** window as one-to-many, and then apply them in diagrams, create view relations based on them, in direct (one-to-many) or inverse (many-to-one) order. To invert a view relation, right-click the relation in the diagram and select **Invert** from the context menu that appears.

Custom View Relations

A custom view relation is not based on any simple relation between tables. It is used when you need a more complicated algorithm than simple equality of table fields. In a custom view relation, the set of child and (optionally) parent rows is defined in code in special events.

> **Note:** For an example where a custom relation is used to represent a many-to-many relation that cannot be based on a single simple relation between tables, see the **CustomRelations** sample, which is installed with the **Studio for WinForms** samples.

A view relation is considered custom if its GetRowsEvent property is set to **True**. Then it uses GetChildRows event to obtain child rows, and, if OneWay = False, it uses GetParentRow event to obtain parent row. If OneWay is set to **True**, the relation is one-way, only child rows are defined, and the parent row is not accessible from a child row.

To create a custom relation, drag an arrow from parent to child table view in a data set diagram using the **Relation** button on top of the data set editor. Select the arrow to set properties. Specify the Name property, set GetRowsEvent to **True** (leave the Relation property empty). Use a C1DataSetLogic component attached to the data set to write code in the events GetChildRows / GetParentRow specifying the algorithm for obtaining child rows and (optionally) the parent row for this relation.

How to Access Table View Data

The **DataObjects for .NET** gateway to data is the C1DataSet component. Every **DataObjects for .NET** client uses one or more C1DataSet components to bind or gain access to data. Setting up a C1DataSet component, you need to:

- Attach it to the schema either by setting the SchemaDef property (if a C1SchemaDef component resides on the same design surface, a case that is called *direct client*) or by setting its DataLibrary property (if the schema resides in a *data library*, case called *library client*) and, optionally, setting the DataLibraryUrl property (if the data library resides on a remote computer, a case called *remote client*). See Application Configurations for detailed description of different configuration and deployment options and explanations of the terms direct client, data library and remote client.
- When the C1DataSet component is attached to a schema, set its DataSetDef property choosing a data set

definition name in the combo box showing all data sets defined in the schema.

Data Binding

Using a C1DataSet component as your data source, you can bind data-aware components such as a grid or a text box to the data. The C1DataSet component exposes table view rowsets. Data-aware controls can bind to any of these rowsets. To bind to a table view rowset, set the **DataSource** property of the data-aware control to the C1DataSet component and open the **DataMember** property combo box. It will show the selection of available table views. Select the table view you wish to bind to.

DataObjects for .NET also supports master-detail (parent-child) data binding. Parent and child table views are related by View Relations. You can bind one control to a master (parent) table view and another control to a detail (child) table view. After doing this, the detail control will show the child rows related to the parent row that is current in the master control at the moment. Parent-child hierarchy is shown in the **DataMember** property combo box as a tree of view relations starting from topmost (master) table views. Topmost table views, roots of parent-child hierarchies, are distinguished by the leading underscore in their names. The same table view also appears in the DataMember list without an underscore, as a stand-alone table view. Select a stand-alone table view if you do not want this data-aware control to participate in a parent-child hierarchy. Select a topmost master table view, starting from underscore, if you want to bind it to a master (topmost parent) rowset. Select a node in a hierarchy tree, a view relation name, if you want to bind it to a child in a parent-child hierarchy.

You can also bind data-aware controls at run time, setting their **DataSource** and **DataMember** properties, for example:

To write code in Visual Basic

VB

```
ParentGrid1.DataMember = "_Customers"
ParentGrid1.DataSource = dataSet1
ChildGrid1.DataMember = "_Customers.Customers-Orders"
ChildGrid1.DataSource = DataSet1
```

To write code in C#

C#

```
parentGrid1.DataMember = "_Customers";
parentGrid1.DataSource = dataSet1;
childGrid1.DataMember = "_Customers.Customers-Orders";
childGrid1.DataSource = dataSet1;
```

Here *Customers* is the parent table view, *Orders* is the child table view, *Customers-Orders* is a view relation between them.

Another run-time option is to bind directly to the table view's C1DataTable object, as in the following example:

To write code in Visual Basic

VB

```
Dim Dt As C1DataTable
Dt = C1DataSet1.TableViews("Customers")
DataGrid1.DataSource = Dt
```

To write code in C#

```
C#
C1DataTable dt = c1DataSet1.TableViews["Customers"]
dataGrid1.DataSource = dt
```

If you need additional sorting and filtering, you can bind to a C1DataView component, both at design time and at run time:

To write code in Visual Basic

```
VB
DataGrid1.DataSource = DataView1
```

To write code in C#

```
C#
dataGrid1.DataSource = dataView1;
```

C1DataView allows filtering and sorting the data using its RowFilter and Sort properties. The RowFilter property filters data rows according to a Boolean expression. If you need a more advanced filtering algorithm that cannot be specified as a simple expression you can use the GetRows event to filter data in code.

C1DataView can also be used to filter/sort the data set itself, using the IsDefault property. If IsDefault = False (default), filtering/sorting is applied to the C1DataView component only, affects only controls bound to that C1DataView. If IsDefault = True, filtering/sorting defined in the C1DataView is applied to the data set itself, affects all controls bound to the data set in any way, not just those using the C1DataView as their DataSource. This filtering/sorting applies to hierarchical master-detail binding as well as to simple binding.

Another design-time option for data binding is to bind to a C1DataTableSource that is attached to a certain table view of a data set. Using a C1DataTableSource allows you to handle business logic events in your client application code, in addition to the business logic specified in the data library (see Business Logic for details). Using C1DataTableSource is mandatory if your client application works in virtual mode (see Tutorial 4: Virtual Mode: Dealing with Large Datasets for details).

If you bind third-party (not ComponentOne or Microsoft) controls to **DataObjects for .NET** data sources, some of them (very few) may work incorrectly with notifications, that is, with the IBindingList.**ListChanged** event. **DataObjects for .NET** uses a slightly different notification scheme than ADO.NET does. **DataObjects for .NET** notifications are fully .NET data binding-compliant and optimized for data bound controls. All ComponentOne bound controls and Microsoft bound controls work properly with this notification scheme. In fact, all data bound controls should work with **DataObjects for .NET** notifications without problems, but in the non-ideal world that can be broken. To prevent this problem and allow using any bound controls, **DataObjects for .NET** provides a special NotificationModeFlags property.

Programmatic Access

To access a table view rowset, obtain a C1DataTable object from a C1DataSet component (using the TableViews collection property), and use the Rows collection containing C1DataRow objects representing table view rows. To perform actions on table view rows or table view data as a whole, use appropriate methods and properties of the C1DataTable and C1DataRow classes. Their object model is similar to that of ADO.NET.

DataObjects for .NET also allows typed access to table and table view rows. For each table and table view, **DataObjects for .NET** generates a class representing its row. For example, **ProductsRow** is an object (business object, data object) where each field has a corresponding property (**ProductsRow.UnitPrice**, **ProductsRow.UnitsInStock**, and so on) and each relation has a corresponding method (**Products.GetOrder_DetailsRows**, and so on) allowing you to obtain child rows and the parent row). Using these classes, you can write your business logic code in a convenient,

type-safe way, and benefit from Visual Studio code completion features giving you the lists of properties and methods to choose from. See Using Typed Data Objects in Business Logic, for details about *data object classes*.

To use a data object class in your code, call the static **Obj** method implemented in every data object class. It obtains a business object given a C1DataRow object as its argument. For example, the following code obtains a **ProductsRow** business object from a C1DataRow:

To write code in Visual Basic

VB

```
Dim dataRow As C1.Data.C1DataRow
Dim product As DataLibrary.DataObjects.DataSet.ProductsRow
product = DataLibrary.DataObjecs.DataSet.ProductsRow.Obj(dataRow)
```

To write code in C#

C#

```
C1.Data.C1DataRow dataRow;
DataLibrary.DataObjects.DataSet.ProductsRow product;
product = DataLibrary.DataObjects.DataSet.ProductsRow.Obj(dataRow);
```

Business Logic

DataObjects for .NET uses the standard business object paradigm to allow you to develop business logic components (data libraries) and reuse them in multiple client projects. This provides clear separation of business and data logic from the presentation (GUI) layer. See Application Configurations for the detailed description of the *data library* concept. Developers write business logic code in the events of special C1TableLogic and C1DataSetLogic components in a data library.

DataObjects for .NET enables unprecedented code reuse power and flexibility by allowing you to specify business logic on all levels: from the most general level of simple tables for rules that must be enforced for tables regardless of the context where they are used, to more specific levels of composite tables, then data sets, and finally the most specific level of a concrete rowset instance. For more information, see How Business Logic Works on Different Levels.

When writing business logic code, you can use Using Typed Data Objects automatically generated by **DataObjects for .NET**. In this way, you can write code in a convenient, type-safe way, and benefit from Visual Studio code completion features giving you the lists of properties and methods to choose from.

In addition to business logic events, **DataObjects for .NET** allows you to define business methods in your data library, see Business Methods. Business methods are called by the client and executed on the server. They can fill data sets with data, update the database and perform any other required operations.

How Business Logic Works on Different Levels

The same business logic events, such as AfterFieldChange or BeforeDelete, can be handled on different levels:

Level	Description
Simple table level	Events of a C1TableLogic component (see Table Business Logic Events). They are triggered in any context, wherever the table is involved: in a composite table including this table, and in a table view based on this table. For example, if you write code converting *CustomerID* value to upper case in *table_Customers_BeforeFieldChange*, it will work for the *Customers* table itself, and for a *CustomersOrders* composite table that includes *Customers*, and for any table view

	based on such tables. In other words, the business rule will be automatically enforced everywhere it should be, in any possible context where the *Customers* table is used.
Composite table level	Events of a C1TableLogic component representing a composite table. They are triggered in any data set where the composite table is used. They are not triggered for constituent simple tables when they are used outside of this composite table context. For example, if you write some code in a *CustomersOrders* composite table event, it will be executed when the user works with *CustomersOrders*, but not when they work with the *Customers* table stand-alone.
Table view level	Events of a C1DataSetLogic component (see Table View Business Logic Events). They are active in a particular data set. If there are multiple C1DataSet objects with the same schema definition (DataSetDef; see Schema Objects), they are active in any such object. This business logic level represents data set-specific (in other words, table view-specific) rules that must be enforced in the context of a particular data set and table view, but do not always apply to the underlying table.
Rowset level	Events of a C1DataTableSource control. This level is different from all the rest in that it belongs to a particular client application, as opposed to a data library. All other logic levels are enforced by **DataObjects for .NET** in any application that uses their data library. Rowset level logic is enforced only in the application where this code is written.

Consider for example, a *CustomersOrders* composite table, including a simple *Customers* table, and a data set *MyOrders* having *MyOrdersView* table view based on the *CustomersOrders* table. Imagine a form in a client application with a C1DataSet dataSet1 connected to MyCustomersView, a C1DataTableSource C1DataTableSource1 connected to dataSet1, and a grid bound to C1DataTableSource1. Imagine that we have business logic event code BeforeFieldChange specified on all levels. When the end user modifies the **CustomerAddress** field in the grid (the field belongs to the *Customers* table inside the *CustomersOrders* composite table), the order in which the events are triggered will be from the most specific to the most general:

```
C1DataTableSource1_BeforeFieldChange (rowset level) →
dataset_ MyOrders_BeforeFieldChange (table view level) →
table_CustomersOrders_BeforeFieldChange (composite table level) →
table_Customers_BeforeFieldChange (simple table level)
```

For the corresponding "after" event, AfterFieldChange, the order will be the inverse, from the most general to the most specific:

```
table_Customers_AfterFieldChange (simple table level) →
table_CustomersOrders_AfterFieldChange (composite table level) →
dataset_ MyOrders_AfterFieldChange (table view level) →
C1DataTableSource1_AfterFieldChange (rowset level)
```

Business Logic Events

These sections describe particular **DataObjects for .NET** business logic events in detail.

Each event is fired by **DataObjects for .NET** at appropriate time defined below. Firing an event, **DataObjects for .NET** uses the inheritance chain of levels described in How Business Logic Works on Different Levels. For example, when we say that "the BeforeFieldChange fires", it means that the event is fired on all levels where it is handled. It may be in a table view, in a composite table, in a simple table, and so on, whatever applies and contains a handler code.

Events on Editing Row

Edit Mode

The **DataObjects for .NET** concept of *edit mode* is exactly the same as in ADO.NET. A row is in edit mode after a BeginEdit method call. Calling EndEdit ends the edit mode state. Edit mode is intended for making successive changes to a row without validating the whole row. A single change does not have to satisfy validation conditions. Row validation occurs when the row exits edit mode, on EndEdit. Place such validation code in the BeforeEndEdit event.

Edit mode is used by bound controls: when a row becomes current, it is automatically placed in edit mode (BeginEdit is called implicitly). When the user moves to another row, the row exits edit mode (EndEdit is called implicitly). Bound controls maintain a single row in edit mode at any given time, it is the row corresponding to the current row position of System.Windows.Forms.CurrencyManager. Programmatic calls to BeginEdit / EndEdit allow multiple rows in edit mode simultaneously.

While a row is in edit mode, changes made to its fields since the moment it entered edit mode can be canceled (reversed) by calling CancelEdit. It can also be done through some bound controls, a grid, for example. The row remains in edit mode after CancelEdit, only field changes are canceled.

Events in Edit Mode Transitions

When a bound control or user code calls BeginEdit, **DataObjects for .NET** fires the BeforeBeginEdit. This event can be used to disallow transition to edit mode by throwing an exception, if necessary, although this is not a recommended design pattern. Note that transition to edit mode does not yet mean that the row is being changed. If you want to block changes to a row, the proper place to do it is in the BeforeFirstChange event, if you need to block the whole row, or in BeforeFieldChange, if you want to block a particular field.

After a row has entered edit mode, the AfterBeginEdit event is fired.

Before a row leaves edit mode, the BeforeEndEdit event is called. This event is the most important among the edit mode transition events, because it is the point where data validation code resides in most cases. If you have data validation rules dependent on multiple fields, program these rules in the BeforeEndEdit event. If validation rules are not satisfied, to signal validation failure, throw an exception. That will prevent row from leaving edit mode. Note that in many cases you can use Constraints, expressions specified as table or table view properties, instead of writing validation code in the BeforeEndEdit event. Record-level constraints are tested before the BeforeEndEdit event.

After a row has successfully left edit mode, the AfterEndEdit event is fired.

When the user cancels changes in edit mode (CancelEdit is called or a bound control cancels changes), **DataObjects for .NET** first fires the BeforeCancelEdit edit. Throwing an exception in that event disallows canceling changes. After successfully canceling, reversing the changes, **DataObjects for .NET** fires the AfterCancelEdit event.

Edit mode by itself does not mean that the row is being changed; it only signifies a special state in which changes to row fields do not trigger row validation. When the row is actually changed first time after entering edit mode, the BeforeFirstChange event fires. By throwing an exception in that event, you can disallow changes to the row. After the first change has been successfully performed, the AfterFirstChange event fires. You can use that event, for example, to show some user interface clues indicating that the row has been changed (canceling them in AfterCancelEdit and AfterEndEdit, if necessary). After the AfterFirstChange event has been fired for the first time, successive changes to row fields do not fire the BeforeFirstChange/ AfterFirstChange events.

Events on Modifying Row

The sequence of events invoked when the end user or user code modifies a field is as follows:

1. Field-level constraints are tested. If a constraint is not satisfied (does not evaluate to **True**), an exception is thrown and the change aborted. In a constraint expression, the field being modified returns the new (proposed) value, although the value has not yet been assigned to the field. If you need the old value of the field in an expression, use a special function: *BeforeChange(field)* (see DataObjects for .NET Expressions).

2. The BeforeFieldChange event fires. This event allows you to specify validation logic that cannot be specified in the form of constraint expressions. Throwing an exception in this event aborts the change. When this event is fired, the field value is not yet modified. The new (proposed) values are available in the *NewValue* argument of the event.
3. The new value is assigned to the field.
4. Any calculated fields (Calculation expression) dependent on the modified field acquire their new values immediately after the new value is assigned to the field (before the AfterFieldChange event). In fact, **DataObjects for .NET** does not perform any special processing in order to update calculated fields after changing the value. It simply marks the calculation results that depend on the changed value, internally, as out-of-date. Whenever a calculated value is requested, in code, or by a bound control, for display, the calculation expression is re-evaluated to obtain the up-to-date value. Note that bound controls are not notified of the change until the whole process of handling the field modification is completed. Only after all necessary changes are done, bound controls will be notified of all changes resulting from the field modification, including the calculated expressions that have gone out-of-date. At that point, the bound control will request the new values, so the calculation expressions will be re-evaluated. If, for some reason, you need to force re-evaluation of calculated expressions before that, it can be done calling the Refresh method for a single row or Refresh for a whole rowset.
5. The AfterFieldChange event fires. This event allows you, among other possible actions, to modify other fields/rows/tables depending on this field, for example, update some counters. The new value is already assigned to the field, and the old field value is also available in this event, in its *OldValue* argument. This can be useful, for example, in updating counters, where you may need to add the new value and subtract the old one.
6. At this point, after the AfterFieldChange event is handled, the process of changing a field value is finished. However, that does not mean that **DataObjects for .NET** has finished the processing and relinquished control to the user. The business logic code that handled the events could make other changes to row fields in the process, or it could change other rows or other tables. When this handling is done, **DataObjects for .NET** collects all changed rows, all changes resulting from the field modification, and fires the AfterChanges event for each such row. If the business logic code did not change any fields in rows other than the row where the original field change occurred, then AfterChanges will fire for that row only. If, however, other changes resulted from the field modification, AfterChanges will fire for all changed rows. The AfterChanges event is designed specifically to ensure that this is the "last change", so the developers can put code that relies on the "finality" of changes there. For instance, this event is the best place to put your code calculating some calculated field values (supposedly, for calculations that cannot be specified as simple calculation expressions).
7. The CurrentRowChanged event fires with **ChangeType** set to RowChangedTypeEnum.***FieldsChanged***. Strictly speaking, this event is not a part of business logic, because it is fired in client components, such as C1DataSet, not in data library components. It is used in the client application for GUI purposes, in scenarios like, for example, synchronizing detail data with the master row on every change occurring in the master row.
8. Finally, **DataObjects for .NET** notifies bound controls of all changes that occurred during the processing of field modification, so they can update their display with new data. Notifying bound controls does not occur immediately when you change a field in your business logic code. It is postponed until the end, because a field modification can cause other changes (see above), so it is better, in more senses than one, to notify of all changes at once, when they are final.

Events on Adding Row

Before adding a new row, **DataObjects for .NET** fires the BeforeAddNew event. When this event fires, the new row is not yet created. Throwing an exception in this event aborts the process of creating a new row and adding it to the table.

Once a new row has been created and filled with default values (see the field DefaultValue property), **DataObjects for .NET** fires the AfterAddNew event. That event can be used to fill default values of the fields according to some rules that cannot be specified as simple values of the DefaultValue property. It can also be used to specify the primary key values, see Keys Assigned by Client: New Row Detached and Attached State.

When a new row is created, it is empty, filled with default values, its primary key undefined. A row with an undefined

primary key is in a special transitory state called *detached*. For more details, see Keys Assigned by Client: New Row Detached and Attached State. A row becomes a full-fledged table row (for example, it can be sent for updating the database) only after its primary key is set. There are two main scenarios in setting the new row primary key:

- The key may be set immediately after the row is created, using the AutoIncrement field property, or in the AfterAddNew event. See Adding Rows and Primary Keys for detailed explanation and examples. In this case, the end user sees the row as a normal row with a primary key from the very beginning.
- The key may be left empty until the user sets it manually or it is set programmatically at the user's request, see Adding Rows and Primary Keys.

In both cases, the point in time when the row becomes a "regular", full-fledged row with a defined primary key value (becomes *attached*) is an event that is potentially important to the business logic. Before attaching a row, **DataObjects for .NET** fires the BeforeEndAddNew event. Throwing an exception in this event aborts the process of attaching the row (and aborts EndEdit, if the row is being attached as a result of an EndEdit call). After the row has been successfully attached, **DataObjects for .NET** fires the AfterEndAddNew event.

To inquire about the state of a newly added row, use the RowState property. For a detached newly added row the state is *Detached*. For an attached newly added row the state is *Added*.

Events on Deleting Row

Before deleting a row, **DataObjects for .NET** fires the BeforeDelete event. Throwing an exception in this event aborts deleting the row.

After a row has been successfully deleted, **DataObjects for .NET** fires the AfterDelete event.

Events on Updating Database

All business logic events described in the previous sections are client-side events; they are fired on the client. The update events listed in this section are server-side events, that is to say, they are fired on the server. You do not need to write a different program to support this client-server division. Your business logic acts as a whole, you do not need to divide it, it all resides in your *data library*. The same data library is used both on the client and on the server. **DataObjects for .NET** manages the distinction between client and server and their interaction automatically, transparently to the developer. However, you must be aware that the database update process, and hence the update business logic events, take place on the server, not on the client. Because the data existing on the server is different from that existing on the client, only changed rows and rows related to them exist on the server, see Update in 2-Tier and 3-Tier Configurations. For more details about **DataObjects for .NET** data libraries, their client and server use, see Application Configurations.

Note that this distinction between client and server does not apply to a *direct client* application (see Application Configurations), that is a 2-tier application updating the database directly from the client, an application where C1SchemaDef and C1DataSet components reside on the same design surface. In this case, the client and the server are one and the same application.

Update events are special also in the sense that controlling the order of actions using the ExecutionMode property, as described in Action Order and Execution Mode, is not available for update events. Update events are triggered without an action context stack. They are always handled in the default execution mode *ExecuteImmediately*.

Business logic inheritance levels apply to update events in the same manner as to other **DataObjects for .NET** business logic events. You can handle update events on any suitable level: on a simple table level, composite table level, table view level, and so on. For more information, see How Business Logic Works on Different Levels.

Before updating the database, **DataObjects for .NET** fires the BeforeUpdate event. When the database update is finished, it fires the AfterUpdate event. The AfterUpdate event is fired after both successful and unsuccessful updates.

When updating database rows, **DataObjects for .NET** fires the BeforeUpdateRow and AfterUpdateRow for each row that it changes in the database.

See Updating the Database for detailed explanation of the update process and of the role of the events in it.

Business Methods

In addition to business logic events, **DataObjects for .NET** allows you to define business methods in your data library. These are methods added by the programmer to the RemoteDataService-derived class in the data library.

They can be called from the client (both in 2-tier and 3-tier configuration) using the RemoteDataService property. In a 3-tier configuration, business methods are executed on the server. They can fill data sets with data, update the database and perform any other required operations. In business methods, you can use either the default or pre-created connections and/or transactions. C1DataSet objects can be passed to business method as parameters.

In a 3-tier configuration, a data set passed as a parameter is serialized on the client (C1DataSet implements ISerializable) and deserialized on the server. If a business method returns a data set, it is serialized on the server and deserialized on the client.

In many cases, business methods are not needed, if standard **DataObjects for .NET** Fill and Update methods are enough for performing database operations.

Also, even if you feel that Fill and Update methods do not satisfy all application needs, you can consider customizing them without defining additional business methods. Fill and Update can be customized with appropriate business logic events, and also you can define your own, completely customized Fill and Update by overriding virtual methods Fill / Update in the RemoteDataService-derived class in the data library.

> **Note:** For an example showing how to use business methods to fill data sets and update the database, see the **CustomFillUpdate** sample, which is installed with the **Studio for WinForms** samples.

If your custom database processing is not covered by standard or customized Fill and Update methods, then you can implement it with business methods in the RemoteDataService-derived class in the data library.

If you fill a data set in a business method and need to pass it back to the client, you can simply return the C1DataSet object from the method.

To update the database in business methods, use the C1DataSet. GetChanges and Merge methods.

GetChanges creates a C1DataSet containing only modified simple table rows, so they can be passed to the business method on the server for updating the database. To refresh the original data set on the client after update, return refreshed (retrieved from the database after update) rows from the business method and apply the Merge method.

Using Typed Data Objects

For each table and table view, **DataObjects for .NET** generates a class (*data object class*) representing its row. For example, **ProductsRow** is an object (business object, data object) where each field has a corresponding property (**ProductsRow.UnitPrice**, **ProductsRow.UnitsInStock**, and so on) and each relation has a corresponding method (**Products.GetOrder_DetailsRows**, and so on) allowing you to obtain child rows and the parent row). Using these classes, you can write your business logic code in a convenient, type-safe way, and benefit from Visual Studio code completion features giving you the lists of properties and methods to choose from.

To use a data object class in your code, call the static **Obj** method implemented in every data object class. It obtains a business object given a C1DataRow object as its argument. For example, the following code obtains a **ProductsRow** business object from a C1DataRow:

To write code in Visual Basic

VB

```
Dim dataRow As C1.Data.C1DataRow
Dim product As DataLibrary.DataObjects.DataSet.ProductsRow
```

```
product = DataLibrary.DataObjects.DataSet.ProductsRow.Obj(dataRow)
```

To write code in C#

C#

```
C1.Data.C1DataRow dataRow;
DataLibrary.DataObjects.DataSet.ProductsRow product;
product = DataLibrary.DataObjects.DataSet.ProductsRow.Obj(dataRow)
```

For a complete example see the **WorkingWithData** sample, which is installed with the **Studio for WinForms** samples.

Data object classes are hosted in the data library. They are automatically regenerated by **DataObjects for .NET** at design time every time you save a schema in the **Schema Designer**. They are hosted in a special assembly (called *data objects assembly*)

```
<data_library_project_name>.DataObjects.dll
```

Data object classes belong to the namespace

```
namespace <data_library_namespace>.DataObjects
```

where <data_library_namespace> is the namespace containing the C1SchemaDef component in the data library.

Since there is a data object class for each table and for each table view, a naming collision can occur for data object classes if a table view has the same name as the table on which it is based. Although all naming collisions are automatically resolved by adding "_table" and "_tableView" suffixes to the names, it is better to avoid such collisions because they result in ungainly class names. To avoid naming collisions between table and table view classes, use the DataObjectsAssemblyFlags property. Setting DataObjectsAssemblyFlags.**DataSetNamespaces** flag in that property (it is set by default for new projects) creates a separate namespace for each data set, and table view classes belong to that namespace. For example, **Northwind.DataObjects.CustOrders.EmployeesRow** for table view *Employees* in data set *CustOrders*. Table classes still belong to the root namespace: **Northwind.DataObjects.EmployeesRow** for table *Employees*.

A data library project contains a reference to its data objects assembly. When you create a new data library project using the **C1DataObjects Project Wizard**, the wizard adds a reference in the data object assembly to your data library project References. The data library assembly created by the wizard (initially empty) resides in the *obj* sub-directory of your project directory.

By default, **Schema Designer** regenerates the data objects assembly every time you save a schema. If you do not want this to happen (for example, if you do not need the data object assembly in your application), uncheck the "Recreate DataObjects DLL on saving schema in Schema Designer" check box in the **Schema|Options** dialog box of the **Schema Designer**. If you leave it checked, you can optionally specify the location of the data objects assembly and version information in that assembly. Note that if you change the data objects assembly location, you have to change the reference to that assembly in your project accordingly.

Data object classes can only be used with data libraries; they cannot be used in "direct client applications" that do not use data libraries; for more information, see Application Configurations.

A data object class is generated for each table in the schema (both for simple and composite tables) and for each table view in every data set in the schema.

A data object class (for example, **OrdersRow**) contains the following members:

- <field_name> – a property for each table/table view field. Example: **OrdersRow.ShipAddress**.
- Is_<field_name>Null() – a Boolean method without parameters returning **True** if the value of the field is **Null**.
- Set_<field_name>Null() – a method without parameters setting the field value to **Null**.
- Get_<parent_name>Row – a method without parameters returning the single parent row (or null) with regard to a relation in which this table/table view is the child. For inverse (many-to-one) relations, it returns the single

child row.
- Get_<parent_name>Rows – a method without parameters returning the array of child rows with regard to a relation in which this table/table view is the parent. For inverse (many-to-one) relations, it returns the array of parent rows.

Using the DataObjectsAssemblyFlags property, setting the DataObjectsAssemblyFlags.**StaticNameFields** flag (it is set by default for new projects), adds more members to the data object classes. Those are static members returning constant names: field, table, table view and relation names, so that they can be used instead of constant strings in code thus making it compile-time safe (errors detected at compile time), for example:

- Northwind.DataObjects.EmployeesRow.FieldNameEmployeeID (field name returning "EmployeeID" for field EmployeeID of table Employees)
- Northwind.DataObjects.EmployeesRow.TableName (table name returning "Employees", for table Employees)
- Northwind.DataObjects.EmployeesRow.RelationNameEmployees__Orders (relation name returning "Employees - Orders", for simple relation "Employees - Orders")
- Northwind.DataObjects.CustOrders.EmployeesRow.TableViewName (table view name returning "Employees", for table view Employees in data set CustOrders)
- Northwind.DataObjects.CustOrders.EmployeesRow.FieldNameEmployeeID (field name returning "EmployeeID" for field EmployeeID of table Employees)
- Northwind.DataObjects.CustOrders.CustOrdersRow.RelationNameCO__ODP (relation name returning "CO - ODP", for view relation "CO - ODP" between table views CustOrders and OrderDetailsProducts in data set CustOrders)

Action Order and Execution Mode

When handling an event in business logic code, you can perform an action that triggers another business logic event, that in turn is handled and triggers yet another event, and so on. For example, handling a field change event, you can set another field, or perform any other possible action. In most data frameworks, such actions are either blocked or can only be handled in a straightforward way creating uncontrolled recursion. Uncontrolled event triggering leads to complicated situations often creating unintended effects and side effects. Every database developer can recall situations where a harmless code intended to, say, commit a record once certain fields are entered by the end user, either does not work or produces unexpected and often disastrous results.

DataObjects for .NET solves this problem by allowing the developer to exercise full control over the order of actions performed by business logic code. Various actions, including method calls and field value assignments, can be specified to take place either immediately or after **DataObjects for .NET** has completed performing previous actions that are already in the queue, and completed handling the events that are being handled.

When the end user initiates an action leading to a business logic event (for example, sets a field value, or deletes a row), **DataObjects for .NET** starts executing it by creating an *action context stack*. The initial state of this stack is one level with the requested action contained in it. If no actions have been requested by the business logic code while this action was handled, the stack is destroyed. If, however, an action (that itself can trigger a business logic event), is requested while handling this action, the requested action can be processed by **DataObjects for .NET** in two different ways:

- **Immediate** (default)**: DataObjects for .NET** adds a new level to the action context stack and starts handling this new action. The original action handling is waiting until the new action is handled. This is the "straightforward", recursive event handling mentioned above.
- **Deferred:** Alternatively, **DataObjects for .NET** can add the requested action to the action queue, to be processed after all actions already in the queue are finished. In this case, **DataObjects for .NET** does not do anything with the requested action, it just adds it to the queue and immediately returns, continues to handle the event. The requested action will be processed before **DataObjects for .NET** relinquishes control, "in due course", when the event code has finished executing and all the actions that got to the queue earlier have been processed.

To control the action order, use the ExecutionMode property. Its two possible values, **Immediate** (default) and **Deferred**, specify the two modes described above.

ExecutionMode is a read-only property. You never need to set execution mode permanently. You only need to set it for the duration of a certain fragment of code, after which it must be restored to its previous value. Since changing execution mode can inadvertently break your code, we strongly recommend using try-finally blocks to ensure that previous execution mode is always restored.

To set execution mode, call the PushExecutionMode method, passing it the new mode as a parameter. To restore the previous execution mode, call PopExecutionMode.

For example, the following code (taken from Adding Rows and Primary Keys) sets the primary key field in a newly added row immediately after the row has been created.

To write code in Visual Basic

VB

```
Private Sub table_Customers_AfterAddNew(ByVal sender As Object, ByVal e As
C1.Data.RowChangeEventArgs) Handles TableLogic1.AfterAddNew
    e.DataTable.DataSet.PushExecutionMode _
         (C1.Data.ExecutionModeEnum.Deferred)
    'The following field value assignment and a method call
    'will actually execute only after all other actions handling
    'the row addition will be finished
    e.Row("CustomerID") = TextBox1.Text
    e.Row.EndEdit()
    e.DataTable.DataSet.PopExecutionMode()
End Sub
```

To write code in C#

C#

```
private void table_Customers_AfterAddNew(object sender, C1.Data.RowChangeEventArgs e)
{
    e.DataTable.DataSet.PushExecutionMode(ExecutionModeEnum.Deferred);
    // The following field value assignment and a method call
    // will actually execute only after all other actions handling
    // the row addition will be finished
    e.Row["CustomerID"] = textBox1.Text;
    e.Row.EndEdit();
    e.DataTable.DataSet.PopExecutionMode();
}
```

Although this code would probably work without changing execution mode, doing so would be unsafe, since it would trigger **EndEdit** before **AddNew** is completely finished (for example, a grid or another bound control could be not yet notified at that time that a row has been added).

What we really want in cases like this ("after this, do that") is to emulate end user actions, as if the end user typed the key value after adding a new row, which is essentially what **DataObjects for .NET** is doing in the *Deferred* execution mode.

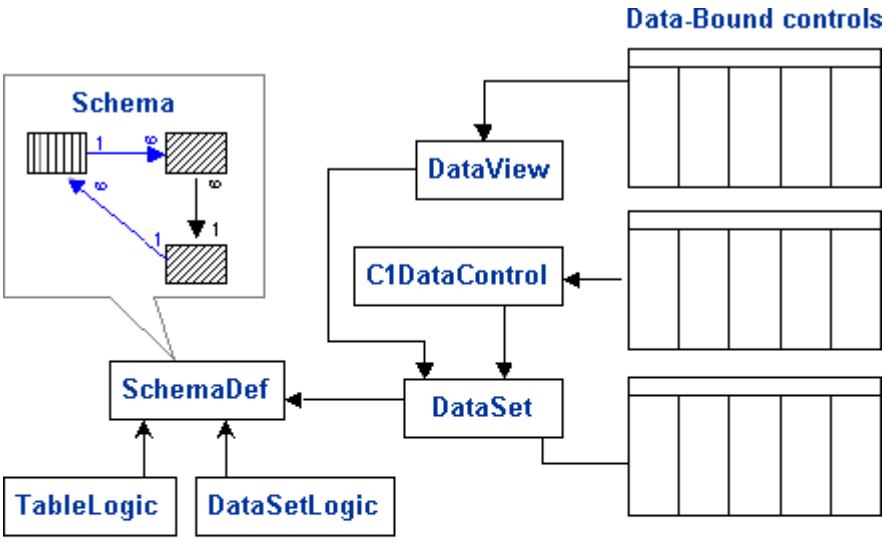
Application Configurations

Encapsulating business logic in a *data library*, **DataObjects for .NET** makes it possible to use business objects,

developed once, in a variety of application deployment scenarios. In particular, **DataObjects for .NET** completely automates the task of developing distributed 3-tier applications, with no special server-based code necessary. **DataObjects for .NET** is the only tool for .NET that allows developers to create applications in an easy, uniform way, so that they can be deployed in either 2-tier or 3-tier configurations without changing the application code. In essence, creating fully scalable Web-based distributed applications becomes a matter of point-and-click, as easy as creating a simple desktop application used to be in Visual Basic.

Direct Client

Direct client is the simplest configuration where all application components, with the exception of the database server, reside on a single machine and in a single project. It is distinguished from other configurations in that it does not use a *data library*. Business logic code (components C1SchemaDef, C1TableLogic, C1DataSetLogic) resides in the same project, on the same design surface with the components using the data (C1DataSet components, C1DataView and C1DataTableSource, data-bound GUI controls). There is no separation between the server and the client parts – they are one and the same application.



Business logic code

```
Private Sub table_Order_Details_BeforeFieldChange(ByVal sender As Object,
ByVal e As C1.Data.FieldChangeEventArgs) Handles
table_Order_Details.BeforeFieldChange
   If e.Field.Name = "Quantity" Then
      If(CShort(e.NewValue < 1) Then
         e.NewValue = CShort(1)
   EndIf
```

In a direct client application, a C1DataSet component resides on the same design surface with the C1SchemaDef component hosting the schema. A C1DataSet component must be attached directly to the C1SchemaDef component by setting its C1SchemaDef property (whereas in other configurations, a C1DataSet is attached to the data library instead, by setting its DataLibrary property). Data-bound GUI controls, such as grids, are attached to a C1DataSet either directly or indirectly (through C1DataView or C1DataTableSource), see How to Access Table View Data and How to Access Table Data.

Direct client configuration is recommended for desktop applications and small 2-tier client-server applications that do not merit encapsulating business logic code in a separate project (data library). Developing a direct client application is very similar to traditional database application development using data controls to connect to the database and bound controls to display the data. It must be understood that a direct client application cannot be easily transformed to a data library; it will need to be re-created (although it may be a simple matter of copy/paste). So, if you have

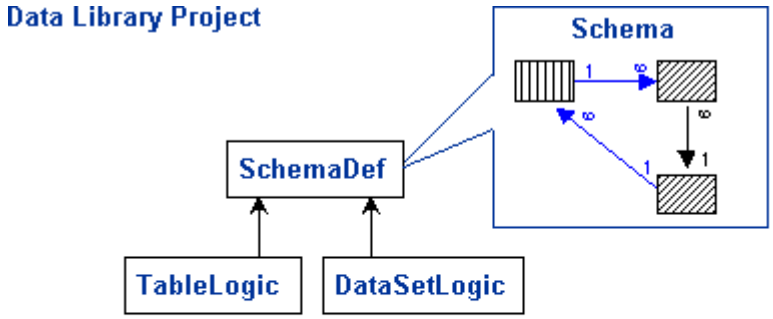
scalability in mind, it may be worth considering a data library from the start.

Data Library

DataObjects for .NET enables developers to define business objects in a separate assembly, *data library*, so it can be used by multiple applications. Although this is not mandatory, a special team of "data-oriented" developers can be assigned to the task of creating business object projects (data libraries) and another team of "GUI-oriented" developers to creating client applications using data libraries.

A data library project contains the following:

- A schema stored in a C1SchemaDef component, one such component per data library project, see Schema Objects.
- Business logic code, handling the events of business logic components, C1TableLogic and C1DataSetLogic; for more information, see Business Logic.




```
Private Sub table_Order_Details_BeforeFieldChange(ByVal sender As Object, ByVal e As
C1.Data.FieldChangeEventArgs) Handles table_Order_Details.BeforeFieldChange
    If e.Field.Name = "Quantity" Then
        If(CShort(e.NewValue) < 1) Then
            e.NewValue = CShort(1)
    End If
```

A data library can be used in any project by simply referencing the library in the project and using a **DataObjects for .NET** C1DataSet component to connect to the library. All database access and business logic code is encapsulated in the library, so it can be created and maintained independently of client applications.

Creating a Data Library Project

Data library projects are created using the **C1DataObjects Data Project Wizard**. Select **File | New Project** in the Visual Studio menu, and in the **New Project** dialog box, under **Project types**, choose either **Visual Basic** or **Visual C#**, according to your language preference and select **ComponentOne Data Library** in the right pane.

The main file of the resulting data library project is a component class hosting the schema and business logic. The schema is stored in the C1SchemaDef component. To edit the schema, use the **Schema Designer**; for more information, see Schema Objects. To open the designer, select **Schema Designer** from the C1SchemaDef component's context menu.

To attach business logic code to a schema object (table or table view) use *business logic components*. Business logic components (components C1TableLogic and C1DataSetLogic) have events in which you can write code responding to various occurrences in data objects. Business logic components reside on the same design surface (component class) with the C1SchemaDef component (in large projects, it is also possible to distribute business logic code over multiple files). Only one business logic component is allowed for each schema object: a single (or none) C1TableLogic component for each table and a single (or none) C1DataSetLogic component for each data set in the schema.

To create business logic components, right-click the C1SchemaDef component and select **Create Business Logic Components**from the context menu that appears. This will create a business logic component for each table and each

data set in the schema. Alternatively, you can create business logic components manually, from the Toolbox, but in that case you need to set their properties: SchemaComponent = *schemaDef* to connect them to the schema component, as well as Table and DataSetDef to specify which schema object each of them represents.

In large data library projects, it may be desirable to use multiple source code files to write business logic code. It can be done using a special C1SchemaRef component. A C1SchemaRef component's only purpose is to represent the C1SchemaDef component holding the library's schema when it resides in a different file in the project. Add a new component class file to the project, put a C1SchemaRef component on the design surface and connect it to the C1SchemaDef component selecting its name in the SchemaDef property combo box. Note that working with multiple files using C1SchemaRef, the file with the C1SchemaDef component must be open. Once a C1SchemaRef component is set up, you can add business logic components, C1TableLogic and C1DataSetLogic to the component class file, dropping them from the Toolbox. For each business logic component, set its SchemaComponent property to the C1SchemaDef component and select the table (Table) or the data set (DataSetDef) represented by the business logic component.

You can attach event code to business logic components either by selecting the components on the designer surface, or using the **Business Logic Events** tool window (select **Business Logic Events** from the C1SchemaDef or C1SchemaRef context menu to open the tool window). The **Business Logic Events** tool window shows the list of all tables and data sets. When you select a table in the tool window, the table's business logic events appear in the Properties window (in Visual C#, when the Events radio button is selected in the Properties window; in Visual Basic use the Method Name combo box in the code editor).

Using a Data Library

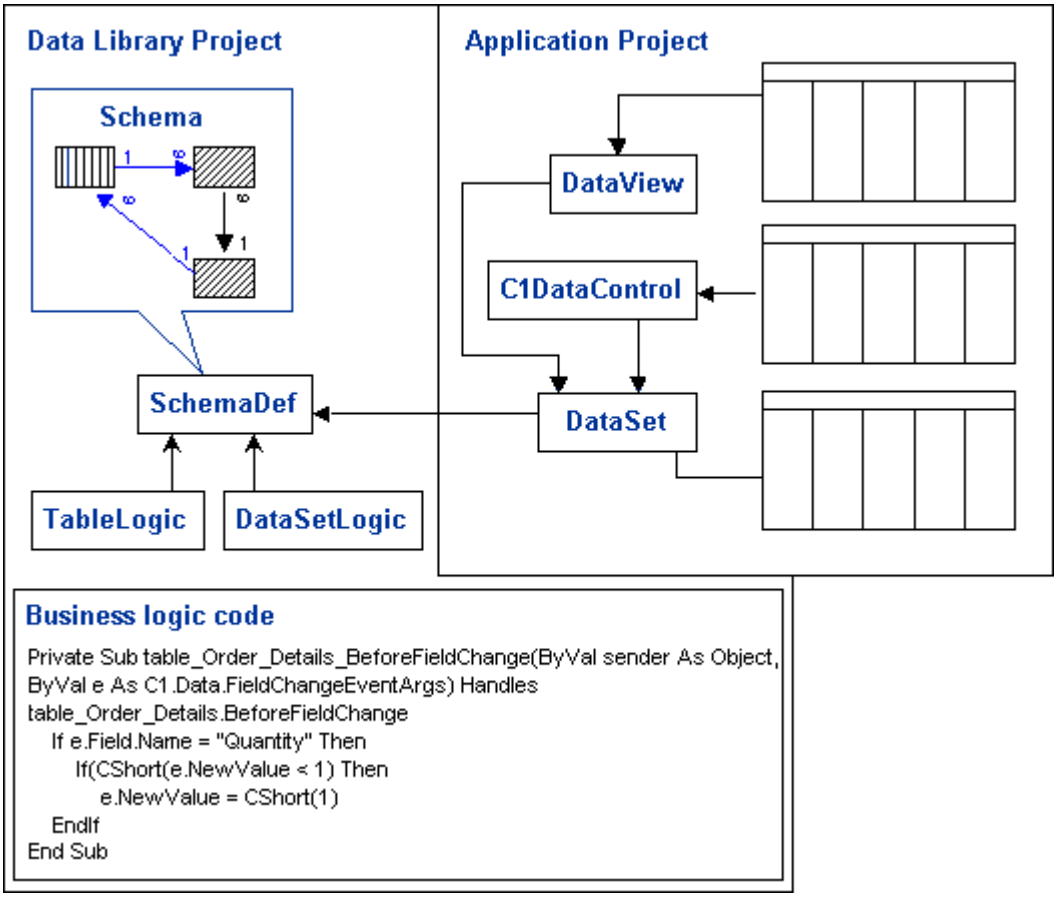
To use a data library in a client application:

- Compile the data library project to produce the data library assembly.
- Add a reference to the data library assembly to the client application project's References.
- To bind to the data on a client application form, create a C1DataSet component on the form and type the data library name (without the .dll extension) in the DataLibraryUrl property. After that, the list of all data set definitions in the schema becomes available for selection in the **DataSetDef** property.
- Bind data-aware GUI controls to the C1DataSet either directly or indirectly (through C1DataView or C1DataTableSource), see How to Access Table View Data and How to Access Table Data.

2-Tier Application

If you set up a client application exactly as described in Using a Data Library, it will work as a 2-tier application. Note that to make it 3-tier, only one additional step is necessary: setting the DataLibraryUrl property. For more information, see 3-Tier Application.

In a 2-tier application deployment configuration, a client application and the data library reside on the same machine, and there are no other computers involved except possibly a database server.



C1DataSet components belonging to the client application interact with the data library. They request data from the data library on Fill and send modified data to the data library on Update. Responding to these requests, the data library sends data requests to the database server, which can reside on the same or on different computer, according to the ConnectionString specified in the schema.

The client application fires business logic events in the data library. For example, when the end user modifies a field value in a grid, the C1DataSet component (to which the grid is bound with its **DataSource**/**DataMember** properties) calls the data library, and the BeforeFieldChange event is fired, handled by business logic code inside the data library.

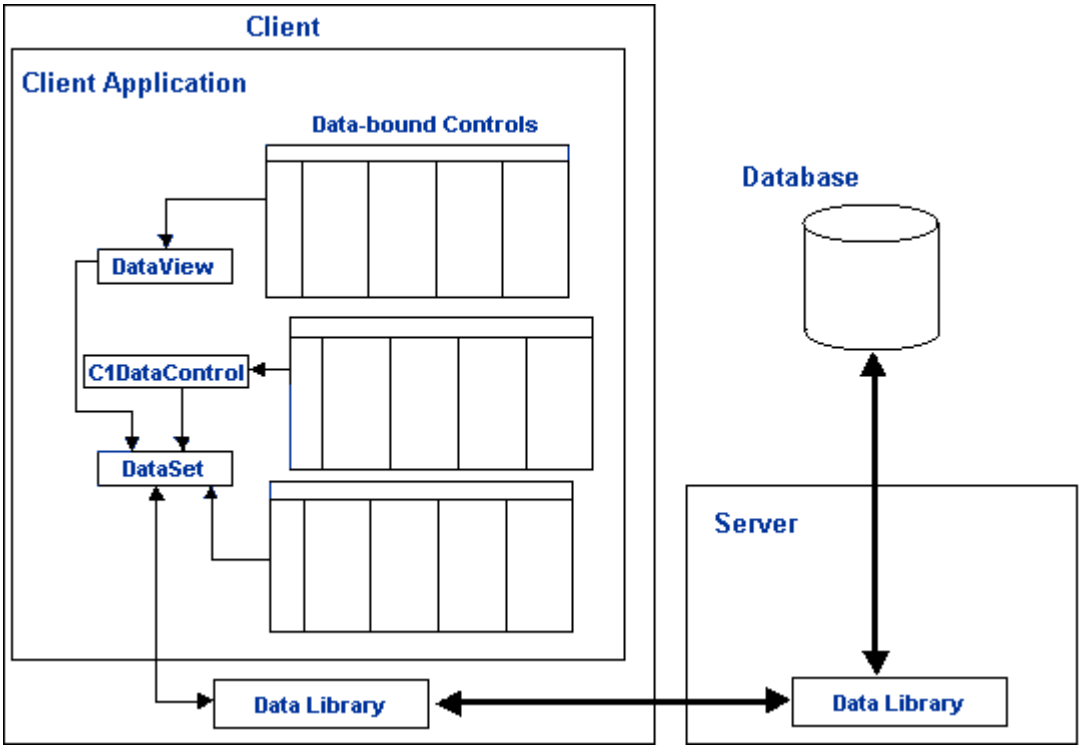
3-Tier Application

In contrast with standard ADO.NET and other frameworks, you can turn a **DataObjects for .NET** application into a distributed 3-tier application without additional coding. With **DataObjects for .NET**, it is a simple matter of deployment. **DataObjects for .NET** uses the following mechanism to enable 3-tier configuration:

You use the same data library assembly in both 2-tier and 3-tier configurations. The only difference is that when your application wants to use the data library assembly in a 3-tier configuration, it signals this intent by setting the DataLibraryUrl property to a server location, a URL where the same data library assembly is installed and ready for invocation on a remote server. For example:

```
dataSet1.DataLibraryURL: http://www.mycompany.com/mydatalibrary/ThreeTierServer.soap
```

Your client machine already has the data library assembly, since it is used by your client application, included in its project's References. This is the data library your client application talks to.



When your application requests data (Fill), the request first goes to the client data library. Knowing the URL, the client data library invokes the data library on the server (server data library) passing it the request for data. Note that both client and server run the same version of the data library assembly that is ensured by Microsoft. NET remoting support. The server data library retrieves data and sends it to the client, where the client data library receives it and exposes it to the client application.

When your application updates the database, sends modified data to the database, the client data library takes the modified data, packages it and sends to the server data library for update. The server data library reconstructs the data set on its end, so that business logic code can work properly, see Update in 2-Tier and 3-Tier Configurations, and updates the database. Then it sends the refreshed data retrieved from the database back to the client.

So, the process involves two running instances of a data library: one on the client and one on the server. Developing a data library, you do not have to be concerned with interaction between these two instances. You do not need to write two different programs for the server and for the client, because the server and the client use the same assembly built from the same data library project. **DataObjects for .NET** handles the interaction automatically and transparently for the developer.

If you need to know, in data library event code, which instance is currently running, the client or the server one, this information can be obtained from the RunsAt property.

Most business logic events are handled on the client, by the client data library instance. For example, when the end user modifies a field value in a grid, the C1DataSet component (to which the grid is bound with its **DataSource**/**DataMember** properties) calls the client data library, and the BeforeFieldChange event is fired and handled by business logic code inside the client data library. However, update business logic events, BeforeUpdate, BeforeUpdateRow, AfterUpdateRow and AfterUpdate are handled by the server data library, see Update in 2-Tier and 3-Tier Configurations.

Configuring the Data Library on the Server

To deploy your application as a 3-tier distributed application, you need to make your data library on the server available for invocation by clients, to publish it at the DataLibraryUrl location. You can use any remote invocation mechanism supported by .NET Remoting. The most popular one is the *Web service*, where the published assembly is placed under IIS (Internet Information Server) control and made available by creating an IIS virtual directory. In this case, remoting uses an HTTP channel and the SOAP protocol. Deploying your data library under IIS, you use a configuration file. A configuration file *web.config*for IIS deployment can look, for example, as follows:

```
<configuration>
```

```
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton" type="Northwind.RemoteService, Northwind" objectUri="ThreeTierServer.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Northwind is the name of your data library assembly.

Northwind.RemoteService is the full name of a class in your data library derived from C1.Data.RemoteDataService. The **C1DataObjects Data Project Wizard** generates this class automatically. This class is necessary for **DataObjects for .NET** remoting support. By default, it is empty, just derives from C1.Data.RemoteDataService. If necessary, it can be used for configuring **DataObjects for .NET** remoting on the client, see below.

ThreeTierServer.soap is the URI (arbitrary string) used to form the DataLibraryUrl string, for example, http://www.mycompany.com/mydatalibrary/ThreeTierServer.soap, where *mydatalibrary* is the IIS virtual directory containing the data library assembly

Although IIS deployment is the most common, it is by no means the only option. You can host the server instance of your data library in any server host available for remote invocation via .NET Remoting. For example, Tutorial 3 shows how to host it in a Windows application running on the server. Instead of HTPP channels used by Web services, Tutorial 3 uses a TCP/IP channel. The flexibility of **DataObjects for .NET** remoting mechanism is based on the fact that there is nothing **DataObjects for .NET**-specific in hosting and configuring a data library on the server; it is all based on the standard .NET Remoting. Use whatever server deployment option and configuration you find necessary, as long as it exposes an object derived from C1.Data.RemoteDataService (Northwind.RemoteService in the example above) as a well-known object type in Singleton mode (see .NET Remoting documentation for a detailed description of well-known objects and activation modes). To register an object type, you can call the RemotingConfiguration.RegisterWellKnownServiceType method (namespace System.Runtime.Remoting), or use a configuration file (in IIS deployment configuration file is the only option).

Normally, **DataObjects for .NET** remoting does not need configuring on the client. However, if you use a custom remoting configuration on the server that requires configuring the client, you can do that using the **ConfigureClient** method of the C1.Data.RemoteDataService class. As noted above, in order to allow 3-tier deployment, your data assembly needs to derive a class from RemoteDataService. You can override the virtual **ConfigureClient** method in that class. The method is called once for the client data library, to give you an opportunity to execute whatever code you need to configure remoting on the client. Another RemoteDataService method that can be used for configuring client is **ConfigureProxyObject** that is called every time the client creates a proxy object for calling the server. Overriding the **ConfigureProxyObject** virtual method you can configure remoting on a per-call basis, for every proxy object created, if that is necessary.

Security Considerations

Authentication in IIS Deployment

By default, **DataObjects for .NET** client uses HTTP channels for its communications with the server. HTTP channels have built-in authentication and authorization support in .NET remoting. If you deploy your server data library in IIS, you can rely on IIS and .NET remoting security support. You can configure your server data library in IIS for whatever authentication methods you prefer, including Integrated Windows Authentication (recommended), Basic authentication, anonymous access, and so on. You can also use Secure Sockets Layer (SSL) to encrypt all messages by using the "https://" protocol in your DataLibraryUrl. Summarizing, you do not need to write special code for security authorization if you use default configuration, which is deploying server data library in IIS. There is only one programmatic feature you may need in this case; this is how to specify user credentials (domain, username, password):

By default, the **DataObjects for .NET** remoting channel uses the default user credentials (domain, username, password) when it calls the server. Sometimes you may need to make calls on behalf of a specific user. For example, you may use an authentication scheme on the server that is different from one used on the client, and your client application may ask the user for credentials to pass to the server. In this case you will need to tell **DataObjects for .NET** to use these credentials communicating with the server. This can be done using the Credentials property of a C1DataSet object, for example:

To write code in Visual Basic

VB

```
Dim Credentials As New System.Net.NetworkCredential("myusername", "mypassword",
"mydomain")
C1DataSet1.Credentials = Credentials
```

To write code in C#

```
C#
System.Net.NetworkCredential credentials = new
System.Net.NetworkCredential("myusername", "mypassword", "mydomain");
c1DataSet1.Credentials = credentials
```

If the Credentials property is set, every call from the C1DataSet component to the server, filling the component with data or sending its updates to the database, will use the supplied credentials.

Authentication with Custom Channels

If you have special security needs, such as a custom authorization mechanism or a custom transport encryption, you can customize the channels as described in Configuring the Data Library on the Server. You can use all the power and flexibility of .NET Remoting to add your own security support and other features to the remoting channel. If you use custom channels, take into account that only HTTP channels have built-in authorization support in .NET remoting, TCP/IP channels do not have such support out of the box.

Database Access Authorization and Connection Strings

Another aspect of application security with **DataObjects for .NET** is database access authorization. Security information, such as username and password for database access is included in ConnectionString in the schema, and the schema is included in the client application. There are two options of securing this information:

- You can use Windows Integrated Security, in which case security information is not stored in the connection string, current user credentials are used for database access authorization.
- If you have to specify username and password in the connection string, do not store them in ConnectionString at design time. Use the **User name** and **Password** edit boxes in the **Schema Designer** to send credentials to the database at design time. At run time, add **User ID** and **Password** attributes to the ConnectionString in code, in the CreateSchema event that is fired before any database access is performed on the server.

You may also need to change the ConnectionString on the server for purposes other than security. For example, you can route calls to different database servers. You can do that in the same way, using the CreateSchema event on the server to set the ConnectionString property of Connection objects in your schema. This method changes ConnectionString permanently, for all calls from all users.

If you need to set ConnectionString on a per-call basis, you can do it overriding virtual methods in the RemoteDataService-derived class implemented in your data library, see Configuring the Data Library on the Server. Override virtual methods **GetData** and **Update**, which are the methods that are called by the client, to set up connections for the duration of a call. The RemoteDataService class has a Connections property holding a ConnectionCollection which is empty by default. If you add a Connection object to this collection and set its Name to the name of a schema connection, that Connection object will be used instead of the schema connection. Since a new RemoteDataService object is created for each call, this connection substitution works on a per-call basis.

Virtual Mode Dealing with Large Datasets

With an innovative *virtual mode* technology, **DataObjects for .NET** allows you to use large datasets, unlimited in size, in .NET WinForms applications. This feature is not supported in Visual Studio and ADO.NET without **DataObjects for .NET**.

Understanding Large Data Sets

Visual Studio and its underlying data engine, ADO.NET support only one mode: disconnected access to data. An ADO.NET data set is always pre-fetched in its entirety from the server to the client. Since it is often not practical to

pre-fetch large data sets over the wire, this creates two serious problems:

- Even distributed Web applications often include large data sets, such as, for example, a list of available products and other data originated in database tables with thousands records and more. It is very inefficient and in many cases impossible to transfer such data from server to the client in its entirety. This forces developers to produce makeshift solutions (such as asking the user to enter a few initial letters of the product name before showing the list of products) severely reducing the quality of end user experience, application performance and scalability.
- The absence of large data set support makes it effectively impossible to develop classic client-server and desktop database applications in Visual Studio without **DataObjects for .NET**. A popular belief is that you only need large data sets if you have not designed your application correctly. This is a misconception. The real problem is the lack of tools supporting the right data access modes. Another popular misconception is that disconnected model (also referred to, somewhat loosely, as "Web application", "3-tier application", and so on) necessarily means pre-fetching all the data from the server to the client at once, that any other approach is necessarily the old "live connection per user" that is not scalable.

DataObjects for .NET fills this gap, offering a solution to the problem of large data sets. It gives you the tool to achieve the best of both worlds, to have data access that is both disconnected (no live connection is maintained on the server for particular users) and therefore scalable, and at the same time, unlimited in data size.

To use a large data set in **DataObjects for .NET** is as easy as to set a property, DataAccessMode (you also need to use C1DataTableSource as your data source if you need to bind GUI controls in virtual mode, see Using C1DataTableSource and Bound Controls).

Table Views in Virtual Mode

One or more table views in a schema can be specified to function in virtual mode by setting their DataAccessMode property to **Virtual**, **VirtualUnlimited**, or **VirtualAutomatic**. We will call them *virtual table views*. For now, we will consider only **Virtual** mode. The other two, **VirtualUnlimited** and **VirtualAutomatic**, are variations that will be discussed a little later, in Asynchronous Fetch Modes.

A virtual rowset can contain an unlimited number of rows. For example, Tutorial 4 demonstrates a rowset containing 2.7 million rows that is available on the client with limited memory consumption and startup time of about three seconds.

When data is fetched from the database, a virtual table view fetches only an initial chunk (*segment*) of data. It can fetch other data segments from the server later, on its own accord, transparently to the user. Segment fetch is triggered by the end user repositioning to a row close to the end of the current segment, see When a Virtual Table View Fetches Data Segments.

At any given time, a virtual table view contains and exposes to the user a single contiguous segment of data, the *current segment*. The size of a data segment is specified by the VirtualSegmentSize property (default: 400 rows). The data exposed to the user in the C1DataTable object and in GUI controls bound to the virtual table view, is exactly the current segment, other segments (that may be fetched from the server, see When a Virtual Table View Fetches Data Segments) are kept internally and not exposed.

A virtual table view also maintains a segment cache containing a limited number of segments (controlled by the VirtualSegmentCount property, default: 4). The cache is inaccessible to the user. It is used internally for optimization, to avoid roundtrips to the server when possible.

Although a virtual table view exposes only one segment of data at a time, it automatically and transparently maintains continuity. So, for example, if the end user presses thePageDown key in a bound grid, or uses the MoveNext method to go beyond the current segment bounds, **DataObjects for .NET** will automatically adjust the current segment, fetching data from the server, if necessary, so the user will not notice the segment change. Fetching data and changing the current segment is done automatically, so the user perceives it as a continuous large rowset. Moreover, **DataObjects for .NET** uses a background asynchronous fetch technique to make fetching data from the server as much imperceptible to the user as possible, see When a Virtual Table View Fetches Data Segments.

If you navigate through a virtual table view using a bound grid, or programmatically using MoveNext and other methods of C1DataTableSource, the virtual rowset appears as a large continuous rowset. But it contains only one segment at a time on the client, so, if you use the C1DataTable object to access its data, you will see only limited number of rows.

Virtual mode table views fully support data modification. Modified rows are always visible, always kept in the current segment, and always accessible through the C1DataTable object (until Update, AcceptChanges or RejectChanges) – even if they were originally fetched in other segments.

When a Virtual Table View Fetches Data Segments

This section describes optimization techniques used in **DataObjects for .NET** to make fetching data from the server transparent and largely imperceptible to the user. It has no effect on the formal functionality, so you can skip it if you are not interested in this description.

When the end user navigates through the current segment, they usually position on rows belonging to the current segment. In fact, it is impossible to position on any other row, except by using a C1DataTableSource that has special methods for navigating the whole virtual rowset, or by using the scrollbar of a bound grid control to position to the end or the beginning of the rowset, if it is not covered by the current segment.

When the user positions on a row that is out of bounds of the current segment (this can be done only with C1DataTableSource or with a bound grid going to the end of the rowset), **DataObjects for .NET** looks for the required segment in the cache and, if not found, fetches it from the server. After that, the new segment becomes current.

In order to optimize segment fetch performance, **DataObjects for .NET** also performs a "preemptive" asynchronous fetch when the user positions to a row that is inside the current segment but "dangerously" close to its boundary (the closeness threshold determined by the VirtualSyncThreshold and VirtualAsyncThreshold properties):

- If it is "very close" to the current segment boundary (determined by the VirtualSyncThreshold property), **DataObjects for .NET** fetches the required segment (one that has the current row in the middle rather than near the end) immediately, suspending all other activities (synchronously). When the segment is fetched, it becomes current.
- If it is "not so close" (determined by the VirtualAsyncThreshold property), **DataObjects for .NET** starts the fetch process in a separate thread, so it is performed in the background, without interrupting other activities (asynchronously). When the segment is fetched, it is placed in the cache, but not made current. The segment is prepared in case it will be needed when the user moves closer to the current segment boundary.
- These techniques provide for the following transparency and optimization behaviors:
- Database fetch and segmentation are transparent to the user. The rowset is presented to the user in its entirety, appears to contain all rows regardless of whether they belong to the current segment or not. Any part of it, including the end of the rowset and any given row specified by a primary key value (see C1DataTableSource Seek) is available to the user.
- Database fetches are optimized with a heuristic "preemptive" background fetch technique. If the end user navigates through the rowset slowly enough to allow the optimization to work, segment changes occur without delays, as if the data were already on the client. A jump to a specified position in the rowset (for example, using the C1DataTableSource Seek method) occurs with a delay necessary to fetch the data from the server, but that delay only occurs if the specified position is far enough from any cached segment.

Virtual Mode in Distributed 3-Tier Applications

DataObjects for .NET fully supports virtual mode in distributed 3-tier applications. See Application Configurations for the description of **DataObjects for .NET** support of distributed 3-tier applications. To create a 3-tier application in **DataObjects for .NET** you do not need to write any additional code. It is just a matter of changing a property and deploying your data library on a server. Likewise, you do not need any special adjustments or coding to make virtual mode work in a 3-tier application. **DataObjects for .NET** virtual mode is always 3-tier-ready.

DataObjects for .NET does not maintain a live database connection on the server for the lifetime of a virtual table view on a client. The server uses the connection only to serve client requests for data, at which point it fetched a limited amount of rows (a segment), sends the data to the client and forgets all about this client request, disposes of the query result and closes the database connection (due to OLE DB connection pool optimization, the connection may remain open for some time to allow its effective reuse in subsequent requests). So**, DataObjects for .NET** uses a stateless, scalable server technology, the same as ADO.NET, both in regular mode (DataAccessMode = *Static*) and in virtual mode.

Combining its innovative virtual mode and 3-tier technologies, **DataObjects for .NET** makes creating sophisticated, fully scalable Web-based distributed 3-tier applications a matter of point-and-click, as easy as creating a simple desktop application always used to be in Visual Basic. In fact, the developer of an application can be totally unaware of all these concepts and work as in classic client-server (or even desktop) environment. **DataObjects for .NET** makes it possible to take such application and turn it into a distributed one by a simple change of a few property settings. This is the real scalability of an application development framework.

Without full understanding of the trade-off between scalability and large data sets, a skeptic, when critiquing **DataObjects for .NET** virtual mode, can say: "OK, but that is yesterday's technology. For this type of virtual mode you need a permanent live database connection on the server for each user, so it is necessarily unscalable, it is classic client-server (2-tier) as opposed to the latest and greatest disconnected mode, which is distributed, Web-based, 3-tier, and so on and so forth". This argument is wrong and is a common misconception. These are two entirely different things:

a. One can have a live connection on the server for each user, as in classic client-server (non-scalable, bad) as opposed to a stateless server that connects on demand, as in ADO.NET (scalable, good), and
b. One can have all data fetched at once, as in ADO.NET (bad) as opposed to data fetched on demand, as in virtual mode (good).

So, ADO.NET is good in (a) and bad in (b). Some people wrongly assume that being good in (a) necessarily means being bad in (b). That is a misconception, and **DataObjects for .NET** virtual mode proves it. There is one qualification, though: If you consider arbitrary SQL, then yes, you cannot be good in (a) and (b) simultaneously since you need a live connection. But that is only for complex SQL statements, with GROUP BY, subqueries and other advanced features that are rarely if ever used in GUI front-ends. **DataObjects for .NET** virtual mode does not support full SQL (it does not support SQL-based and unbound tables, see Bound, SQL-Based and Unbound Tables). It supports only common cases (bound tables). This restriction together with sophisticated caching and optimization techniques allow it to be good both in (a) and in (b).

Using C1DataTableSource and Bound Controls

In order to access a virtual table view data, both programmatically and via data binding, you need to use a C1DataTableSource. This component serves two goals:

- It allows you to handle business logic events in your client application code, in addition to the business logic specified in the data library, see Business Logic for details.

AND

- It enables virtual mode in bound controls.

A C1DataTableSource is attached to a data set by setting its DataSet property and to a table view in that data set by setting its TableView property. Thus attached to a table view, a C1DataTableSource can be used as a DataSource for bound controls.

If you use another data source for your bound controls (if you bind them directly to the C1DataSet or to a C1DataView), then virtual mode will not work properly, **DataObjects for .NET** will not fetch segments on row repositioning, so the only data available will be the first fetched segment. Binding your controls to a C1DataTableSource attached to a virtual table view enables virtual mode in the bound controls.

C1DataTableSource also provides methods and properties for programmatic access to a virtual table view allowing navigation of the entire virtual rowset:

Member	Description
MoveFirst	Positions to the first row.
MoveLast	Positions to the first row.
MoveNext	Positions to the next row after current.
MovePrevious	Positions to the previous row before current.
Seek	Positions to a row specified by its primary key value.
CurrentRow	Returns the current row C1DataRow object.

You can bind any data-aware GUI controls to a C1DataTableSource, but there is one important consideration: In multi-record data-bound controls, such as grids, special functionality is required to make the scrollbar behave correctly in virtual mode. The **True DBGrid for .NET** control included in ComponentOne Studio Enterprise includes this functionality and fully supports **DataObjects for .NET** virtual mode. Therefore, if you need a grid control in virtual mode, we strongly recommend using the **True DBGrid for .NET** control. You do not need to purchase it separately, since it is provided together with **DataObjects for .NET** in ComponentOne Studio Enterprise.

Other grid controls, such as the standard Microsoft **DataGrid** control, work with **DataObjects for .NET** virtual mode too, but with one restriction: if you go to the end of the rowset using the scrollbar, the grid will show the end of the current segment, not the real end of the virtual rowset. Other than that, **DataObjects for .NET** virtual mode works with any bound controls without restrictions. For instance, if you position to the last row of the segment (or close to the last row), **DataObjects for .NET** will fetch more rows and the grid will show them. So, if this peculiarity in scrollbar behavior does not concern you, you can use the standard **DataGrid** and third-party grid controls with **DataObjects for .NET** virtual mode.

Asynchronous Fetch Modes

DataAccessMode = **Virtual** is recommended for especially large table views that not only take too much time to fetch, but also would consume too much client memory if fetched entirely to the client. In many cases the situation is somewhat different: the rowset is too large to be pre-fetched to the client at startup, that would take too much time, but it is small enough so all rows can be gradually brought to the client and stored there. For such situations, **DataObjects for .NET** supports two more virtual modes: *VirtualUnlimited* and *VirtualAutomatic* (DataAccessMode = *VirtualAutomatic* is more common, it is usually more suitable than *VirtualUnlimited*).

Data access mode *Virtual* means that data is fetched in chunks (called *segments*) of limited size (the size of each segment is determined by the VirtualSegmentSize property, default: 400), and the number of segments cached at the client at any given time is limited (this number is determined by the VirtualSegmentCount property, default: 4). Mode *Virtual* works for rowsets of virtually unlimited size, for example, in Tutorial 4 we demonstrate how it can be used to display 2.7 million rows in a grid.

Data access mode *VirtualUnlimited* is similar to *Virtual* in that data is fetched in segments, but the number of segments in the cache is unlimited, the VirtualSegmentCount property value is ignored. Once a segment is brought into the cache, it remains there. This setting is appropriate when you want to enhance performance by eliminating redundant roundtrips to the server, but it should not be used with very big rowsets if the user is expected to fetch too many segments into memory.

In *VirtualAutomatic* mode, data is fetched in segments, as in the previous two modes, and the number of segments in the cache is unlimited as in *VirtualUnlimited* mode, and, in addition to that, fetch is continually performed in background mode, asynchronously, until all data is fetched. This mode is appropriate for large rowsets that are big enough to make it undesirable to fetch all data at startup time, but not too big, so they still fit in client memory. This mode is additionally qualified by a Boolean VirtualConsolidateRows property. If it is set to **True** (default), then

DataObjects for .NET will rebuild the rowset when fetch is complete. While fetch is incomplete, Rows contains only rows of the current segment. Once fetch is complete, Rows contains all fetched rows, the whole rowset. In the other two virtual modes, Rows always contains only rows of the current segment.

The *VirtualAutomatic* mode is similar to the asynchronous fetch feature of ADO, the OLE DB-based data access framework used in Visual Studio 6. ADO.NET does not support asynchronous fetch, so **DataObjects for .NET** *VirtualAutomatic* mode is useful for developers that need this feature in .NET environment. Moreover, the **DataObjects for .NET** *VirtualAutomatic* mode is superior to the ADO asynchronous mode in one important aspect: the whole rowset up to the last row becomes accessible to the user immediately. The user does not have to wait until the fetch is complete to access the last row or any other row of the rowset. If, for example, the user positions to the last row while fetch is in progress, **DataObjects for .NET** will immediately fetch the last segment and display it to the user, without waiting for the background fetch to reach the end of the rowset.

Sorting Data in Virtual Mode

DataObjects for .NET supports sorting data in virtual mode, specified either at design (TableView.FillSort) or at run time (FillSort). However, sorting data in virtual mode will only have good performance if corresponding index exists in the database. An index is usually necessary for the database SQL optimizer to produce an efficient execution plan for a statement with ORDER BY.

In the **Schema Designer**, sort order is specified for a virtual mode table view by setting its FillSort property. If you specify TableView.FillSort for a table view in virtual mode, that sort order must form a unique key (there is no such requirement for table views in static mode). If FillSort does not represent a unique key, a run-time exception will occur when two different row with equal key values are fetched. You can always ensure uniqueness of a sort order by adding primary key fields in the end.

If no TableView.FillSort is specified for a table view in the schema, primary key order is used as the default sort.

It is also possible to specify sort order for a table view at run time. To change sort order of a table view at run time, call C1DataSet.Fill with a filter condition and a list of table view names containing the name of the table view in question. Specify FilterCondition.FillSort for the table view in one of the filter conditions. This FillSort string will override TableView.FillSort specified in the schema. This FillSort string must satisfy the same uniqueness requirement as TableView.FillSort specified in the schema. This FillSort string must satisfy the same uniqueness requirement asFillSort. In **DataObjects for .NET Express**, use FillSort property to specify sort order. To change sort at run time, set FillSort and call Fill.

Virtual Mode Restrictions

The following topics explain the necessary requirements and functionality restrictions in virtual mode.

Necessary Requirements for Virtual Mode

The following requirements must be satisfied for a table view to function properly in virtual mode:

- Only bound tables are allowed. Unbound and SQL-based tables cannot be used in virtual mode. See Bound, SQL-Based and Unbound Tables.
- The table must have a primary key, see Table Properties.
- An appropriate index must exist in the database for the specified sort order. More exactly, this is not a necessary condition, but virtual mode can have very poor performance if the index does not exist. An index is usually necessary for the database SQL optimizer to produce an efficient execution plan.
- The database must have sufficiently advanced query optimization capacity to interpret SQL statements issued by **DataObjects for .NET** in the optimal way, the way they intended to be interpreted. Microsoft SQL Server and Oracle satisfy this requirement, whereas Microsoft Access, being a low-end desktop database does not qualify. **DataObjects for .NET** virtual mode works correctly with Microsoft Access, but queries fetching data

may take a long time to execute on the server. SQL statements generated by **DataObjects for .NET** in virtual mode look approximately like this:

```
SELECT TOP 400 FROM T WHERE K1=? AND K2>? ...ORDER BY K1, K2,…
```

where K1, K2 – primary keys. You can see the actual generated SQL statements in the AfterGenerateSql event.

- The database must support the TOP n keyword in SQL limiting the resultset to the first n rows. An exception to this is Oracle, which does not support TOP n, but works with **DataObjects for .NET** virtual mode, because **DataObjects for .NET** uses a different construct with Oracle: the hint /*+ FIRST_ROWS */. To distinguish Oracle from other databases, the Syntax enumerated property is used, available in the **Connection Editor** in the **Schema Designer**.
- Primary key types must be ordered and comparable, that is, the > comparison operation must be applicable to them. All common SQL data types satisfy this requirement.
- If you use filter conditions with virtual table views, the conditions added to the WHERE clause of the generated SQL statement should not break query optimization. Basically, it means that if you restrict the sort order key values by filter conditions, try to restrict only an initial segment of the sort order key sequence. Avoid situations where a key is restricted and one of its predecessors in the sort order key sequence is not. This is not a mandatory requirement. Virtual mode will work correctly regardless of filter conditions. However, with very large rowsets, filter conditions breaking optimization can significantly affect performance.

Functionality Restrictions in Virtual Mode

Some **DataObjects for .NET** features change their behavior or are unavailable in virtual mode:

- The C1DataTable object of a virtual table view does not contain all rows of the virtual rowset. It contains only the rows of the current segment. To access all virtual rowset rows, use the C1DataTableSource methods and properties, see Using C1DataTableSource and Bound Controls.
- View relations having a virtual table view as the parent or a child do not return all child (parent) rows, they return only rows belonging to the current segment. This is simply because not all rows are fetched to the client.
- Binding GUI controls to a virtual table view, you have to use a C1DataTableSource as their data source; see Using C1DataTableSource and Bound Controls for more information.
- If you need a grid control bound to a virtual table view, use the **True DBGrid** control included in ComponentOne Studio Enterprise, see Using C1DataTableSource and Bound Controls.
- Scrollbar position in a grid bound to a virtual table view does not represent row number in the whole rowset, it represents row number in the current segment. This also causes the scrollbar thumb jumping from near the end to the middle of the scrollbar area when current segment changes. However, the scrollbar functions properly when clicked and dragged, including dragging to the end and the beginning of the rowset.

Virtual Mode Performance Tuning

The TableView schema object has several properties controlling performance parameters in virtual mode. In most cases, they can be left at their default values. However, you may want to try tuning them if you feel a need to improve performance.

Property	Description
VirtualSegmentSize	Number of rows fetched from the database and exposed to the user as a segment of the rowset. Default: 400. This is the minimum segment size. **DataObjects for .NET** can fetch more rows in a segment if necessary. That can happen if a grid with large number of rows is bound to the table view. With a bound grid, **DataObjects for .NET** always maintains the state where the first and last rows visible in the grid are on the "safe" distance from the ends of the current segment, the "safe distance" defined by the VirtualSyncThreshold property. Therefore, the segment

	size is always greater than the number of visible rows in the grid plus twice the threshold value.
VirtualSegmentCount	Maximum number of segments in the cache (only for DataAccessMode = *Virtual*).
VirtualSyncThreshold	Percent value from 0 to 100 (default: 12%). Determines the distance (number of rows) to the end of a segment that is considered a "danger zone", in the sense that the current segment must be changed to another segment (taken from the cache or fetched from the database) when the user positions on a row inside that zone. By default, it is 12% of 400 = 48 rows. See When a Virtual Table View Fetches Data Segments for details.
VirtualAsyncThreshold	Percent value from 0 to 100 (default: 30%). Determines the distance (number of rows) to the end of a segment that is considered a "preemptive fetch zone", in the sense that a new segment must be fetched from the database (if not found in the cache) when the user positions on a row inside that zone. By default, it is 30% of 400 = 120 rows. See When a Virtual Table View Fetches Data Segments for details.

Updating the Database

Transparency of database updates is one of the main power features of **DataObjects for .NET**. **DataObjects for .NET** solves the problem of updateable views and performs database updates automatically. For more information, see How the Data is Modified. Developers do not have to write server-side code updating the database. **DataObjects for .NET** can achieve this because it stores data internally preserving the original database structure, so it always knows what modifications must be applied to every database table. For more information see Structured Data Storage: Tables and Table Views and How the Data is Modified.

Although you can rely on **DataObjects for .NET** to perform database updates and do not have to write code, it is important to understand how the process of updating the database works. It may require customization for special application needs, and it is often necessary to specify business logic code resolving potential conflicts when different users concurrently modify the same data.

Before updating the database, **DataObjects for .NET** collects all modified simple table rows and sends them to the server for update (commit). Note that simple table modifications are sufficient for the update process. Composite tables and table views do not participate in update. Every data modification made by the end user ends up in a specific simple table, modifying a certain simple table row. Knowing the data structure, **DataObjects for .NET** tracks all modifications to their destination simple table rows, so in the end only simple table rows must be sent to the server for update. For more information, see Structured Data Storage: Tables and Table Views and How the Data is Modified.

For instance, considering the *Customer-Orders-Employees* example in Structured Data Storage: Tables and Table Views, the user could modify some rows of the *CustOrders* composite table (for example, using a grid control showing the combined customer-order-employee information), and also add and delete some rows. Since *CustOrders* is a composite table, it means that some *Customers* rows could have been modified, as well as some *Orders* and some *Employees* rows. Also, other modifications to these simple tables can be made in the same session as well, via other table views. All these modifications are just modified rows of the simple tables. Therefore, **DataObjects for .NET** already knows what database table rows must be modified, added or deleted to commit this transaction. It goes to the server and does just that, applies the modifications to the database tables *Customers*, *Orders* and, *Employees*. The rest of this chapter will be devoted to the process of applying modifications to database tables.

When the Database is Updated

By default, **DataObjects for .NET** works in **batch update mode**, standard in distributed applications (this is the only

update mode available in ADO.NET). Database update is performed on explicit command from the user or application code, a special method call, when the user presses a button, for example. This method is Update. Between Update calls, data modifications done by the end user are cached on the client, in the C1DataSet object. The end user can freely navigate between data rows, modifying them. Multiple rows can be in a modified state at the same time. Batch update mode is the only update mode supported by ADO.NET and the standard .NET Windows Forms framework.

DataObjects for .NET also supports the classic **automatic update mode**. This mode, common in classic client-server and desktop applications, ensures that all changes made in the current row are committed to the database when the end user leaves this row and moves to another row in a table. This mode is not supported in ADO.NET and standard WinForms data binding. **DataObjects for .NET** adds yet another enhancement to the .NET data framework by implementing the automatic update mode. If you use C1DataTableSource as your data source, and set its UpdateLeavingRow property to **True**, it will perform update automatically when the end user leaves a row after modifying it.

Update in 2-Tier and 3-Tier Configurations

When Update is issued on the client, **DataObjects for .NET** first collects all modified simple table rows. Further processing depends on whether it is a 2-tier or a 3-tier configuration (see Application Configurations):

- In a **2-tier** client (classic client-server configuration, that we also call a *direct client*), the client itself performs database updates. It generates SQL statements updating the database and sends them to the database server for execution. If you need to customize the update process, your business logic code can handle the C1TableLogic BeforeUpdate /C1TableLogic AfterUpdate events. These events fire for each modified row. The C1DataRow object passed to the event handler is a modified data row of the same client data set.
- In a **3-tier** configuration (distributed application), there are two identical instances of your data library participating in the process: one on the client and one on the server, see Application Configurations for details. An update request on the client sends the modified rows to the server. On the server, the server instance of your data library (having the same business logic code) takes over. It reconstructs a data set from the modified rows and then performs the update: generates SQL statements updating the database and sends them to the database server for execution. If you need to customize the update process, your business logic code can handle the C1TableLogic BeforeUpdate/C1TableLogic AfterUpdate events. These events fire for each modified row. The C1DataRow object passed to the event handler is a modified row received from the client. It belongs to a C1DataSet reconstructed on the server (we call it the *server data set*). The server data set does not include all rows that were originally fetched to the client. It only includes modified rows and those related to them through relations. Other than that, the structure of this data set is identical to the client data set. The server data set exists for the duration of serving the update request.

Update Process on the Server

Both in 2-tier and in 3-tier configurations, each update request is executed as a database transaction, that is, either all changes are successfully committed to the database or no changes are committed to the database. If a schema contains multiple database connections, a separate transaction is used for each database connection, all the transactions committed or rolled back together when all updates are done.

For each modified simple table, **DataObjects for .NET** applies updates in a certain order: it processes added rows first, then deleted rows, followed by modified rows. So, it first inserts rows that must be added, and then deletes rows that must be deleted, before modifying rows that must be changed. For each row involved, **DataObjects for .NET** applies the update by executing a generated SQL statement (in case of a bound table), see Generated SQL Statements. This cycle repeats for each simple table whose rows must be updated.

After a successful update, the updated and added rows are retrieved from the database to refresh the data according to the latest database state. This is necessary because the database can change the data, set default values, and so on. The process of refreshing rows after update can be customized with properties and events as well as the update process; for more information, see Generated SQL Statements and Changing Data as a Result of Update (Refresh).

Generated SQL Statements

When updating a row of a bound table, that is, a simple table with non-empty Connection and DbTableName properties, **DataObjects for .NET** generates and executes a SQL statement. Generated SQL statements are described in the following sections for each of the three possible cases: inserting, deleting and modifying rows. SQL statements are not generated for unbound and SQL-based tables, see Updating SQL-based and Unbound Tables.

The update process consists of three phases:

- *locating* the record in the database,
- *changing* (updating) the record, and
- *refreshing* the row, that is, retrieving the changed record from the database, as it could be changed by database own logic (triggers, and so on).

Row DELETE has only *locate* and *change* phases, no *refresh*. Row INSERT has only *change* and *refresh* phases, no *locate*. Row UPDATE has all three phases: *locate*, *change*, and *refresh*. The *locate* and *change* phases are performed in one SQL statement, *locate* being the condition in its WHERE clause, *change* – the action clause (INSERT/DELETE/UPDATE). The *refresh* phase is performed by a separate SQL SELECT statement finding a single record by its primary key value.

Generated SQL statements contain parameters specifying values used to locate the record in the database and values to assign to record fields. Parameter values are determined by the original and current values in the C1DataRow object:

Parameters used to locate the record are assigned the original field values, that is, the values as they have been fetched from the database before the user could modify them:

```
row["field_name", DataRowVersionEnum.Original]
```

Parameters used to set values of record fields are assigned the current values of the row:

```
row["field_name", DataRowVersionEnum.Current]
```

The generated parameterized SQL statement is used to set up and execute an IDbCommand object (for OLE DB: .NET Framework class System.Data.OleDb.OleDbCommand, for native SQL Server connection: System.Data.SqlClient, and so on). If necessary, this IDbCommand object can be customized in event code, see Events in Updating a Row.

Executing generated SQL statements is the default processing. It can be customized or substituted altogether with custom code using the BeforeUpdateRow, AfterUpdateRow events, see Events in Updating a Row.

Deleted Row

To delete a row from the database, **DataObjects for .NET** generates and executes the following SQL statement:

```
DELETE FROM table_name WHERE field1 = ? AND field2 = ? AND …
```

Parameter values are set to original row values: row["field_name", DataRowVersionEnum.Original].

If the original value of a field is Null, the condition is *field IS NULL*instead of *field = ?*.

The collection of fields used to locate the record in the WHERE clause depends on the table property UpdateLocateMode and field properties UpdateIgnore and UpdateLocate:

- Unbound fields (with empty DbFieldName), fields with UpdateIgnore = **True** and fields with UpdateLocate = **False** are not included regardless of the UpdateLocateMode value.
- If UpdateLocateMode = **PrimaryKey**, only primary key fields (those with PrimaryKey = *True*) are included.
- If UpdateLocateMode = **AllFields**, all fields are included provided they are not excluded by conditions above.
- If UpdateLocateMode = **PrimaryKeyAndChangedFields** (default), the locate fields include primary keys and fields whose values have changed, row["field_name", DataRowVersionEnum.Current] is not equal to

row["field_name", DataRowVersionEnum.Original].

Locating the record for delete can fail (no record found), due to changes made to the database by other users between the time the record was fetched and the time it is deleted. This is usually called optimistic concurrency; see Handling Concurrency Conflicts for more information. Including more fields in the WHERE clause (manipulating the UpdateLocateMode and UpdateLocate properties) makes the concurrency check stricter: changes made by other users to one of this field can fail the delete. Excluding fields from the WHERE clause makes the check less strict, allowing more concurrency, more tolerant to changes made by other users.

If no record has been found for delete, it may mean that the row has been already deleted. Since this does not necessarily represent an error condition (for example, the row could be deleted as a result of cascade delete), a special property IgnoreDeleteError allows you to suppress the error in this case. To check the "row deleted" condition **DataObjects for .NET** tries to locate the database row with the original primary key value (as in UpdateLocateMode = **PrimaryKe***y*). If such a row cannot be located, it means the row has been deleted. Handling this situation depends on the table property IgnoreDeleteError. If IgnoreDeleteError = **True** (default), this error is ignored, it does not generate an update error. If IgnoreDeleteError = **False**, this error generates an update error.

Added Row

To add a row to the database**, DataObjects for .NET** generates and executes the following SQL statement:

```
INSERT INTO table_name (field1, field2, …) VALUES (?, ?, …)
```

Parameter values are set to the current row values: row["field_name"].

The collection of fields used to set the values in the newly added record depends on the field properties UpdateIgnore and UpdateSet and DataSourceReadOnly:

- Unbound fields (with empty DbFieldName), fields with UpdateIgnore = **True** and fields with DataSourceReadOnly = **True** are not included regardless of the UpdateSet value.
- If UpdateSet = **Always**, the field is included provided the other properties do not preclude it.
- If UpdateSet = **Never**, the field is not included.
- If UpdateSet = **IfChanged** (default), the field is included if its value has been set by the user.

After a new row has been successfully created, **DataObjects for .NET** generates and executes another SQL statement, to retrieve the new row from the database:

```
SELECT selectField1, selectField2, … FROM table_name WHERE keyField1 = ? AND keyField2
= ? AND …
```

This *refresh phase* is needed because some field values could be assigned by the database itself, such as autoincrement fields, and so on.

The refresh phase is controlled by the table property UpdateRefreshMode:

- If UpdateRefreshMode = **Always**, the refresh phase always takes place, and its failure generates an error.
- If UpdateRefreshMode = **Never**, the refresh phase is skipped.
- If UpdateRefreshMode = **IfPossible** (default), the refresh phase takes place, but its possible failure does not generate an error.

Fields selectField1, selectField2, … are those whose values are retrieved from the new database row. The retrieved values, if different from the row values sent for update, substitute the row values and the client receives them back in the refreshed row.

A bound field (with non-empty DbFieldName) is included in selectField1, selectField2, …if its UpdateRefresh property is set to **True**.

Fields keyField1, keyField2, … used to locate the record in the WHERE clause, are the primary key fields of the table.

Parameter values are set to the current row values of primary keys: row["field_name"].

Modified Row

To modify a database row, **DataObjects for .NET** generates and executes the following SQL statement:

```
UPDATE table_name SET (setField1 = ?, setField1 = ?, …) WHERE whereField1 = ? AND
wherefield2 = ? AND …
```

The collection of fields used to set the values in the database row, setField1, setField2, … depends on the field properties UpdateIgnore and UpdateSet and DataSourceReadOnly:

- Unbound fields (with empty DbFieldName), fields with UpdateIgnore = **True** and fields with DataSourceReadOnly = **True** are not included regardless of the UpdateSet value.
- If UpdateSet = **Always**, the field is included provided the other properties do not preclude it.
- If UpdateSet = **Never**, the field is not included.
- If UpdateSet = **IfChanged** (default), the field is included if its value has been changed by the user, row["field_name", DataRowVersionEnum.Current] is not equal to row["field_name", DataRowVersionEnum.Original].

Parameters used to set values are assigned the current row values: row["field_name"].

The collection of fields used to locate the record for update in the WHERE clause, whereField1, whereField2, … depends on the table property UpdateLocateMode and field properties UpdateIgnore and UpdateLocate:

- Unbound fields (with empty DbFieldName), fields with UpdateIgnore = **True** and fields with UpdateLocate = **False** are not included regardless of the UpdateLocateMode value.
- If UpdateLocateMode = **PrimaryKey**, only primary key fields (those with PrimaryKey = *True*) are included.
- If UpdateLocateMode = **AllFields**, all fields are included provided they are not excluded by conditions above.
- If UpdateLocateMode = **PrimaryKeyAndChangedFields** (default), the locate fields include primary keys and fields whose values have changed, row["field_name", DataRowVersionEnum.Current] is not equal to row["field_name", DataRowVersionEnum.Original].

Parameter used to locate the record are set to original row values: row["field_name", DataRowVersionEnum.Original].

If the original value of a field is Null, the condition is *field IS NULL*instead of *field = ?*.

Locating the record for update can fail (no record found), due to changes made to the database by other users between the time the record was fetched and the time it is updated. This is usually called optimistic concurrency, see Handling Concurrency Conflicts for more information. Including more fields in the WHERE clause (manipulating the UpdateLocateMode and UpdateLocate properties) makes the concurrency check stricter: changes made by other users to one of this field can fail the update. Excluding fields from the WHERE clause makes the check less strict, allowing more concurrency, more tolerant to changes made by other users.

After the database row has been successfully modified, **DataObjects for .NET** generates and executes another SQL statement, to retrieve the modified row from the database:

```
SELECT selectField1, selectField2, … FROM table_name WHERE keyField1 = ? AND keyField2
= ? AND …
```

This *refresh phase* is needed because some field values could be changed by the database itself, using triggers, and so on.

The refresh phase is controlled by the table property UpdateRefreshMode:

- If UpdateRefreshMode = **Always**, the refresh phase always takes place, and its failure generates an error.
- If UpdateRefreshMode = **Never**, the refresh phase is skipped.

- If UpdateRefreshMode = **IfPossible** (default), the refresh phase takes place, but its possible failure does not generate an error.

Fields selectField1, selectField2, ... are those whose values are retrieved from the new database row. The retrieved values, if different from the row values sent for update, substitute the row values and the client receives them back in the refreshed row.

A bound field (with non-empty DbFieldName) is included in selectField1, selectField2, ...if its UpdateRefresh property is set to **True**.

Fields keyField1, keyField2, ... used to locate the record in the WHERE clause, are the primary key fields of the table.

Parameter values are set to the current (row["field_name"]) or to the original (row["field_name", DataRowVersionEnum.Original]) values of primary keys, depending on whether a primary key field's value has been set by the UPDATE SQL statement. If it has not been set in UPDATE, the original value is used. If it has been set, the current value is used.

Events in Updating a Row

After an IDCommand is set up with the generated SQL statement, **DataObjects for .NET** fires the C1TableLogic**.**BeforeUpdateRow event. Using this event, you can customize the command; for example, change the parameter substitution, if necessary. This event is also used to implement updates in SQL-based and unbound tables, see Updating SQL-based and Unbound Tables. In this event procedure, you can set the *Status* argument specifying how to proceed after the event is handled. The default value, *Continue* means the command is to be executed. If you want to take full control over update and cancel the default processing (command execution), do whatever is necessary to update the row in your event handler procedure and set the *Status* argument to *SkipCurrentRow*. There are also *ErrorsOccurred* and *SkipAllRemainingRows* options used to handle update failures. Another event argument, Row contains the C1DataRow being updated. It can be used to extract the current and original values of each field, if you need them to perform a custom update operation. After update, you can modify the field values in this row to customize data refresh after update, see Changing Data as a Result of Update (Refresh).

If not skipped by setting the *Status* argument of the BeforeUpdateRow event, the update itself takes place, the command containing the INSERT, DELETE or UPDATE SQL statement is executed. After that, except for deleted rows, the SELECT command is executed to refresh the data with the latest database values, see Changing Data as a Result of Update (Refresh).

After executing generated commands, **DataObjects for .NET** fires the C1TableLogic**.**AfterUpdateRow event. This event can be used, for example, for customizing data refresh after update, by setting the values in its Row argument.

Changing Data as a Result of Update (Refresh)

After a successful update, the updated and added rows are retrieved from the database. This is necessary because the database can change the data, for example, set default values, and because the data could be modified by other users, if such concurrency is allowed by the UpdateLocateMode and UpdateLocate properties, see Generated SQL Statements.

Retrieving the current data from the database is called the refresh phase of the update process, see Generated SQL Statements. The retrieved values, if different from the row values sent for update, substitute the row values in the originally sent row, and the client receives them back in the refreshed row.

The process of refreshing rows after update can be customized using the UpdateRefreshMode and UpdateRefresh properties. It can also be customized in the AfterUpdateRow event. Setting values in the *Row* argument of the event, you can specify the values returned to the client in the refreshed row. The *Row* values can also be changed in the BeforeUpdateRow event, but then they are also used in updating the database row, whereas setting values in AfterUpdateRow only affects the values returned to the client. In addition to BeforeUpdateRow/AfterUpdateRow events, you can use the AfterUpdate event to programmatically change the refreshed rows. That event is fired once after all rows are updated.

For bound tables (with non-empty Connection and DbTableName properties), the refresh occurs automatically, using a generated SQL statement, see Generated SQL Statements. For an SQL-based table with attached DataAdapter component, the refresh is also done automatically, by the DataAdapter component. For SQL-based without DataAdapter component and for unbound tables, the refresh must be done in code, if necessary, in the AfterUpdateRow event (or in BeforeUpdateRow, see above) by setting the row values in the *Row* argument. For details, see Bound, SQL-Based and Unbound tables.

Updating SQL-Based and Unbound Tables

For bound tables (with non-empty Connection and DbTableName properties), and for SQL-based tables (with non-empty Connection and SelectCommandText properties) with DataAdapter component, the update process is automatic, although it can be customized with properties and events.

For SQL-based without DataAdapter component and for unbound tables, using events is required to provide update functionality, see Bound, SQL-Based and Unbound Tables.

Updating SQL-Based Tables

An SQL-based table is a table with DataMode=SqlBased and non-empty Connection property, see Bound, SQL-Based and Unbound Tables. A SQL-based table can have a DataAdapter component associated with it, via the DataAdapter property of the C1TableLogic component (in **DataObjects for .NET Express**, DataAdapter is a property of C1ExpressTable).

For an SQL-based table with DataAdapter, the update is performed automatically by the DataAdapter component. For an SQL-based table without DataAdapter component, or if DataAdapter does not contain *UpdateCommand*, *InsertCommand*, or *DeleteCommand*, the update is performed in code.

To allow updating the database from an SQL-based table without DataAdapter, write code in the C1TableLogic**.**BeforeUpdateRow event. According to the state (added/deleted/modified) of the row passed to the event in the *Row* argument, create one of the three commands (SQL statements in the form of an IDbCommand object), for insert, delete or update correspondingly, and assign the created command object to the corresponding argument: *InsertCommand*, *DeleteCommand*or, *UpdateCommand*.

Optionally, if you need to refresh the updated row from the database, you can create a SELECT command retrieving the updated row and assign it to the *SelectCommand* argument. For more information, see Changing Data as a Result of Update (Refresh).

Note also, that if the table is read-only or just does not need update functionality, there is no need for update code in any case.

Updating Unbound Tables

An unbound table is a table with DataMode = *Unbound*. In this case the Connection, DbTableName and SelectCommandText properties are empty. An unbound table has its data fetched and updated entirely in code. It can be used to represent a custom non-SQL data source, see Bound, SQL-Based and Unbound Tables.

To allow updating of the data source from an unbound table, write code in the C1TableLogic**.**BeforeUpdateRow event. In that code, perform the necessary update programmatically, doing whatever actions the custom data source requires, and set the *Status* event argument to *SkipCurrentRow*, indicating that the update has been done.

Optionally, if you need to refresh the updated row from the data source (see Changing Data as a Result of Update (Refresh)), retrieve the field values that you need from your data source and assign them to the fields of the *Row* argument passed to the event.

Note also, that an unbound table can be read-only or it can be modifiable without update functionality, in which case there is no need for any update code.

Controlling the Update Process

Before any processing of modified rows starts on the server, it fires the BeforeUpdate event. This event is fired on the server, in the same manner as the BeforeUpdateRow/AfterUpdateRow events, but, unlike them, it is fired only once. Its *DataSet* argument contains the data set passed to the server for update. By modifying this data set, developers can customize the update process and control the set of rows and field values that undergo the database update. The C1DataSetLogic component whose event is fired corresponds to the client-side C1DataSet component that originated the update.

After the BeforeUpdate event, **DataObjects for .NET** starts scanning all modified rows in a certain order and applying the changes to the database. For more information, see Update Process on the Server. Before processing each row, it fires first the C1DataSetLogic**.**BeforeUpdateRow event, then the C1TableLogic**.**BeforeUpdateRow event. The C1DataSetLogic component whose event is fired corresponds to the client-side C1DataSet component that originated the update. The C1TableLogic component represents the simple table containing the row in question.

Using C1TableLogic**.**BeforeUpdateRow is more common than using C1DataSetLogic**.**BeforeUpdateRow. A C1TableLogic component represents business logic pertaining to a certain table regardless of the data set where it is used. However, if you need some specific logic depending on the data set context, use the C1DataSetLogic**.**BeforeUpdateRow event. This, of course, applies to the C1DataSetLogic**.**AfterUpdateRow event as well.

Both BeforeUpdateRow events have *Status* argument. Setting it to *Skip* allows you to cancel further processing for the current row.

After processing the row as described in Generated SQL Statements, **DataObjects for .NET** fires the C1TableLogic.AfterUpdateRow event and C1DataSetLogic.AfterUpdateRow event, in this order. For more information, see Generated SQL Statements. The values in its *Row* argument are the values that are later passed back to the client as the current database values in the process of refreshing the rows after update, see Changing Data as a Result of Update (Refresh). Changing the values in the AfterUpdateRow event allows developers to control the refreshed values, if necessary.

Finally, when all rows are processed, the AfterUpdate event is fired. Among other possible uses, it allows you to control the whole data set (via its *DataSet* argument) sent back to the client for refreshing updated rows, see Changing Data as a Result of Update (Refresh).

All these events, BeforeUpdate / AfterUpdate and BeforeUpdateRow / AfterUpdateRow are server-side events. They fire on the server only. See Application Configurations for the description of **DataObjects for .NET** client and server. However, this does not apply to a *direct client* situation, where a 2-tier application updates the database directly from the client, an application where C1SchemaDef and C1DataSet components reside on the same design surface. In this case, the client and the server are one and the same application.

Handling Errors in Update

There can be two different kinds of errors during update:

Concurrency Conflicts

Updating a row can fail because another user has changed the same row between the time the row was fetched and the time it is updated. Different applications handle this situation differently, and it does not always represent an error condition. See Handling Concurrency Conflicts for details.

Program and Physical Errors

An update can fail due to a program error, for example, an incorrect DbTableName property setting, or to a physical

error, such as failed database connection. In this case, the Update method execution is aborted, the database transaction is rolled back, and an exception is thrown. Before throwing an exception, the AfterUpdate event is fired, giving the developer an opportunity to clean-up whatever resources could be allocated in the BeforeUpdate event. After that, the exception is passed to the client. On the client, it first fires the UpdateError event, the exception is passed to it as the *Error* argument. In that event, the developer has an opportunity to handle this situation without throwing an exception, by setting the *Status* argument to *Continue* or *Skip*. If the *Status* argument is left at the default value *ErrorsOccurred*, then the exception is thrown, aborting the Update method execution. That exception can then be handled the usual way. See Handling Update Errors on the Client for details.

Handling Concurrency Conflicts

Updating a row can fail because another user has changed the same row between the time the row was fetched and the time it is updated. Allowing concurrent changes to rows with "first tried – first succeeded" policy is usually called optimistic concurrency. When updating a row, your application must be sure that it is actually updating the same row as it had originally fetched from the database. In different application scenarios, the notion of "the same row" can be different. In some cases, you may need all the fields in the row in the database to remain intact from the moment the row was fetched till the moment it is updated. In other cases, it is enough that an ID, a primary key field remains intact, and all other fields can be allowed to be freely modified by various users on the "first tried – first succeeded" basis. To set this concurrency control policy, use the UpdateLocateMode and UpdateLocate properties. They control the collection of fields used in the WHERE clause of the action SQL statements to locate the record for update. A row passes the concurrency check if the WHERE clause finds a row in the database for update. Including more fields in the WHERE clause (manipulating the UpdateLocateMode and UpdateLocate properties) makes the concurrency check more strict: changes made by other users to one of these fields fail the update. Excluding fields from the WHERE clause makes the check less strict, making it more tolerant to changes made by other users.

When a concurrency conflict occurs, it fires the C1TableLogic.AfterUpdateRow event with its *SqlStatus* argument set to *ConcurrencyConflicts*. At this point, developers have an opportunity to reconcile this conflict directly on the server. Business logic does not always allow you to reconcile conflicts on the server, because this must be done without user interface and without asking for user choices. Still, if that is possible from the business point of view, it is usually preferable. To reconcile a conflict in the AfterUpdateRow event, do whatever needs to be done to update the row (remember that default update by the generated SQL statement has not been done because of the conflict), change the field values in the *Row* argument to reflect possible changes to the fields due to conflict resolution, and set the *SqlStatus* argument to *Succeeded*.

When the C1TableLogic.AfterUpdateRow fires on a conflicting row, its *Status* argument is set to *Continue*. This means that the conflict will not be regarded as an exceptional situation, a fatal failure, that **DataObjects for .NET** will continue processing other modified rows after that. However, if you need to abort processing at this point, you can do so by setting the *Status* argument to *SkipAllRemainingRows* or to *ErrorsOccurred*. The latter also has the additional effect of raising an exception on the client.

Conflicts that are not resolved on the server are passed to the client and have to be reconciled there, see Handling Update Errors on the Client. Concurrency conflicts do not raise an exception on the client, but fire the UpdateError event. Rows that caused conflicts can be found in the C1DataSet using the properties and methods RowError, HasErrors, HasErrors and GetErrors.

Note that this distinction between client and server is not applicable to a *direct client* application (see Application Configurations for the description of **DataObjects for .NET** client and server), that is a 2-tier application updating the database directly from the client, an application where C1SchemaDef and C1DataSet components reside on the same design surface. In this case, the client and the server are one and the same application so you can reconcile conflicts in the AfterUpdateRow event with user interface, see Handling Update Errors on the Client.

Handling Update Errors on the Client

When an error condition of any kind occurs on the server during an update, the UpdateError event fires. If the error is a fatal failure, the *Error* argument of the UpdateError event contains the exception object describing the failure. For

more information on a program or physical error, see Handling Errors in Update. If this is a concurrency conflict that could not be reconciled on the server and so was passed to the client, the *Error* argument is set to *null* (*Nothing* in Visual Basic). In case of a concurrency conflict, the rows that failed update can be found in the C1DataSet using the properties and methods RowError, HasErrors, HasErrors and GetErrors.

When writing code in the UpdateError event, developers can handle errors and reconcile concurrency conflicts. If you consider the error conditions resolved and want to make a new attempt to update the database, set the *Status* argument to *Continue*. That will repeat the Update method call. If there will still be errors, the UpdateError event will fire again, and this loop will continue until the update succeeds or until you exit the UpdateError event with *Status* = *Skip* or *Status* = *ErrorOccurred*. If *Status* is set to *Skip*, the update loop ends without exception, in effect, the Update call is ignored. If *Status* is set to *ErrorsOccurred*, an exception is thrown.

In the case of a fatal error (non-empty *Error* argument), the usual reaction is to set *Status* to *ErrorOccured* (throw an exception), or to *Skip* (ignore the Update call).

In the case of a concurrency conflict (empty *Error* argument), the recommended practice is to present the user with a special dialog(s) offering to reconcile the conflicts, for example, to choose between the values in the data set and the values in the database (to retrieve the current database values, use a separate C1DataSet object with the same schema). After reconciling the conflicts, repeat the update attempt by setting the *Status* argument to *Continue*. Alternatively, you can set *Status* to *Skip* before reconciling conflicts, and then repeat Update programmatically.

Features and Techniques

This chapter demonstrates how you can use **DataObjects for .NET** to solve common problems that arise when developing database applications. It also describes various techniques used in **DataObjects for .NET**.

DataObjects for .NET Expressions

DataObjects for .NET uses expressions to specify constraints and field calculations. **DataObjects for .NET** expression language is based on the ADO.NET expression language; for more information see the reference entry for the System.Data.DataColumn.Expression property in .NET Framework Reference documentation. For your convenience, the expression language reference is partly reproduced below.

Expression variables are fields of tables and table views, see Schema Objects. In an expression belonging to a table or table view, you can use fields from the same table (or table view) and from its parents and children with regard to relations, if they are included in the same data set.

In addition to operators and functions defined in the ADO.NET expression language, **DataObjects for .NET** expressions support the following functions:

Function	Description
Current	Syntax: Current(field). Example: Current(OrderDate). Returns the value of the field in a row as it was before editing of the row started (BeginEdit was called). It is the same value as returned by C1DataRow["field_name", *DataRowVersionEnum.Current*]. If the row is not in edit mode, this function returns the same value as the field without a function. For a row in edit mode, the field without a function returns C1DataRow["field_name", *DataRowVersionEnum.Proposed*].
Original	Syntax: Original(field). Example: Original(OrderDate). Returns the value of the field in a row as it was when the row was fetched from the database. It is the same value as returned by C1DataRow["field_name", *DataRowVersionEnum.Original*].
BeforeChange	Syntax: BeforeChange(field). Example: BeforeChange(OrderDate). When the value

	of a field is being changed (including in events BeforeFieldChange and AfterFieldChange), the field variable returns the new value even if the new value has not yet been assigned to the field (for example, in the BeforeFieldChange event). If you need the field value as it was before the change, use this function. While the field value is being changed (including in events BeforeFieldChange and AfterFieldChange), this function returns the value before change. If the field value is not being changed when the function is evaluated, it returns the same value as the field variable without a function.

ADO.NET Expression Language Reference

DataObjects for .NET expression language is based on the ADO.NET expression language. For a more detailed description of the ADO.NET expression language, see the reference entry for the System.Data.DataColumn.Expression property in .NET Framework Reference documentation.

Escaping Special Characters and Reserved Words

Variable (field) names must be enclosed in square brackets if they include blanks or any of the following special characters, or coincide (ignoring case) with a reserved word.

Special characters: \n (newline) \t (tab) \r (carriage return) ~ ( ) # \ / = > < + - * % & | ^ ' " [ ]

Reserved words: CHILD, PARENT, CURRENT, ORIGINAL, BEFORECHANGE, IN, LIKE, AND, OR, NOT, CONVERT, LEN, ISNULL, IIF, SUBSTRING.

Examples of escaped field name: [Order Date], [CustomerID#]

If a field name contains closing bracket, it must be escaped with \. For example, field name Customer[Name] is written as [Customer[Name\]].

Constant Values

Strings are enclosed in single quotes. Dates are enclosed in pound signs.

Fore example:

Discount = 0.11

CustomerName = 'ALFKI'

OrderDate = #12/31/2001#

Operators

Comparison operators: <, >, <=, >=, <>, =, IN, LIKE (both wildcards * and % can be used in LIKE)

Boolean operators: AND, OR, NOT

Arithmetic operators: +, -, *, /, % (modulus).

String operator: + (concatenation)

Parent/Child Fields

A parent field is prepended with "Parent(relation_name).". For example, Parent(Customers – Orders).CustomerName

used in the Orders table references the CustomerName field of the Customers table. If there is only one relation connecting the tables (or table views), the relation name can be omitted. Example: Parent.CustomerName

A child field is prepended with "Child(relation_name).". For example, Child(Customers – Orders).OrderDate used in the Customers table references the OrderDate field of the Orders table. If there is only one relation connecting the tables (or table views), the relation name can be omitted. Example: Child.OrderDate

Aggregation Functions

Aggregation is usually performed in a parent table over a child table, as a function applied to a child field. Example: Sum(Child(Orders – OrderDetails).Quantity). Used in the Orders table, this expression calculates the sum of the Quantity fields of the OrderDetail table rows related to the Orders row.

Supported aggregation functions: Sum, Avg (average), Min (minimum), Max (maximum), Count, StDev (statistical standard deviation), Var (statistical variance).

Field Value Qualifying Functions

Current(field), Original(field), BeforeChange(field).

These functions are supported by the **DataObjects for .NET** expressions in addition to the functions supported by the ADO.NET expression language. They apply to field variables and allow functions to obtain original field values and values before change, see DataObjects for .NET Expressions.

Other Functions

Other functions include:

Function	Description
Convert(value, type)	Converts a value to the specified type. Example: Convert(Quantity, 'System.Int32')
Len(string)	Returns the length of a string. Example: Len(CompanyName)
IsNull(value, replacement_value)	Returns value if it is not Null and replacement_value if value is Null. Example: IsNull(OrderDate, #1/31/2001#) returns #1/31/2001# if Orderdate is Null.
IIF(condition, true_value, false_value)	Returns true_value if condition is **True**, and false_value if condition is **False**. Example: IIF(A < B, A, B) returns the minimum of A and B.
Substring(string, start_position, length)	Returns a substring of specified length starting at start_position. Example: Substring(CompanyName, 1, 5) returns first five characters of the CompanyName value.

Adding Rows and Primary Keys

Adding new rows often, if not always, presents a serious problem with poor help from standard data frameworks, including ADO.NET. Specifically, the problem of primary key values in the newly added rows. Primary keys, such as, ProductID, OrderID, and so on, are often assigned by a centralized procedure on the server or by the database itself (as AutoIncrement), so they are unknown until the new row is actually added to the database. That creates a lot of

problems for GUI front ends that are commonly solved by restricting functionality, forcing early database updates and other objectionable techniques.

In this section, you will see how **DataObjects for .NET** solves this problem. We distinguish two different cases, two practical mechanisms of assigning primary keys to new rows. Primary key values can be assigned by the client application or by the server (or the database itself).

Keys Assigned by Client: New Row Detached and Attached State

This is the simplest case, where a key value is assigned directly by the client application, in essence, by the end user. For example, in the Northwind sample database, the key of the *Customers* table, *CustomerID*, is a string abbreviation of the customer name: "ALFKI", "ANTON", and so on, assigned by the user. In this case, there is no "unknown key" problem. However, there is still a problem of transitional state in newly added rows:

Each table row in **DataObjects for .NET** must have a definite and unique primary key value. A row with an undefined primary key is in a special transitory state called *detached*. Such a row cannot be used for any purpose except setting its field values. For example, detached rows cannot be used in updating the database. When a new row is added to a table, for example, in a bound grid, or programmatically, with AddNew, it is initially in detached state. To change its state to *attached*, that is, a fully functional row that can be sent for update, you need to set its primary key field(s) and call the EndEdit method (explicitly or implicitly, see next about AutoEndAddNew).

If setting the primary key is the responsibility of the end user, you do not have to worry about the transition from detached to attached state, **DataObjects for .NET** does it for you. Its default behavior supports assigning primary key values by the end user. Table views have a special property, AutoEndAddNew. If it is set to *True*, **DataObjects for .NET** automatically calls EndEdit when the row's primary key is set for the first time. If the primary key consists of multiple fields, EndEdit will be called when all the key fields receive definite values.

Sometimes, you may need programmatic control over setting primary keys for new rows. For example, you may need to set primary key automatically when a new row is added. This can be done in your business logic code, in the AfterAddNew event, for example:

To write code in Visual Basic

Visual Basic

```
Private Sub table_Customers_AfterAddNew(ByVal sender As Object, ByVal e As
C1.Data.RowChangeEventArgs) Handles TableLogic1.AfterAddNew
    e.DataTable.DataSet.PushExecutionMode _
         (C1.Data.ExecutionModeEnum.Deferred)
    e.Row("CustomerID") = TextBox1.Text
    e.Row.EndEdit()
    e.DataTable.DataSet.PopExecutionMode()
End Sub
```

To write code in C#

C#

```
private void table_Customers_AfterAddNew(object sender, C1.Data.RowChangeEventArgs e)
{
    DataTable.DataSet.PushExecutionMode(ExecutionModeEnum.Deferred);
    e.Row["CustomerID"] = textBox1.Text;
    e.Row.EndEdit();
    e.DataTable.DataSet.PopExecutionMode();
```

```
}
```

Execution mode *Deferred* is used here instead of default mode *Immediate*, because we need the setting of the primary key and the EndEdit call to execute after all actions related to adding a new row are completed (including notifying bound controls that a row has been added, and so on; it is not safe to perform actions like **EndEdit** while other actions, such as AddNew have not yet been completed), see Action Order and Execution Mode.

Keys Assigned by Server or Database

It is common to allocate new primary key values on the application server or in the database itself (autoincrement keys, triggers, and so on). In this case, the real primary key values as they will appear in the database are not known until the new rows are actually inserted into the database. This is the "unknown key" problem. **DataObjects for .NET** solves this problem by allowing you to work with temporary key values until the rows are sent to the database for update.

Before an update, you can use arbitrary temporary values for primary keys. The only requirement is that those temporary values must be unique, not intersecting with real (after update) primary key values. For example, the temporary values can be made negative to ensure their uniqueness. This is done automatically by **DataObjects for .NET** if you use property setting AutoIncrement = **ClientAndServer**.

When the rows are created on the client, they acquire temporary primary key values (become attached), as described in Keys Assigned by Client: New Row Detached and Attached State. Rows with temporary primary key values can be used on the client without restrictions, including adding child rows related to them, and so on.

During database update, when the new rows are actually inserted into the database, **DataObjects for .NET** obtains final primary key values and substitutes the temporary values on the client with the final values. This client primary key value refresh can be done automatically by **DataObjects for .NET** if you use property setting AutoIncrement = **ClientAndServer**, as described in Key Assigned Automatically by Database, or it can be performed programmatically, as shown in Programmatic Key Assignment.

Key Assigned Automatically by Database

Let's first consider the case where the new primary key value is created automatically by the database upon executing an INSERT command. It can be an autoincrement database field, or a database trigger setting the field on INSERT.

For integer primary keys with autoincrement functionality in the database, **DataObjects for .NET** provides a property setting, AutoIncrement = **ServerAndClient** that makes the whole process fully automatic. In this case, AutoIncrementSeed = **-1** and AutoIncrementStep = **-1**, which makes temporary key values on the client negative to guarantee their uniqueness. When a row is added to the database, **DataObjects for .NET** uses the database capacity to retrieve the actual key value after the row is added. See IdentityColumnLastValueSelect property for explanation on how to find out if the database supports this functionality. Having retrieved the actual primary key value, **DataObjects for .NET** sends it back to the server as the replacement for the temporary value.

Some databases do not support automatic identity assignment (autoincrement) to database fields on adding new rows, but they support special objects (usually called sequence or generator) for generating unique identity values that are used to set identity key values in INSERT. Such databases are, for example, Oracle (object: sequence) and Interbase (object: generator). To ensure automatic identity value update on adding new rows for such databases, set the IdentityColumnLastValueSelect property to the SQL command retrieving identity value, set the field's AutoIncrementSequenceName property to the corresponding sequence (generator) object name, and set the field's IdentityColumnRetrieveMode to **BeforeInsertCommand**. **DataObjects for .NET** will obtain a new identity value, use it when inserting the new row, and refresh the identity row on the client. This is the recommended way of dealing with autoincrement keys in Oracle and in Interbase. You also have an option of creating a trigger on INSERT and setting the autoincrement key value in the trigger. This is not necessary unless you need this trigger for other purposes or already have such trigger in the database. If you do define a trigger, use IdentityColumnRetrieveMode = **AfterInsertCommand** with IdentityColumnLastValueSelect and AutoIncrementSequenceName to refresh the

autoincremented value on the client.

If your database does not support the functionality of retrieving autoincremented (identity) value required by IdentityColumnLastValueSelect, you can still use AutoIncrement = **ServerAndClient**, but you need to retrieve the key value after row insert using the AfterUpdateRow event, or perform the whole row insert operation including key value retrieval in the BeforeUpdateRow event. For example, the following code using AfterUpdateRow event can be used to retrieve the actual key value after insert from a database that does not support IdentityColumnLastValueSelect:

To write code in Visual Basic

Visual Basic

```
Private Sub table_Orders_AfterUpdateRow(ByVal sender As Object, ByVal e As
C1.Data.RowUpdateEventArgs) Handles table_Orders.AfterUpdateRow
    Dim connection As C1.Data.Connection
    Dim command As System.Data.OleDb.OleDbCommand
    Dim reader As System.Data.OleDb.OleDbDataReader
    connection = SchemaDef1.Schema.Connections("Connect")
    command = New System.Data.OleDb.OleDbCommand( _
        "SELECT TOP 1 OrderID FROM Orders ORDER BY OrderID DESC", _
         CType(connection.DbConnection, System.Data.OleDb.OleDbConnection))
    command.Transaction = CType(connection.DbTransaction, _
        System.Data.OleDb.OleDbTransaction)
    reader = command.ExecuteReader()
    reader.Read()
    e.Row("OrderID") = reader("OrderID")
    reader.Close()
End Sub
```

To write code in C#

C#

```
private void table_Orders_AfterUpdateRow(object sender, C1.Data.RowUpdateEventArgs e)
{
    C1.Data.Connection connection = schemaDef1.Schema.Connections["Connect"]);
    OleDbCommand command = new System.Data.OleDb.OleDbCommand (
        "SELECT TOP 1 OrderID FROM Orders ORDER BY OrderID DESC",
(OleDbConnection)connection.DbConnection);
    command.Transaction = (System.Data.OleDb.OleDbTransaction)
        connection.DbTransaction;
    System.Data.OleDb.OleDbDataReader reader = command.ExecuteReader();
    reader.Read();
    e.Row["OrderID"] = reader["OrderID"];
    reader.Close();
}
```

Programmatic Key Assignment

If the new primary key value is created programmatically, according to a certain rule, on the server, then it is done in the BeforeUpdateRow event. First, a new primary key value is allocated or calculated in code (maybe using a stored procedure or other mechanisms) and assigned to the corresponding field(s) in the *Row* argument. Thus the temporary key value originally present in *Row* is substituted with the final, real value. When **DataObjects for .NET** inserts the row

into the database after the event, it uses the new value. When it then sends the row back to the server for refresh (see Changing Data as a Result of Update (Refresh)), the new value is sent to the client. Having received the new value, the client substitutes the temporary value with the real one.

To write code in Visual Basic

Visual Basic

```
Private Sub table_Orders_BeforeUpdateRow(ByVal sender As Object, ByVal e As
C1.Data.RowUpdateEventArgs) Handles table_Orders.BeforeUpdateRow
    e.Row["OrderID"] = AllocateNewRowID();
End Sub
```

To write code in C#

C#

```
private void table_Orders_BeforeUpdateRow(object sender, C1.Data.RowUpdateEventArgs
e)
{
    e.Row["OrderID"] = AllocateNewRowID();
}
```

Working with ADO.NET Dataset

A C1DataSet component stores its data in an ADO.NET DataSet and that internal storage is accessible via the StorageDataSet property. This enables a powerful combination of **DataObjects for .NET** and ADO.NET in the same code, see DataObjects for .NET and ADO.NET.

You can see an example of these techniques used in code in the ADOStorage sample in the **ComponentOne Samples** directory. The sample project has extensive explanatory comments in the code.

This functionality allows you to access and modify the ADO.NET data set storing table data. You can use both **DataObjects for .NET** and ADO.NET to access the same data. Essentially, you have two views to the same data. The first view is a **DataObjects for .NET** data set (C1DataSet), and the second is the corresponding ADO.NET data set (StorageDataSet). If you change data in one of them and then switch to the other, it will automatically reflect the changed data.

When you fill or modify a C1DataSet, **DataObjects for .NET** stores table data internally in an ADO.NET data set. You can access this data set at any time through the StorageDataSet property. This ADO.NET data set contains only simple table rows. For each simple table, you will find a table with the same name in the underlying ADO.NET data set (you can also access data tables with StorageDataTable and individual columns with StorageDataColumn). Table views are not stored in this ADO.NET data set, because they do not contain data as such, their rows only contain pointers to table rows; for more information, see Structured Data Storage: Tables and Table Views.

Using the StorageDataSet property, you can work directly with the ADO.NET data set or export data from a C1DataSet to an ADO.NET data set or to XML. So, the conversion from C1DataSet to ADO.NET DataSet is easy, just use the StorageDataSet property.

Conversion from an ADO.NET data set to C1DataSet is also possible, but it can require additional work, because we also need to set up table view rows, which are not stored in the ADO.NET data set. By "conversion process" process we mean modifying the underlying ADO.NET data set, StorageDataSet.

You can modify StorageDataSet using any means available in ADO.NET, and then "synchronize" it with the C1DataSet, that is, set up table view rows pointing to the table rows. The conversion, that is, modification/synchronization process has two stages:

First, you modify StorageDataSet (fill simple tables with data or modify their data as you need), and second, you fill

table views with rows that point to table rows.

Changing data in the ADO.NET data tables comprising StorageDataSet is only allowed on the first stage of this process. When the first stage ends, tables are filled with data. You must exercise caution not to modify ADO.NET table data directly except on the first stage of the modification/synchronization process (but of course you can modify it at any time through the regular **DataObjects for .NET** programmatic object model). **DataObjects for .NET** does not enforce this – it is developer's responsibility. Violating this restriction will not be detected by **DataObjects for .NET** and can have unpredictable consequences.

At the beginning of the second stage, all table views are cleared so you can fill them with rows (pointing to table rows). To fill a table view, use the **C1DataSet.SetTableViewRows** method. If you do not call **C1DataSet.SetTableViewRows** for a table view, it will be called automatically at the end of the second stage, with default parameters. So, if your table view contains all possible table rows, without restrictions (filters), you do not need to do anything with it on the second stage. But in many cases, such as when you used filter conditions filling C1DataSet, you need to control the collection of table rows that belong to table views. This is done using the C1DataSet.SetTableViewRows method for each such table view. You have two options: either to enumerate the rows manually (for instance, scanning the existent table rows, finding the ones that must belong to the table view, or to call GetDefaultTableViewRows that provides the full list of all existing table rows so you can filter them and then pass the filtered list to **C1DataSet.SetTableViewRows**.

The complete modification/synchronization process looks like this:

```
// start
c1DataSet1.StorageChangeBegin()
// fill or modify tables (modify data in StorageDataSet)
...first stage...
// first stage complete, tables filled, table views are
// undefined at this point
c1DataSet1.StorageChanged(true)
// fill table views with rows pointing to tables
// (calling SetTableViewRows())
...second stage...
// process complete, both tables and table views filled
c1DataSet1.StorageChangeEnd()
```

Note: For a complete example see the **ADOStorage** sample, which is installed with the **Studio for WinForms** samples.

DataObjects for .NET Enterprise Edition Design-Time Support

DataObjects for .NET provides customized context menus, smart tags, and a designer that offers rich design-time support and simplifies working with the object model. The following sections describe how to use **DataObjects for .NET**'s design-time environment to configure the **DataObjects for .NET** components.

Tasks Menus

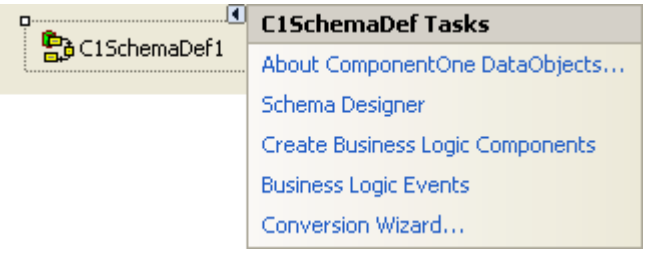
A smart tag represents a short-cut tasks menu that provides the most commonly used properties in each component. You can invoke each component's tasks menu by clicking on the smart tag (▶) in the upper-right corner of the component.

Properties Window

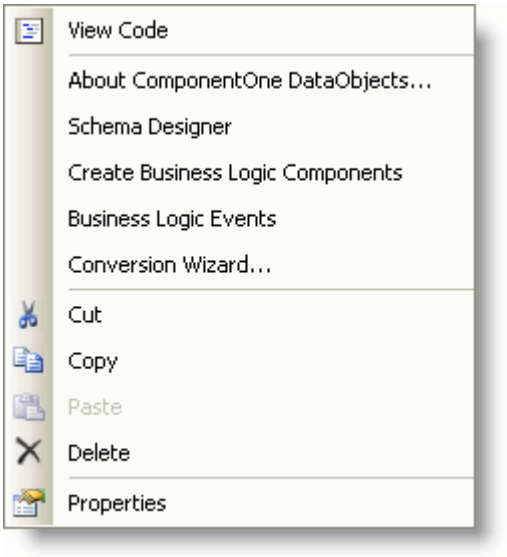
You can also easily configure **DataObjects for .NET** at design time using the Properties window in Visual Studio. You can access the Properties window by right-clicking the control and selecting **Properties**.

C1SchemaDef Tasks and Context Menus

You can access the **C1SchemaDef Tasks** menu by clicking the smart tag in the upper-right corner of the C1SchemaDef component.



You can access the C1SchemaDef context menu by right-clicking the C1SchemaDef component.



About DataObjects

Clicking **About** displays the **DataObjects for .NET**'s **About** dialog box, which is helpful in finding the build number of the component.

Schema Designer

Clicking **Schema Designer** opens the **DataObjects Schema Designer**. A Schema is the basis and starting point of **DataObjects for .NET** development. It contains data structure information, defining basic entities, such as tables and relations, with their properties. Normally, a schema is initially created by importing a database structure using the **Import Wizard** in the **Schema Designer**, and then customized in the **Schema Designer** to suit your business logic needs.

Create Business Logic Components

Clicking **Business Logic Components** creates a business logic component for each table and each data set in the schema. Every schema object can be represented by a special business logic component. Business logic components

(components C1.Data.C1TableLogic and C1.Data.C1DataSetLogic) have events where you can write code responding to various occurrences in data objects.

Business Logic Events

Clicking **Business Logic Events** displays the **Business Logic Events** dialog box. The **Business Logic Events** tool window shows the list of all tables and data sets. When you select a table in the tool window, the table's business logic events appear in the Properties window (in Visual C#, when the Events radio button is selected in the Properties window; in Visual Basic use the Method Name combo box in the code editor).

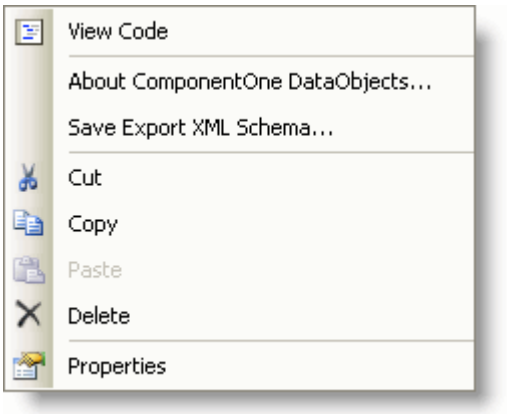
Conversion Wizard

Clicking **Conversion Wizard** brings up the **C1DataObjects Conversion Wizard** which allows you to import schema information. See Importing Schema Information using the Conversion Wizard for more information.

C1DataSet Tasks and Context Menus

You can access the **C1DataSet Tasks** menu by clicking the smart tag in the upper-right corner of the C1DataSet component.

You can access the C1DataSet context menu by right-clicking the C1DataSet component.



About DataObjects

Clicking **About** displays the **DataObjects for .NET**'s **About** dialog box, which is helpful in finding the build number of the component.

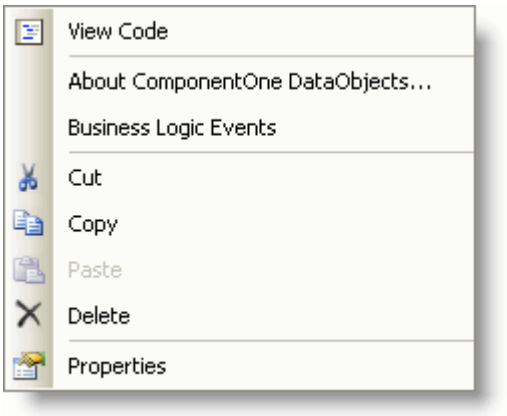
Save Export XML Schema

Clicking **Save Export XML Schema** opens the **Save schema to a file** dialog box which you can use to export data from a C1DataSet to XML. See Exporting Data from a C1DataSet to XML for more information.

C1SchemaRef Tasks and Context Menus

You can access the **C1SchemaRef Tasks** menu by clicking the smart tag in the upper-right corner of the C1SchemaRef component.

You can access the C1SchemaRef context menu by right-clicking the C1SchemaRef component.



About DataObjects

Clicking **About** displays the **DataObjects for .NET**'s **About** dialog box, which is helpful in finding the build number of the component.

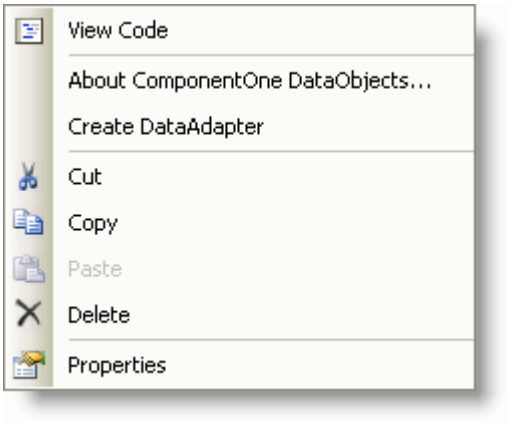
Business Logic Events

Clicking **Business Logic Events** displays the **Business Logic Events** dialog box. The Business Logic Events tool window shows the list of all tables and data sets. When you select a table in the tool window, the table's business logic events appear in the Properties window (in Visual C#, when the Events radio button is selected in the Properties window; in Visual Basic use the Method Name combo box in the code editor).

C1TableLogic Tasks and Context Menus

You can access the **C1TableLogic Tasks** menu by clicking the smart tag in the upper-right corner of the C1TableLogic component.

You can access the C1TableLogic context menu by right-clicking the C1TableLogic component.



About DataObjects

Clicking **About** displays the **DataObjects for .NET**'s **About** dialog box, which is helpful in finding the build number of the component.

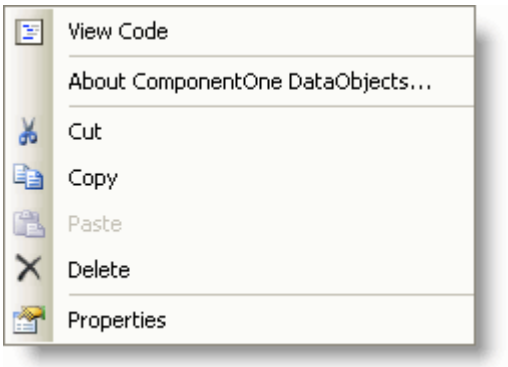
Create DataAdapter

Clicking **Create DataAdapter** creates a DataAdapter in a C1TableLogic component associated with the table. The DataAdapter component will then perform both fetch and update without custom code (but you can customize the default fetch and update behavior in event code if needed).

C1DataTableSource Tasks and Context Menus

You can access the **C1DataTableSource Tasks** menu by clicking the smart tag in the upper-right corner of the **C1DataTableSource** component.

You can access the **C1DataTableSource** context menu by right-clicking the **C1DataTableSource** component.



About DataObjects

Clicking **About** displays the **DataObjects for .NET**'s **About** dialog box, which is helpful in finding the build number of

the component.

DataObjects for .NET Tutorials

The following tutorials demonstrate a variety of **DataObjects** concepts and features, including working with schemas, business logic, application configurations, and using large datasets.

> **Note:** If you are running the pre-built tutorial projects included in **DataObjects for .NET** installation, please be aware that the projects have the sample database location hard coded in the connection string. If you have the Northwind database (standard MS Access sample database included in Visual Studio) installed in a different location, you can change the connection string or copy the **C1NWind.mdb** file to the required location.

Tutorial 1: Creating a Data Schema

In this tutorial, you'll learn the basics of working with **DataObjects for .NET**, including how to:

- Import a database structure so it becomes a *data schema*, the basis of all **DataObjects for .NET** activities.
- Create *composite tables*, to better shape and represent data to meet the needs of your users.
- Specify business logic in the form of calculation expressions and constraints. Note that in this tutorial, you'll specify business logic using *calculated expressions* and *constraints*. More business logic features will be introduced in other tutorials.
- Create data set definitions (DataSetDef) consisting of TableView objects exposing data to the user.
- Bind graphic user interface (GUI) controls to a **DataObjects for .NET** data source.

In this tutorial you'll use two different data bound grid controls, **TrueDBGrid for .NET** and **Microsoft DataGridView**, to demonstrate that **DataObjects for .NET** can serve as a data source to any data-bound GUI controls adhering to .NET databinding specifications. All other tutorials will use only ComponentOne grid controlsas they are most closely integrated with **DataObjects for .NET**. Most **DataObjects for .NET** features, all features that rely only on standard .NET data binding, will work with any third-party data-bound control. However, some **DataObjects for .NET** features that are extensions of standard .NET data binding, most notably, Virtual Mode (see Tutorial 4: Virtual Mode: Dealing with Large Datasets), require that you use ComponentOne data bound controls.

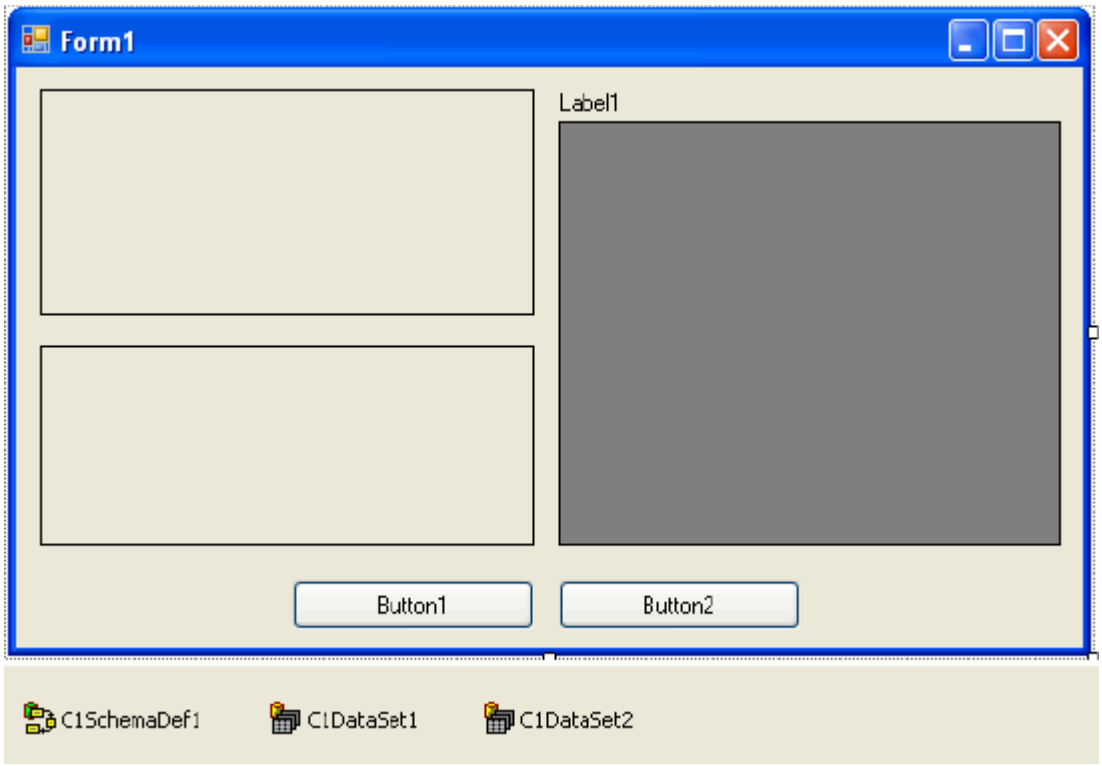
The first step in using **DataObjects for .NET** is creating a data *schema*. A schema is created using the **DataObjects Schema Designer**. Unlike in standard ADO.NET, a schema and associated business logic can be created once and then reused throughout your projects. Normally, you create a new schema by importing a database structure using the schema designer's **Import Wizard**. Alternatively, you can import an existing ADO.NET schema. See Converting Schema from Other Sources for more information.

To set up the project and create a schema:

1. Create a new Windows Application project.
2. Place the following components on the form as shown in the image below.

Number of Components	Names	Namespace
1 C1SchemaDef	C1SchemaDef1	C1.Data.C1SchemaDef
2 C1DataSet	C1DataSet1 C1DataSet2	C1.Data.C1DataSet
2 C1TrueDBGrid	C1TrueDBGrid1 C1TrueDBGrid2	C1.Win.C1TrueDBGrid.C1TrueDBGrid

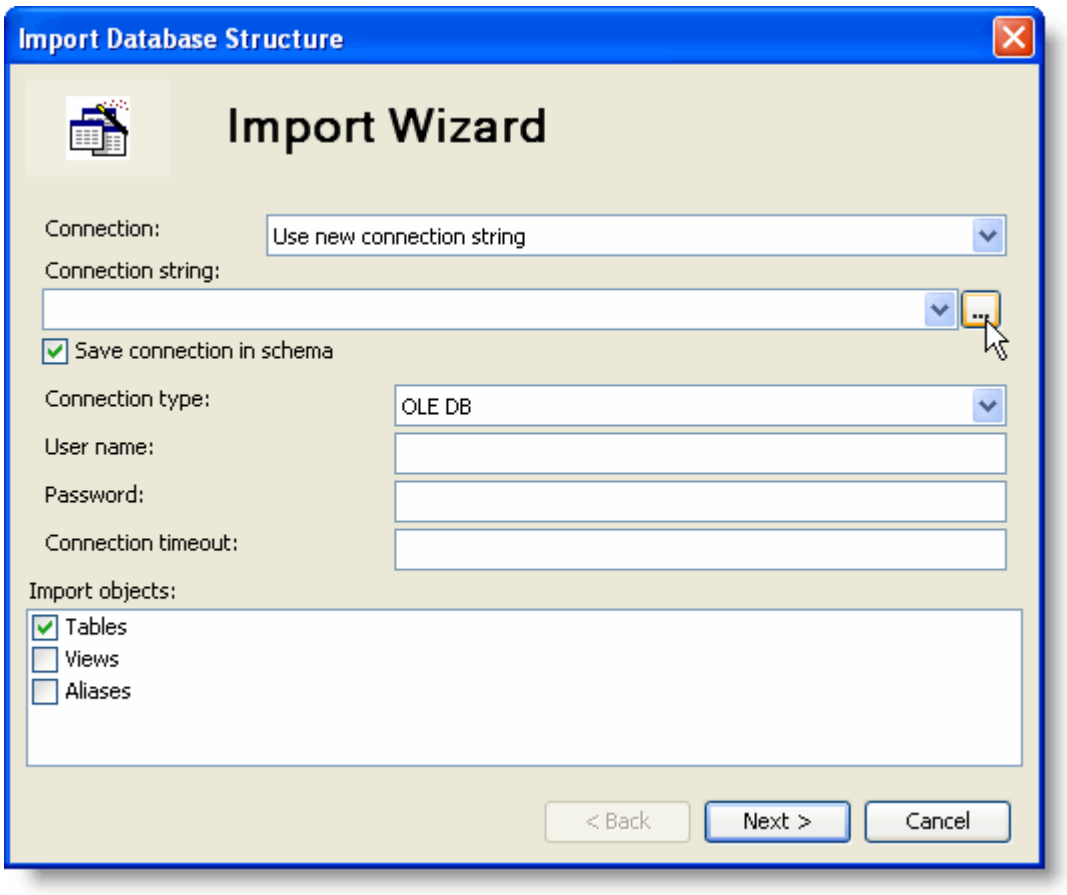
1 Label	Label1	System.Windows.Forms.Label
1 DataGridView	DataGridView1	System.Windows.Forms.DataGridView
2 command Buttons	Button1 Button2	System.Windows.Forms.Button



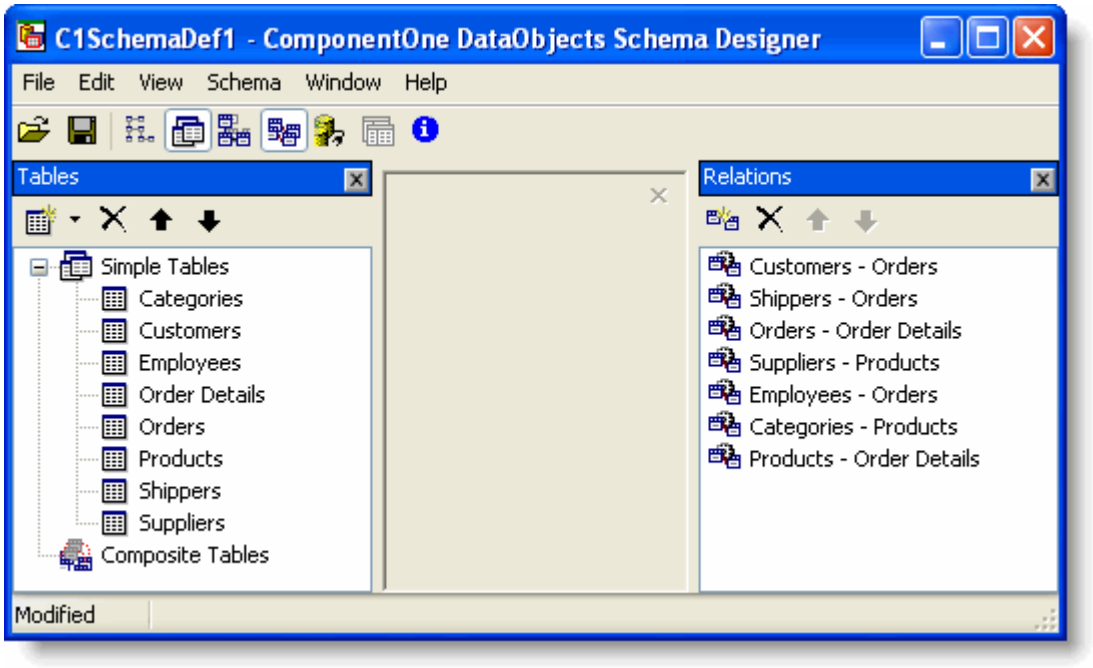
3. Set the **Text** property of **Button1** to "Commit Changes" and the **Text** property of **Button2** to "Products and Orders".
4. Select the **C1SchemaDef1** control, use the smart tags to expand the **C1SchemaDef Tasks** menu and select **Schema Designer**.

The **Schema Designer** opens, and the **Import Wizard** appears.

> **Note:** Select **Schema | Import** database structure in the **Schema Designer** to open the **Import Wizard** if it does not automatically appear.

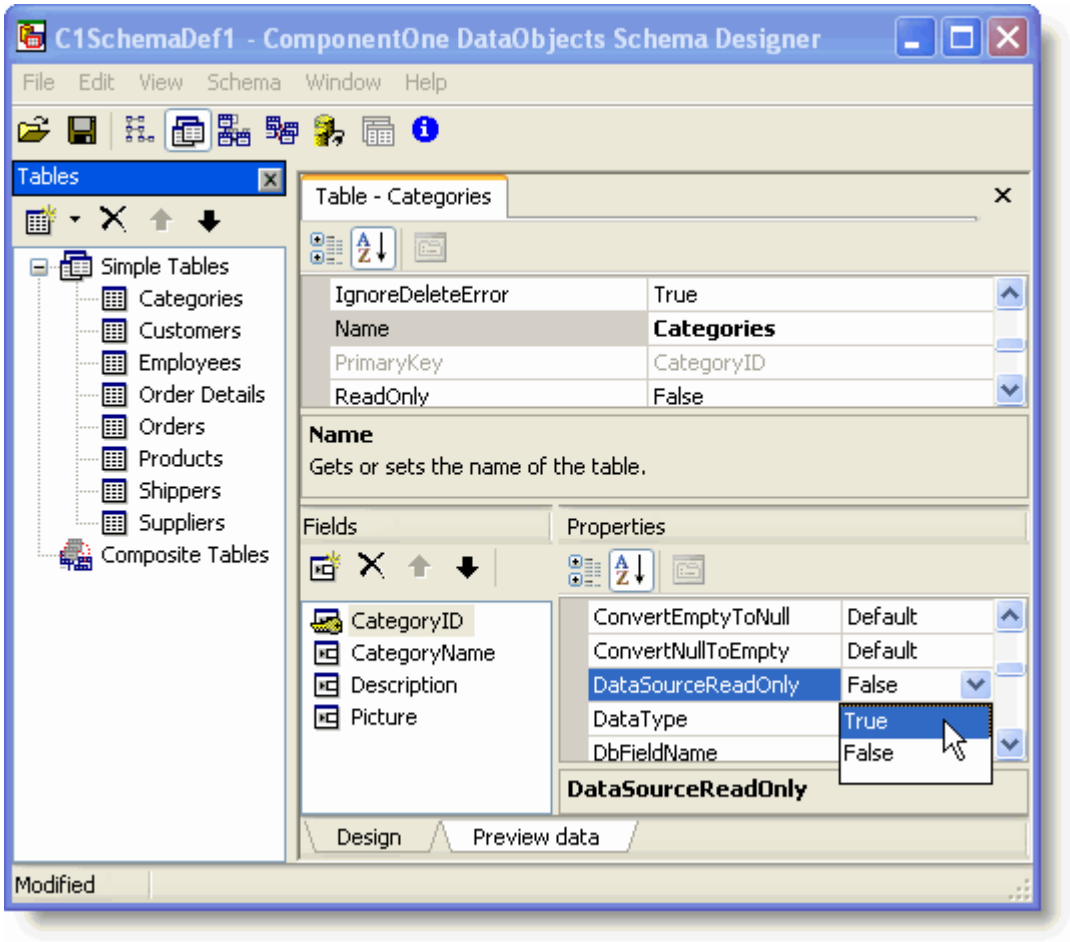


5. In the **Import Wizard** dialog, click the **ellipsis** button ... to the right of the connection string. The standard OLE DB **Data Link Properties** dialog box opens.
6. Select the provider, the database and other necessary connection properties in that dialog box. In this tutorial, the standard MS Access Northwind sample database (C1NWind.mdb) is used.
    1. Click the **Provider** tab, if necessary, and select **Microsoft Jet 4.0 OLE DB Provider**.
    2. Click the **Connection** tab and then click the **ellipsis** button under **Select or enter a database name**.
    3. Locate the **C1NWind.mdb** database, installed by default in the **Common** folder in the **ComponentOne Samples** directory, and click **Open**.
    4. Click **OK** to close the **DataLink Properties** dialog box. The connection string is imported.
7. Click **Next**. In this window, you can select the tables to import into the schema.
8. Click the >> button to select all available tables and then click **Finish**. The **Import Wizard** creates the schema and closes.



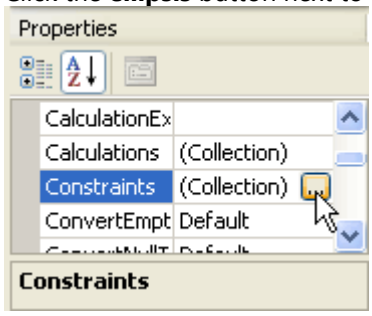
In the **Schema Designer**, the **Tables** window contains the list of all simple tables created by the **Import Wizard** based on the database tables. The **Relations** window contains inter-table relations created by the **Import Wizard** based on the relationships existing in the database.

9. Double-click the **Categories** table in the **Tables** window and select the **CategoryID** field from the **Fields** list in the lower panel of the **Table Editor**.
10. Set its DataSourceReadOnly property to **True**. Note that modifying table properties is only necessary when you use Microsoft Access as your database, since the MS Access OLE DB provider does not consider Autoincrement fields (row identity fields; their values are automatically assigned and maintained by the database) as read-only.

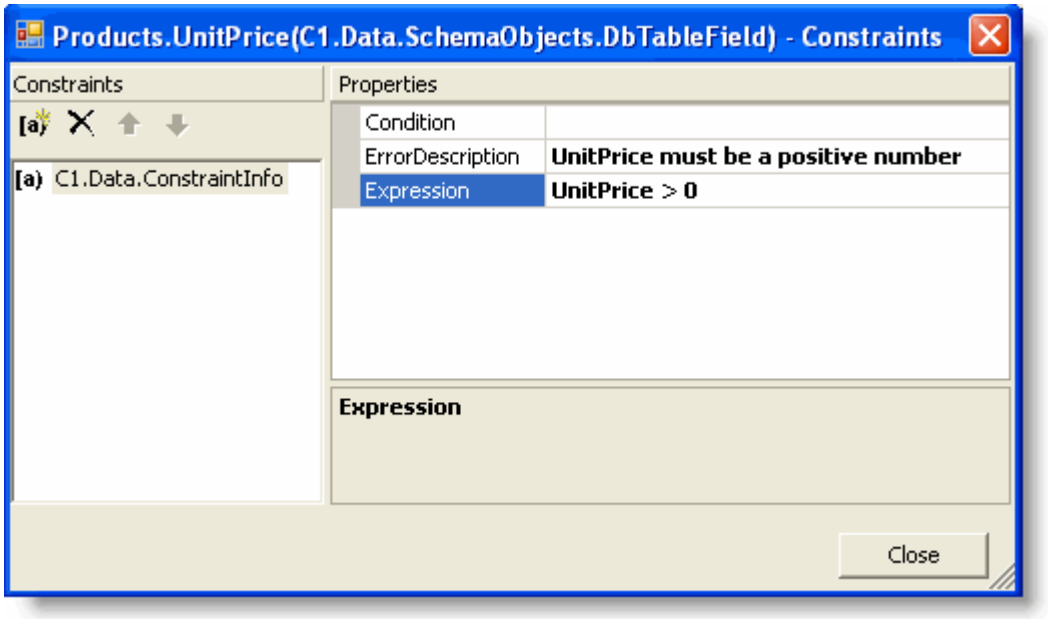


> **Note:** Switch between the **Design** and **Preview data** views of the tables using the tabs at the bottom of the editor window or by using the Shift and F5 keys; press F5 to move from **Design** to **Preview data** view or press Shift+F5 to move from **Preview data** to **Design** view.

11. Double-click the **Products** table in the **Tables** window and select the **UnitPrice** field from the **Fields** list in the lower panel of the **Table Editor**.
12. Click the **ellipsis** button next to the Constraints property to open the **Constraints Editor**.



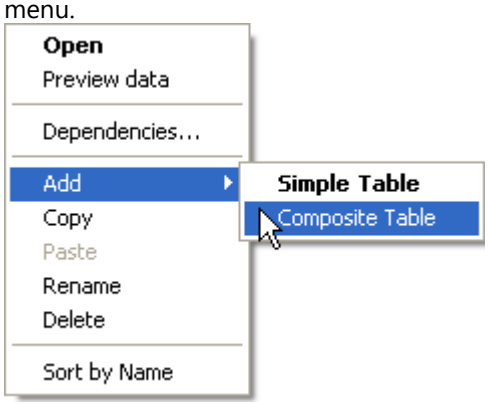
13. Click the **Add** button . A constraint is added to the list, and its properties appear in the right pane of the **Constraints Editor**. Note that a field can have multiple constraints.
    1. Type the following expression in the Expression property:
       UnitPrice > 0
    2. Add the following string in the ErrorDescription property:
       UnitPrice must be a positive number



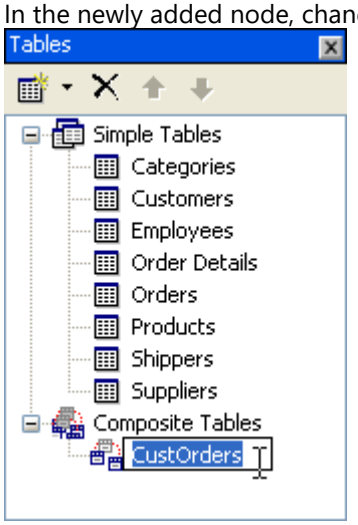
14. Click **Close**. The constraint expression for the **UnitPrice** field of the **Products** table is specified, and now we will create several composite tables.

To create composite tables:

1. In the **Schema Designer**, right-click the **Tables** window and select **Add | Composite Table** from the context menu.

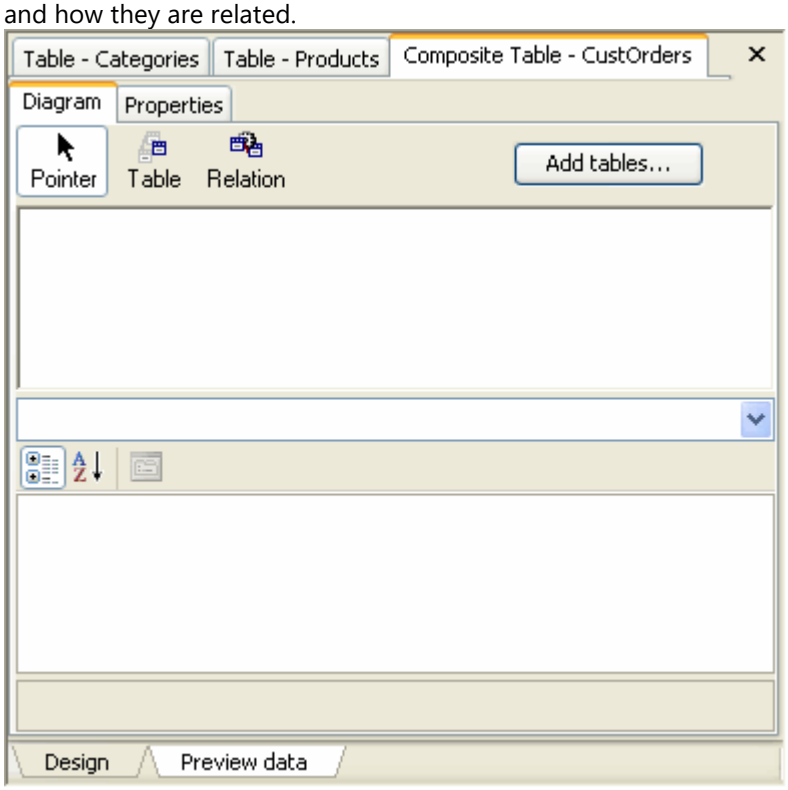


2. In the newly added node, change the default name from *CompositeTable* to "CustOrders".

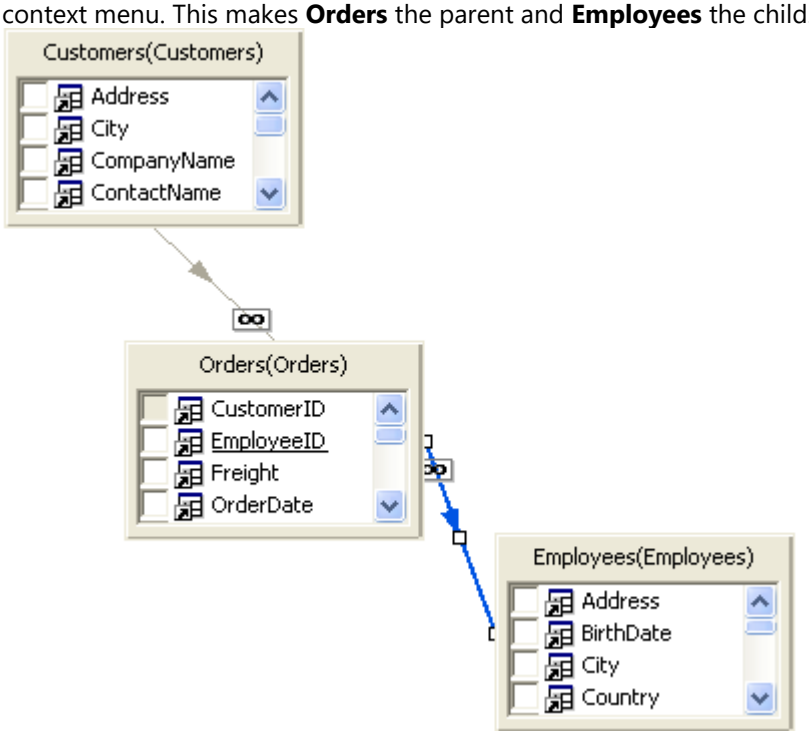


The **Composite Table Editor** opens when a new composite table is created. The editor has two tabs: **Diagram**

and **Properties**. Using the **Diagram** tab, you can specify which simple tables are used in the composite table and how they are related.



3. Click the **Add tables** button at the top of the editor. The **Add Tables** dialog box opens.
    1. Select **Customers**,**Orders**and**Employees** in the **Existing tables** window and click the > button to move them to the **Selected tables** window.
    2. Click **OK**. A diagram is created and consists of three tables connected by two relations. Note that it may be necessary to expand your **Diagram** display pane to see the tables.
4. The relation between **Orders** and **Employees** must be inverted, because we need an employee record (child record) to be uniquely determined by an order record (parent record); currently **Employees** is the parent and **Orders** is the child. Right-click the *Orders – Employees* relation on the diagram, and select **Invert** from the context menu. This makes **Orders** the parent and **Employees** the child.



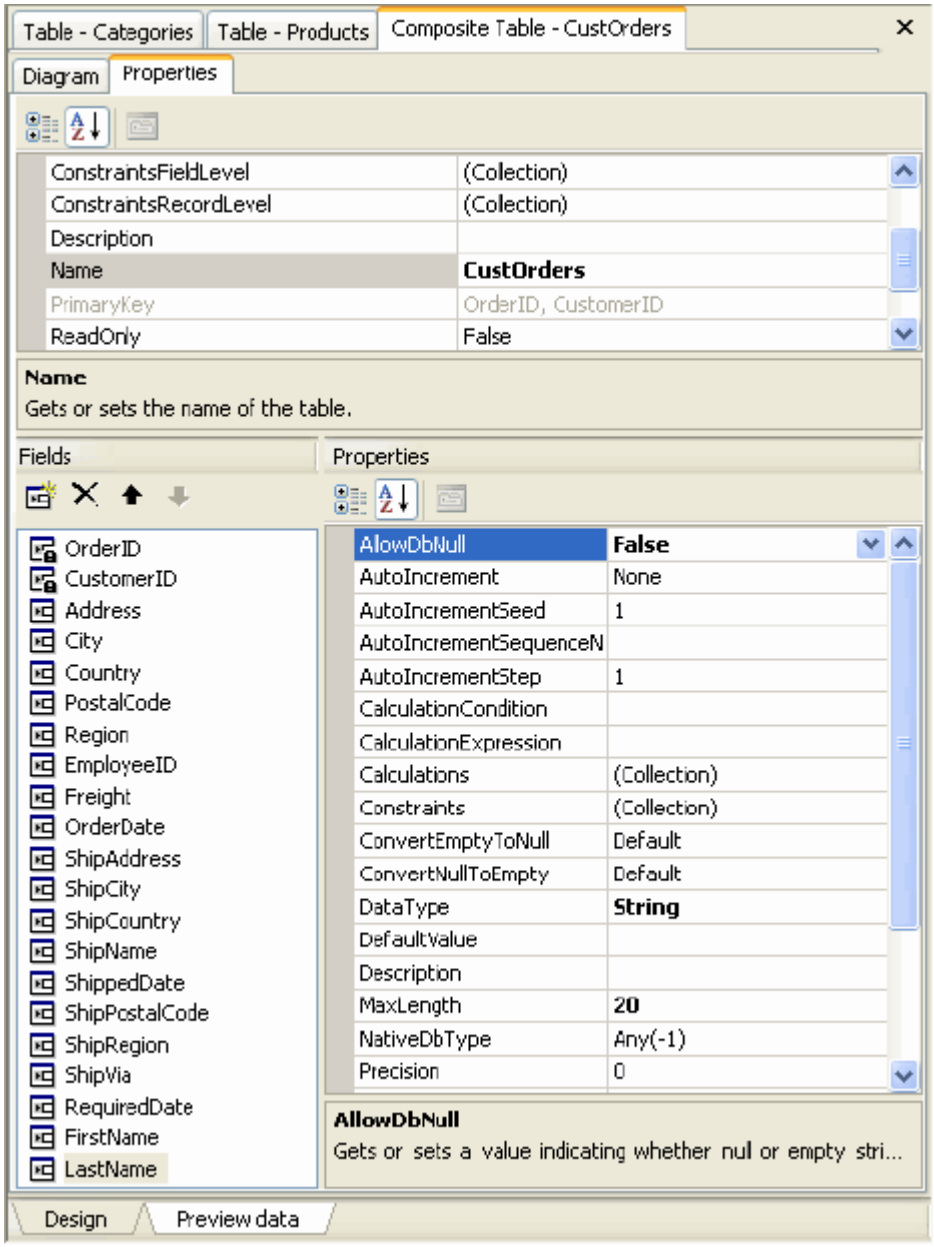
5. Rename the relation correspondingly: with the relation selected, set its Name property to *Orders – Employees* in the property grid below the diagram.
   The structure of the composite table **CustOrders** shown on the diagram can be represented in the following notation:
   Customers →(1-∞) Orders →(∞-1) Employees

   Here →(1-∞) is a one-to-many relation, and →(∞-1) is a many-to-one relation. Relation cardinality, either *OneToMany* or *ManyToOne*, can be seen on the diagram (the infinity symbol designates the *many* part) and in the property grid when the relation arrow is selected.

   According to relation cardinality, each row of the **CustOrders** composite table consists of a *Customers* row, an *Orders* row (one of the child rows of the *Customers* row) and an *Employees* row (the single child row uniquely determined by the *Orders* row).

6. To create the fields in our new **CustOrders** composite table, select fields in the constituent tables in the diagram:
   1. Select the **Customers** table and check the checkboxes for the following fields: *Address*, *City*, *Country*, *CustomerID*, *PostalCode*, and *Region*.
   2. Select the **Orders** table and check the checkboxes for all fields except *CustomerID*. The *CustomerID* is used in the *Customers – Orders* relation to connect an order record to a customer record; it has no other purpose, and *CustomerID* is already selected in the **Customers** table.
   3. Select the **Employees** table and check the checkboxes for the following fields: *FirstName* and *LastName*. *EmployeeID* is not included for the same reason the *CustomerID* field was not included in the **Orders** table.

> **Note:** To view all fields and their properties, select the **Properties** tab in the **Composite Table Editor**. Each field that was checked appears in the lower panel.



7. In the same way we created the **CustOrders** table, create another composite table, **OrderDetailsProducts**, by combining the simple tables **Order Details** and **Products**.
8. Right-click the relation between **Products** and **Order Details** and select **Invert**.
9. Set the relation's Name property to *Order Details – Products* in the property grid below the diagram.
10. Select the **Order Details** table and check the checkboxes for all fields except *UnitPrice*.
11. Select the **Products** table and check the checkboxes for *ProductName* and *UnitPrice*.

    The resulting **OrderDetailsProducts** composite table has the following structure:

    ```
    Order Details →(∞-1) Products
    ```

    A row of the **OrderDetailsProducts** composite table consists of an *Order Details* row and a *Products* row (the single child row uniquely determined by the *Order Details* row).

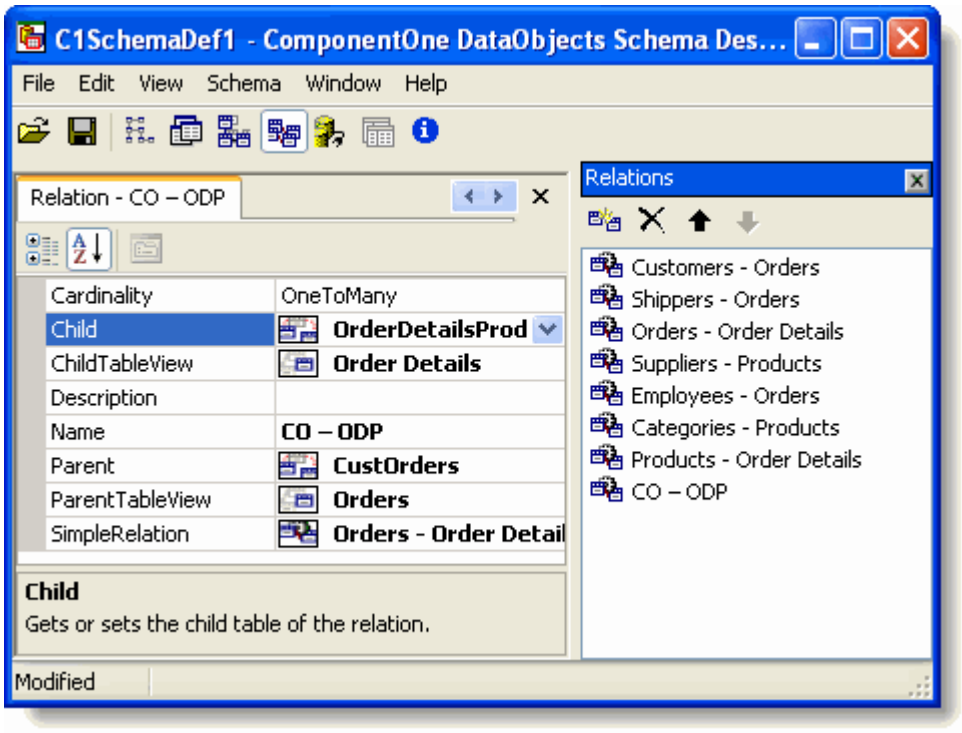
12. Add a new *calculated* field, *ExtendedPrice*, to the **OrderDetailsProducts** composite table:
    1. Select the **Properties** tab of the **Composite Table Editor**.
    2. Right-click the **Fields** list and select **Add** from the context menu. A new *CompositeTableField* is added to the list.
    3. Change the default name to *ExtendedPrice*.

4. With the *ExtendedPrice* field selected, enter the following value in the CalculationExpression property:
   ```
   Quantity * UnitPrice * (1 - Discount)
   ```
5. Set the DataType property to *Decimal*.

> **Note:** Calculated fields can be added to simple tables as well as to composite tables. See Table Fields for details.

13. We are going to create a composite relation between the **CustOrders** and **OrderDetailsProducts** composite tables. Right-click anywhere in the **Relations** window and select **Add** from the context menu. The **Relation Editor** appears.
14. Rename the relation *CO – ODP*, short for *CustOrders – OrderDetailsProducts*.
15. Click the drop-down arrow next to the Parent property and select *CustOrders* from the list of all available tables.
16. Click the drop-down arrow next to the Child property and select *OrderDetailsProducts* from the list of all available tables.

    When you select a composite table in the Parent or Child property, the **Relation Editor** changes. The **JoinConditions** panel disappears, and the set of properties changes, all due to the fact that the relation is now a composite relation not a simple relation. A composite relation, or a relation between two tables, one of which is a composite table, is based on a simple relation that connects two tables, one belonging to the parent composite table and the other belonging to the child composite table.



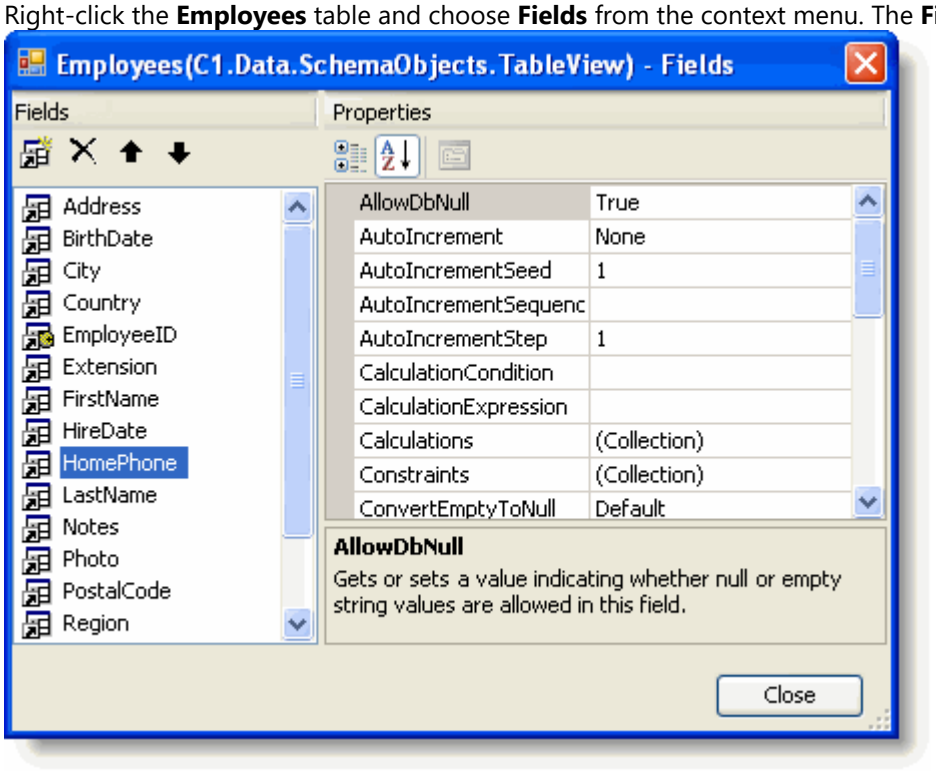
To create data sets:

1. In the **Schema Designer**, select **View | DataSets** so the **DataSets** window is visible.
2. Right-click anywhere in the **DataSets** window and select **Add** in the context menu.
3. Change the default name to *CustOrders*.
4. In the **DataSet Editor**, click the **Add tables** button. The **Add Tables** dialog box opens.
   1. Select **CustOrders**,**OrderDetailsProducts** and**Employees** in the **Existing tables** window and click the > button to move them to the **Selected tables** window. Note that **CustOrders**

and**OrderDetailsProducts** are composite tables. **Employees** is a simple table.

2. Click **OK**. A diagram is created and consists of three tables; **CustOrders** and **OrderDetailsProducts** are connected by a relation. Note that it may be necessary to expand your **Diagram** display pane to see the tables.

5. Right-click the **Employees** table and choose **Fields** from the context menu. The **Fields Editor** appears.



6. Select a field and use the **Move up** and **Move down** arrow buttons to re-arrange the field order.
7. Click **Close**.
8. To demonstrate how multiple data sets can exist in a schema, create one more data set with two composite tables. First, create the composite tables:
    1. Add a new composite table and name it **CustOrdersDetails**:
        - Click the **Add tables** button and add the following tables: **Customers**, **Orders**, **Order Details**,**Products** and **Categories**.
        - Invert the relation between **Products** and **Order Details** and rename it *Order Details – Products*.
        - Invert the relation between *Categories* and *Products*and rename it to *Products - Categories*.
        - Select the **Customers** table and check the checkboxes for the following fields: *Address*, *City*, *CompanyName*, *Country*, *CustomerID*, *PostalCode* and *Region*.
        - Select the **Orders** table and check the checkboxes for all fields except *CustomerID*.
        - Select the **Order Details** table and check the checkboxes for the following fields: *Discount*, *ProductID* and *Quantity*.
        - Select the **Products** table and check the checkboxes for the following fields: *CategoryID*, *ProductName* and *UnitPrice*.
        - Select the **Categories** table and check the checkboxes for the following fields: *CategoryName*.
          The resulting **CustOrdersDetails** composite table has the following structure:
          Customers →(1-∞) Orders →(1-∞) Order Details →(∞-1) Products →(∞-1) Categories

          A row of the **CustOrdersDetails** composite table consists of a *Customers* row, an *Orders* row (one of the child rows of the *Customers* row), an *Order Details* row (one of the child rows of the *Orders* row), a *Products* row (the single child row uniquely determined by the *Order Details* row) and a *Categories* row (the single child row uniquely determined by the *Products* row).

    2. Add a second composite table and name it **ProductsOrderDetailsCust**:
        - Click the **Add tables** button and add the following tables: **Customers**, **Orders**, **Order Details**, **Products** and **Categories**.

- Invert the relation between *Categories* and *Products*and rename it to *Products - Categories*.
- Invert the relation between *Orders* and *Order Details*and rename it to *Order Details - Orders*.
- Invert the relation between *Customers* and *Orders*and rename it to *Orders - Customers*.
- Select the **Products** table and check the checkboxes for the following fields: *CategoryID*, *ProductID*, *ProductName*, *UnitsInStock* and *UnitsOnOrder*.
- Select the **Categories** table and check the checkboxes for the following fields: *CategoryName*.
- Select the **Order Details** table and check the checkboxes for the following fields: *OrderID* and *Quantity*.
- Select the **Orders** table and check the checkboxes for the following fields: *CustomerID*.
- Select the **Customers** table and check the checkboxes for the following fields: *CompanyName*.

The resulting **ProductsOrderDetailsCust** composite table has the following structure:

```
Products →(1-∞) Order Details →(∞-1) Orders →(∞-1) Customers;
Products →(∞-1) Categories
```

Note the branching at the **Products** node. It has two children: **Order Details** connected with a one-to-many relation and **Categories** connected with a many-to-one relation. In general, branching in composite table diagrams is allowed, but only for many-to-one relations. One-to-many relations are not allowed to branch in a composite table, because a composite table should not form a branched tree; it must contain a set of rows having identical structure.

A row of the **ProductsOrderDetailsCust** composite table consists of a *Products* row, a *Categories* row (the single child row uniquely determined by the *Products* row), an *Order Details* row (one of the child rows of the *Products* row), an *Orders* row (the single child row uniquely determined by the *Order Details* row) and the *Customers* row (the single child row uniquely determined by the *Orders* row).

9. Finally, create a DataSet, *ProductsOrders*, including two composite tables, **CustOrdersDetails** and **ProductsOrderDetailsCust**:
    1. Right-click anywhere in the **DataSets** window and select **Add** in the context menu.
    2. Change the default name to *ProductsOrders*.
    3. In the **DataSet Editor**, click the **Add tables** button. The **Add Tables** dialog box opens.
    4. Select **CustOrdersDetails** and**ProductsOrderDetailsCust** in the **Existing tables** window and click the > button to move them to the **Selected tables** window.
    5. Click **OK**. A diagram is created and consists of two tables. Note that it may be necessary to expand your **Diagram** display pane to see the tables.
10. The schema is ready. Select **File | Save As**, enter **Schema1** in the **File name** text box and click **Save**. This schema will be used in other tutorial projects.
11. Close **Schema Designer** and click **Yes** if asked whether you want to save changes. The schema is saved in the **C1SchemaDef1** component (in the form resource file).
12. In your project, set the SchemaDef property of **C1DataSet1** and **C1DataSet2** to **SchemaDef1**. This connects the data set components to the schema.
13. Choose the data set exposed by each of the two components:
    1. Set the DataSetDef property of **C1DataSet1** to *CustOrders*.
    2. Set the DataSetDef property of **C1DataSet2** to *ProductsOrders*.
14. A client application usually needs some subset of data fetched to a data set, so we need some means of restricting, or filtering, data as it is retrieved from the database. This can be done by using FilterConditions. In this tutorial, we will restrict product data to only those products that have *CategoryID* = 1 (beverages) in **C1DataSet2**, the *ProductsOrders* data set, using the BeforeFill event. C1DataSet1, the *CustOrders* data set, will remain unrestricted. Add the following code to create the **C1DataSet2_BeforeFill** event handler to restrict product data:

To write code in Visual Basic

Visual Basic

```
Private Sub C1DataSet2_BeforeFill(ByVal sender As Object, ByVal e As
C1.Data.FillEventArgs) Handles C1DataSet2.BeforeFill
    Dim dataSetDef As C1.Data.SchemaObjects.DataSetDef
    dataSetDef = e.DataSet.Schema.DataSetDefs("ProductsOrders")
    e.Filter.Add(New C1.Data.FilterCondition(dataSetDef.TableViews_
   ("ProductsOrderDetailsCust"), "[CategoryID] = 1"))
    e.Filter.Add(New C1.Data.FilterCondition(dataSetDef.TableViews _
   ("CustOrdersDetails"), "[CategoryID] = 1"))
End Sub
```

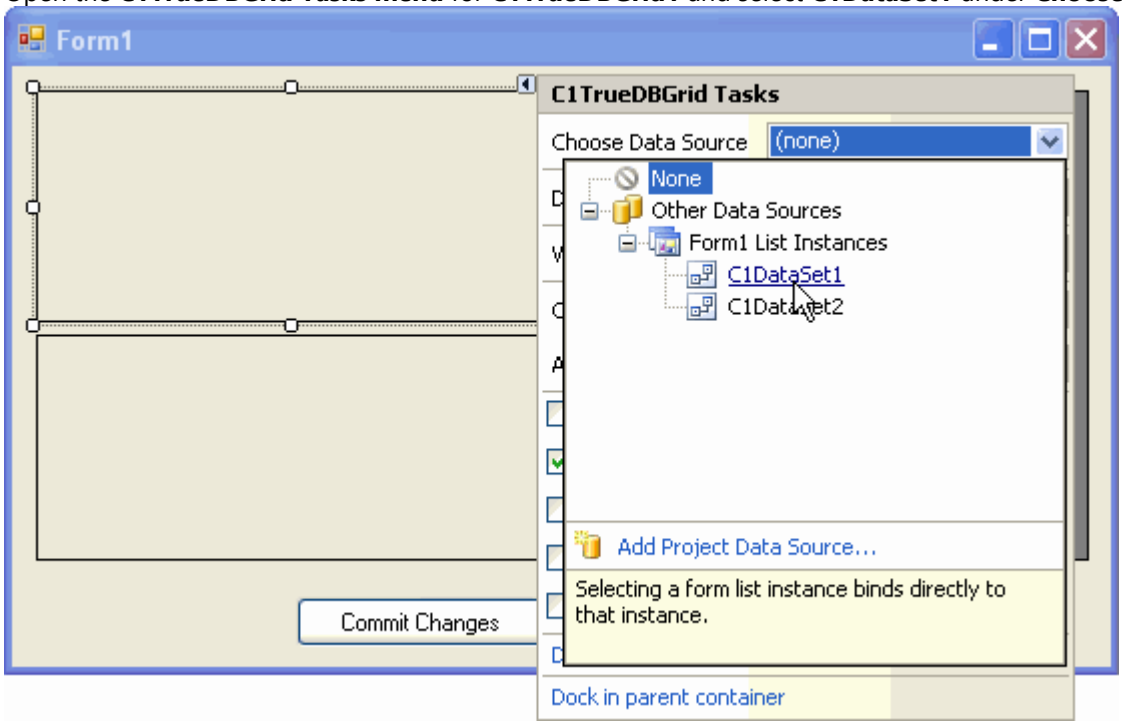
To write code in C#

C#

```
private void c1DataSet2_BeforeFill(object sender, C1.Data.FillEventArgs e)
{
    C1.Data.SchemaObjects.DataSetDef dataSetDef =
e.DataSet.Schema.DataSetDefs["ProductsOrders"];
    e.Filter.Add(new C1.Data.FilterCondition(dataSetDef.TableViews
   ["ProductsOrderDetailsCust"], "[CategoryID] = 1"));
    e.Filter.Add(new C1.Data.FilterCondition(dataSetDef.TableViews
   ["CustOrdersDetails"], "[CategoryID] = 1"));
}
```

To bind GUI controls to a DataObjects for .NET data source:

1. Open the **C1TrueDBGrid Tasks menu** for **C1TrueDBGrid1** and select **C1DataSet1** under **Choose DataSource**.



2. Set the **Caption** property to **Customers**.
3. Open the **C1TrueDBGrid Tasks menu** for **C1TrueDBGrid2** and select **C1DataSet1** under **Choose DataSource**.
4. Set the **Caption** property to **Orders**.
5. Set the **DataSource** property of **DataGridView1** to **C1DataSet1**.

6. Set the **Text** property of **Label1** to **Employees**.
7. Set the **DataMember** property for each grid as follows and click **Yes** to replace the existing column layout:
   - C1TrueDBGrid1.**DataMember** = _CustOrders
   - C1TrueDBGrid2.**DataMember** = _CustOrders.CO - ODP
   - DataGridView1.**DataMember** = Employees

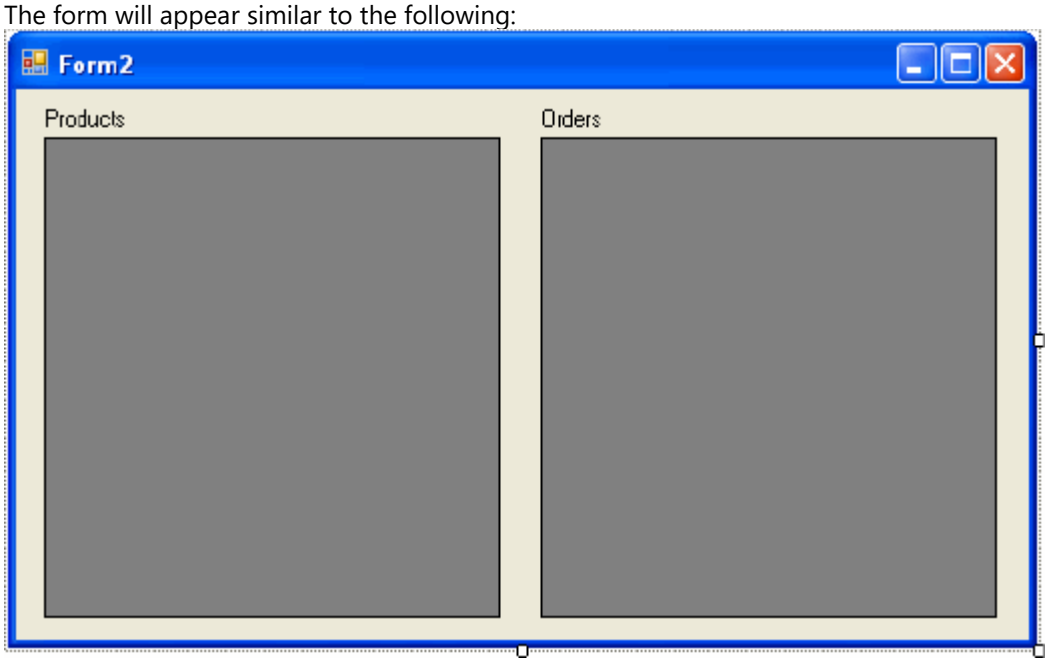
   > **Note:** Data members exposed by a **C1DataSet** data source represent the TableView objects of the data set. Those that have child table views, or relations, can be used in a master-detail hierarchy. They are represented by two data members, one with a leading underscore, the other without it. The data member without a leading underscore is used to connect to the table view as to an isolated data source, without master-detail hierarchy. The data member with a leading underscore is used to connect to it as to the root node, or master, of a master-detail hierarchy. Dependent nodes, or details, are represented by relation names, such as *_CustOrders.CO – ODP*.
8. In order to be able to send data modifications to the database, double-click Button1 to create the **Button1_Click** event and add the following code to the event:

   To write code in Visual Basic

   ```
   Visual Basic
   C1DataSet1.Update()
   ```

   To write code in C#

   ```
   C#
   c1DataSet1.Update();
   ```
9. The second data set, **ProductsOrders**, will be shown in a separate form. Select **Project | Add Windows Form**, choose **Windows Form** in the **Add New Item** dialog box and click **Add**. **Form2** is added to the project.
10. Add two DataGridView components, **DataGridView1** and **DataGridView2**, and two **Label** controls, **Label1** and **Label2** to the form.
11. Set the the **Text** properties for **Label1** and **Label2** to **Products** and **Orders**, respectively.
    The form will appear similar to the following:



12. Add the following method to the **Form2** code:

    To write code in Visual Basic

```
Visual Basic
Friend Sub SetDataSet(ByVal dataSet As C1.Data.C1DataSet)
    DataGridView1.DataMember = "ProductsOrderDetailsCust"
    DataGridView1.DataSource = dataSet
    DataGridView2.DataMember = "CustOrdersDetails"
    DataGridView2.DataSource = dataSet
End Sub
```

To write code in C#

```
C#
internal void SetDataSet (C1.Data.C1DataSet dataSet)
{
      dataGridView1.DataMember = "ProductsOrderDetailsCust";
      dataGridView1.DataSource = dataSet;
      dataGridView2.DataMember = "CustOrdersDetails";
      dataGridView2.DataSource = dataSet;
}
```

13. Go back to **Form1** and add the following code to the **Button2_Click** event. This will activate **Form2**.

To write code in Visual Basic

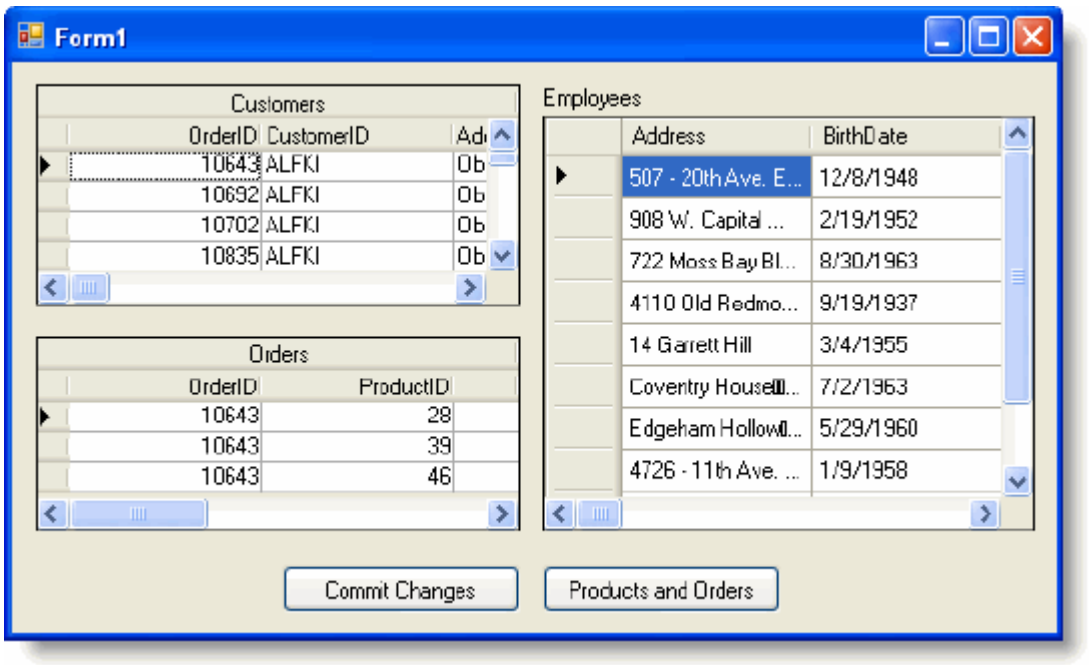
```
Visual Basic
Dim form2 As Form2
form2 = New Form2()
form2.SetDataSet(C1DataSet2)
form2.ShowDialog(Me)
```

To write code in C#

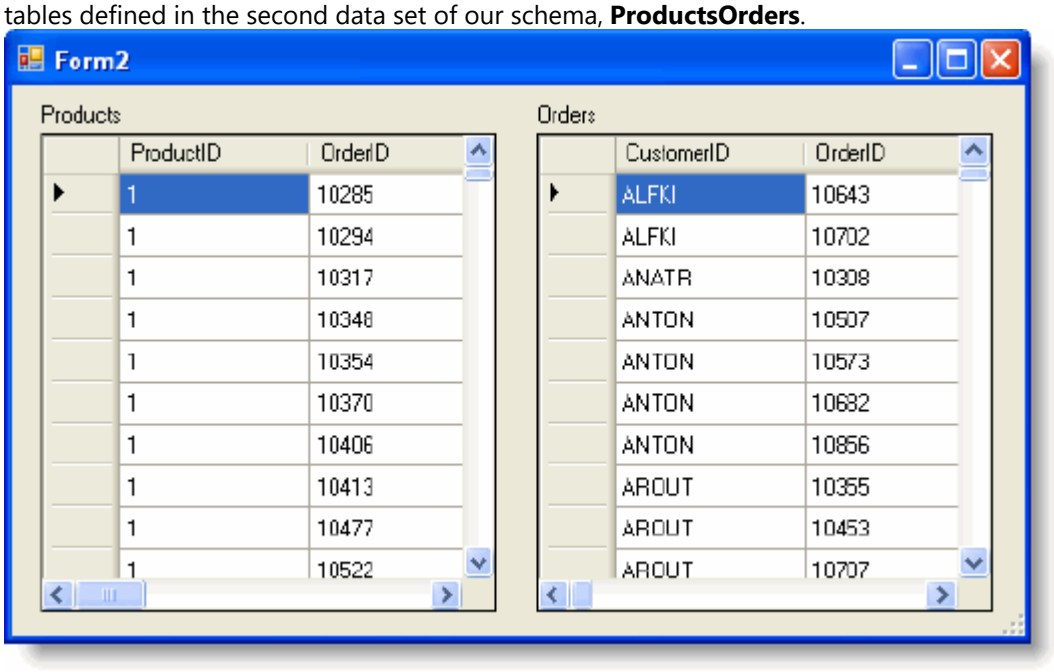
```
C#
Form2 form2 = new Form2();
form2.SetDataSet(c1DataSet2);
form2.ShowDialog(this);
```

Run the program and observe the following:

- The two **C1TrueDBGrid** controls on the left show rows of composite tables **CustOrders** and **OrderDetailsProducts** combining data from multiple database tables. Note that this was achieved without creating database views or writing complex SQL statements with joins. It demonstrates how **DataObjects for .NET** is used to facilitate database application development, playing the role of a data modeling and query building tool. The most compelling benefits of the **DataObjects for .NET**'s structured data model become apparent when you try modifying data. **DataObjects for .NET** does not merely present data as independent simple tables of rows. It maintains the specified structure automatically when you modify data – the task universally required in database applications and one that other data frameworks, including ADO.NET, fall short of supporting.



- To observe how **DataObjects for .NET** maintains underlying structure on data modifications, try changing the *EmployeeID* field in the first row of **C1TrueDBGrid1**, the **Customers** grid, from 6 to 1. The *FirstName* and *LastName* fields, dependent on *EmployeeID*, change correspondingly. Now go to **DataGridView1**, the **Employees** grid, and change the *LastName* of the employee with *EmployeeID*=1. All rows of the **Customers** grid that have *EmployeeID* = 1 reflect this change, showing the new *LastName* value. Alternatively, you can change the *LastName* or *FirstName* field in the **Customers** grid and see the change reflected in the **Employees** grid. Whatever the changes, **DataObjects for .NET** recognizes them so you can rely on proper structure being maintained for your data at all times.
- **DataObjects for .NET** supports master-detail hierarchy, as you can see in the **Customers** and **Orders** grids. The **Orders** grid shows the orders of the customer currently selected in the **Customers** grid. **DataObjects for .NET** also supports sorting: click a column header of a grid, and the data will be sorted by that column.
- In the **Orders** grid, you can see the *ExtendedPrice* column, which is a calculated field we defined in the schema.
- Try entering a negative number in the *UnitPrice* column of the **Orders** grid. Negative numbers are not allowed in accordance with the constraint we defined for the *UnitPrice* field of the **Products** table.
- Click **Button2**, the **Products and Orders** button. **Form2** opens with two grids showing the two composite tables defined in the second data set of our schema, **ProductsOrders**.



- Try entering a negative number in the *UnitPrice* column of the **DataGridView2**, or **Orders** grid, in Form2. Negative numbers are not allowed here as they are not allowed in Form1, because of the same constraint defined in the *UnitPrice* field of the **Products** table. This demonstrates a very important and powerful feature of **DataObjects for .NET**: business logic defined on table level is enforced in all composite tables and in all data sets where this table is used. You define business logic where it belongs, and do it only once, the rest is **DataObjects for .NET**'s responsibility.

Tutorial 2: Defining Business Logic

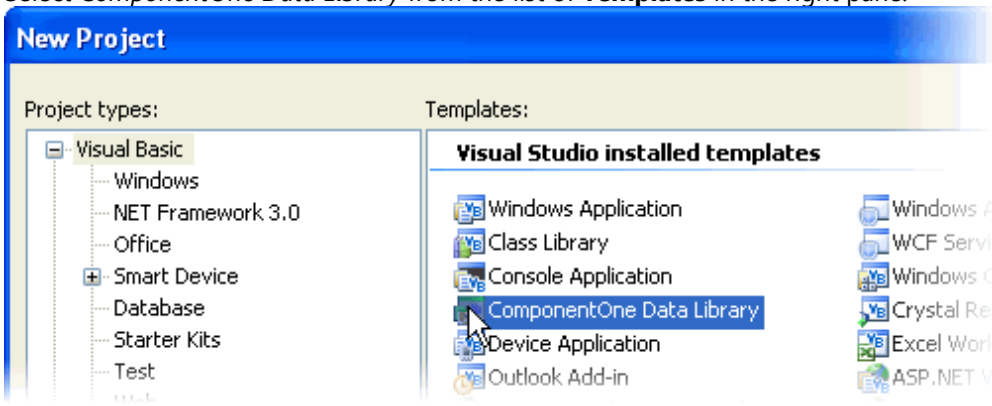
In this tutorial, you will learn how to:

- Create a data library that can be used in multiple projects.
- Define business logic for your data objects that is enforced by **DataObjects for .NET** wherever these objects are used.

In the previous tutorial, we defined and used data objects in the same project. This is not how **DataObjects for .NET** is supposed to be used. A better way to use **DataObjects for .NET** is to define your business objects (data objects) in a separate assembly, data library, so it can be used by multiple applications. Although this is not mandatory, your enterprise can assign a special team of "data-oriented" developers to the task of creating business object projects (data libraries) and another team of "GUI-oriented" developers to creating client applications using data libraries. This is how **DataObjects for .NET** enables you to benefit from the right design and development technology. However, **DataObjects for .NET** fits equally well into smaller development projects, where business objects and client applications are developed by the same team. The main architectural benefit of using **DataObjects for .NET** is in clear separation between business logic and presentation layer (GUI). Encapsulating your data model and logic in a centralized place (data library) that can be reused by multiple clients makes your applications robust, scalable and fun to develop. This is not to mention numerous tools and enhancements **DataObjects for .NET** adds to your data access toolbox, including such a powerful and extremely important one as virtual mode (cached access to large recordsets), see Tutorial 4: Virtual Mode: Dealing with Large Datasets.

A data library is an assembly (DLL) containing **DataObjects for .NET** schema and business logic components and code that defines your data objects. These data objects can then be used in any project by simply referencing the library in the project and using a **DataObjects for .NET** C1DataSet component to connect to the library. All database access and business logic code is encapsulated in the library, so it can be created and maintained independently of client applications.

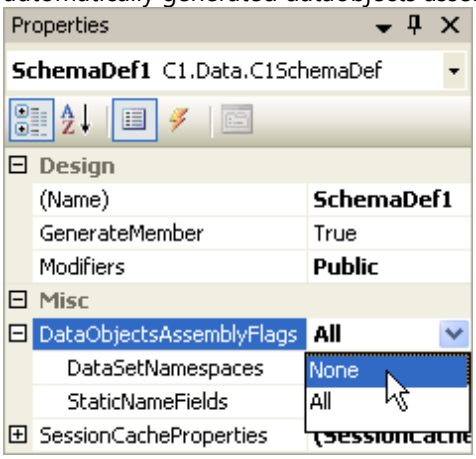
1. Create a new Data Library project:
    1. Select **File | New Project** in the Visual Studio menu, and in the **New Project** dialog box under **Project types**, choose either **Visual Basic** or **Visual C#**, according to your language preference. Note that one of these options may be located under **Other Languages**.
    2. Select *ComponentOne Data Library* from the list of **Templates** in the right pane.



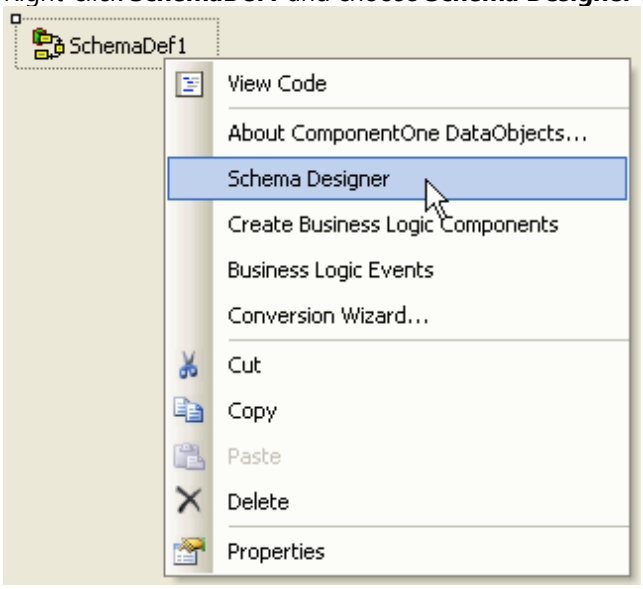
    3. Enter *Northwind* in the **Name** text box, specify a location and click **OK**.
    The *Northwind* data library project is created for you. The main file of the resulting data library project is **DataClass.vb** (.cs), a component class where you will host the schema and business logic. Note that in

large projects, it is also possible to distribute business logic code over multiple files. A **C1SchemaDef** component is automatically added to **DataClass**.

2. Select the **SchemaDef1** component and set the **DataObjectsAssemblyFlags** property to **None** in the **Properties** window. By doing this, we avoid creating a separate namespace for each dataset definition in the automatically generated dataobjects assembly.



3. Right-click **SchemaDef1** and choose **Schema Designer** from the context menu.



The **Schema Designer** opens and the **Import Wizard** appears.

4. We will use the schema created in Tutorial 1: Creating a Data Schema, so click **Cancel** to close the **Import Wizard**.
5. Select **File | Open** in the **Schema Designer** menu and open the schema file that was saved in Tutorial 1: Creating a Data Schema. The schema appears in the designer.
6. Close the **Schema Designer** and click **Yes** to save the changes.
7. Compile the data library project by selecting **Build | Build Solution**.

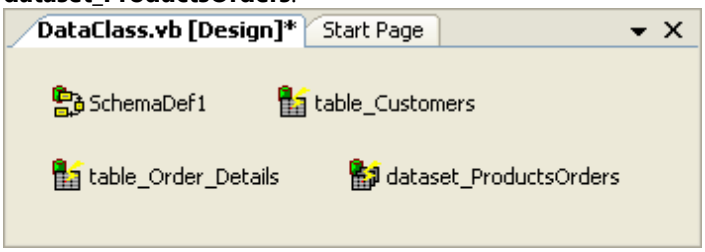
To define business logic:

With the schema now in place, we can specify business logic. You already saw some elements of business logic, namely constraint expressions and calculation expressions, in Tutorial 1: Creating a Data Schema. Expressions are an easy and straightforward way of defining business logic. However, when expressions are not enough, some code must be written. In this tutorial, we will show how to write business logic code.

Every schema object can be represented by a special business logic component. Business logic components (components C1.Data.C1TableLogic and C1.Data.C1DataSetLogic) have events where you can write code responding to various occurrences in data objects.

1. Right-click the **C1SchemaDef** component and select **Create Business Logic Components** from the context

menu. This creates a business logic component for each table and each data set in the schema. We only need three of them, so delete all but the following three components: **table_Customers**, **table_Order_Details**, and **dataset_ProductsOrders**.



Alternatively, you can create the three business logic components manually from the Visual Studio Toolbox by adding two C1TableLogic components and one C1DataSetLogic component. Then you must set each of their properties: set SchemaComponent to **SchemaDef1**, **C1TableLogic1**.Table to **Customers**, **C1TableLogic2**.Table to **Order Details** and **C1DataSetLogic1**.DataSetDef to **ProductsOrders**.

2. Add the following code to create an event handler **table_Customers_BeforeFieldChange** and convert the proposed *CustomerID* value to upper case:

To write code in Visual Basic

Visual Basic

```
Private Sub table_Customers_BeforeFieldChange(ByVal sender As Object, ByVal e As
C1.Data.FieldChangeEventArgs) Handles table_Customers.BeforeFieldChange
    If e.Field.Name = "CustomerID" Then
        e.NewValue = CStr(e.NewValue).ToUpper()
    End If
End Sub
```

To write code in C#

C#

```
private void table_Customers_BeforeFieldChange(object sender,
C1.Data.FieldChangeEventArgs e)
{
    if (e.Field.Name == "CustomerID")
        e.NewValue = ((string)e.NewValue).ToUpper();
}
```

This code will be executed each time the value of a field in the **Customers** table is about to change. As with all other **DataObjects for .NET** business logic events, it is triggered at any attempt to change a field of any object, be it a table view or a composite table, that would change a **Customers** table field. So the logic we put here will be enforced in any object in any context where the **Customers** table is involved.

3. Add the following code to create an event handler **table_Order_Details_BeforeFieldChange**:

To write code in Visual Basic

Visual Basic

```
Private Sub table_Order_Details_BeforeFieldChange(ByVal sender As Object, ByVal e
As C1.Data.FieldChangeEventArgs) Handles table_Order_Details.BeforeFieldChange
    If e.Field.Name = "Quantity" Then
        If (CShort(e.NewValue) < 1) Then
            e.NewValue = CShort(1)
        End If
```

```
    ElseIf e.Field.Name = "Discount" Then
        If CSng(e.NewValue) < 0 Or CSng(e.NewValue) > 1 Then
            Throw New ApplicationException("Discount must be between 0 and 1")
        End If
End If
End Sub
```

To write code in C#

C#

```
private void table_Order_Details_BeforeFieldChange(object sender,
C1.Data.FieldChangeEventArgs e)
{
    if (e.Field.Name == "Quantity")
    {
        if ((short)e.NewValue < 1)
            e.NewValue = (short)1;
    }
    else if (e.Field.Name == "Discount")
    {
        if ((float)e.NewValue < 0 || (float)e.NewValue > 1)
            throw new ApplicationException("Discount must be between 0 and 1");
    }
}
```

The code for *Quantity* turns a negative value to 1 before it is assigned to the *Quantity* field of the **Order Details** table.
The code for *Discount* tests a constraint **0 < Discount < 1** and shows a message if it is not satisfied. It is executed before changing the *Discount* value.

4. Add the following code to create an event handler **table_Order_Details_AfterFieldChange**:

To write code in Visual Basic

Visual Basic

```
Private Sub table_Order_Details_AfterFieldChange(ByVal sender As Object, ByVal e
As C1.Data.FieldChangeEventArgs) Handles table_Order_Details.AfterFieldChange
    If e.Field.Name = "Quantity" Then
        Dim orderDetail As Order_DetailsRow, product As ProductsRow
        Dim oldValue, unitsOrdered As Short
        orderDetail = Order_DetailsRow.Obj(e.Row)
    product = orderDetail.GetProductsRow()
        If Not (product Is Nothing) Then
            If e.OldValue Is Convert.DBNull Then
                oldValue = CShort(0)
            Else
                oldValue = CShort(e.OldValue)
            End If
            unitsOrdered = CShort(CShort(e.NewValue) - CShort(oldValue))
        End If
        product.UnitsInStock = CShort(product.UnitsInStock - unitsOrdered)
        If product.UnitsInStock < 0 Then
            product.UnitsInStock = 0
```

```
        End If
        product.UnitsOnOrder = product.UnitsOnOrder + unitsOrdered
    End If
End Sub
```

To write code in C#

C#

```
private void table_Order_Details_AfterFieldChange(object sender,
C1.Data.FieldChangeEventArgs e)
{
    if (e.Field.Name == "Quantity")
    {
        Order_DetailsRow orderDetail = Order_DetailsRow.Obj(e.Row);
        ProductsRow product = orderDetail.GetProductsRow();
        if (product != null)
        {
            short unitsOrdered = (short)((short)e.NewValue -
                (e.OldValue == Convert.DBNull ? (short)0 :
                (short)e.OldValue));
            product.UnitsInStock = (short)(product.UnitsInStock - unitsOrdered);
            if (product.UnitsInStock < 0)
                product.UnitsInStock = 0;
            product.UnitsOnOrder += unitsOrdered;
        }
    }
}
```

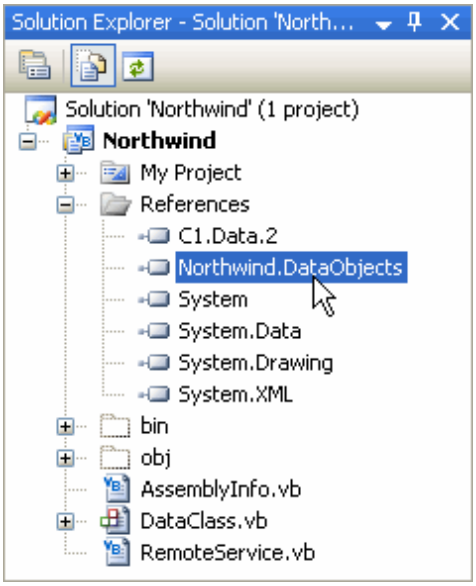
This code executes after the value of the *Quantity* field in the **Order Details** table has changed. It makes necessary changes in the corresponding **Products** row.

Click here for an explanation of some important features in this code.

There are two data object classes generated by **DataObjects for .NET** in this code: **Order_DetailsRow** and **ProductsRow**. **Class Order_DetailsRow** represents an **Order Details** row, and **ProductsRow** represents a **Products** row. Such classes are automatically generated and maintained for each table and table view in the schema. For example, **ProductsRow** is an object (business object, data object) where each field has a corresponding property (ProductsRow.**UnitPrice**, ProductsRow.**UnitsInStock**, and so on), and each relation has a corresponding method (Products.**GetOrder_DetailsRows**, and so on), allowing you to obtain child rows and the parent row.

Using these classes, you can write business logic code in a convenient, type-safe way, and benefit from Visual Studio code-completion features giving you the lists of properties and methods to choose from.

The data object classes belong to the **Northwind.DataObjects** namespace (substitute your data library name instead of **Northwind**, if different). They are hosted in the assembly **Northwind.DataObjects.dll**. This data objects assembly is generated by **DataObjects for .NET** each time you change the schema and save it in the **C1SchemaDef** component. A reference to this assembly is added to your data library project **References**.



5. Create an event handler **dataset_ProductsOrders_AfterEndAddNew** and enter the following code:

To write code in Visual Basic

Visual Basic

```
Private Sub dataset_ProductsOrders_AfterEndAddNew(ByVal sender As Object, ByVal e
As C1.Data.RowChangeEventArgs) Handles dataset_ProductsOrders.AfterEndAddNew
    If e.DataTable.SchemaTable.Name = "CustOrdersDetails" Then
        Dim order As CustOrdersDetailsRow_tableView
        order = CustOrdersDetailsRow_tableView.Obj(e.Row)
        If order.IsShipNameNull() Then order.ShipName = order.CompanyName
        If order.IsShipAddressNull() Then order.ShipAddress = order.Address
        If order.IsShipCityNull() Then order.ShipCity = order.City
        If order.IsShipRegionNull() Then order.ShipRegion = order.Region
        If order.IsShipPostalCodeNull() Then order.ShipPostalCode =
order.PostalCode
        If order.IsShipCountryNull() Then order.ShipCountry = order.Country
    End If
End Sub
```

To write code in C#

C#

```
private void dataset_ProductsOrders_AfterAddNew(object sender,
C1.Data.RowChangeEventArgs e)
{
    if (e.DataTable.SchemaTable.Name == "CustOrdersDetails")
    {
        CustOrdersDetailsRow_tableView order =
CustOrdersDetailsRow_tableView.Obj(e.Row);
        if (order.IsShipNameNull())
            order.ShipName = order.CompanyName;
        if (order.IsShipAddressNull())
            order.ShipAddress = order.Address;
        if (order.IsShipCityNull())
```

```
            order.ShipCity = order.City;
        if (order.IsShipRegionNull())
            order.ShipRegion = order.Region;
        if (order.IsShipPostalCodeNull())
            order.ShipPostalCode = order.PostalCode;
        if (order.IsShipCountryNull())
            order.ShipCountry = order.Country;
    }
}
```

This code executes after a new row has been added to the **CustOrdersDetails** table view, and all primary key fields have been specified, so the row is no longer a "temporary new row", meaning the primary key is yet undefined. This code fills shipping attributes given known values of billing attributes. Apart from another example of using data object classes in business logic code, this code shows yet another important **DataObjects for .NET** feature, specifically:
You can specify business logic on the table level, as shown in the previous examples, and then it will be executed wherever this table is involved. But you can also specify business logic on the dataset level, in other words, on the table view level, as in this example. Such code will be executed only in this dataset. Using dataset-level business logic, you can enforce rules that are specific to a certain data set.

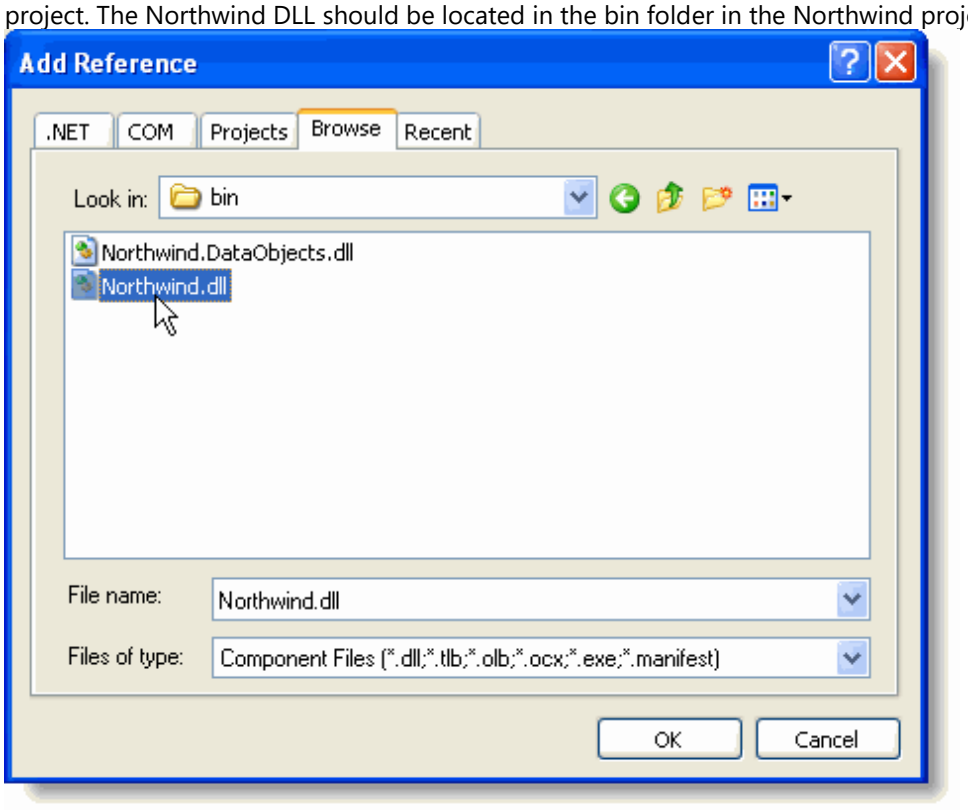
Click here for a list of business logic events.

This tutorial demonstrates only a limited number of business logic events. Here is a brief list of business logic events available in **DataObjects for .NET** (we omit prefixes Before and After pertaining to most events, retaining the prefix only if an event occurs only Before or only After):

Event	Description
AddNew	Fired when a new (empty) row is added.
AfterChanges	Fired when all changes initiated by a field change are done and handled by the business logic code, see the FieldChange event.
BeginEdit	Fired when the user starts editing a row (data-bound controls start editing a row immediately after they position on it, even though no changes have been made yet).
CancelEdit	Fired when the user cancels editing a row reverting the changes made to it.
Delete	Fired when a row is deleted.
EndAddNew	Fired when a newly added row becomes a regular row in the rowset. When a row is added, it is added empty, its primary key is unknown. A row with unknown primary key is in special transitory state, it is not a regular rowset row. Only after its primary key is set it becomes a regular (added) row, which is signaled by this event.
EndEdit	Fired when the user finishes editing a row (data-bound controls finish editing a row when they leave that row, even if no changes have been made).
FieldChange	Fired when a field value is set. Inside this event, your code can set other fields triggering recursive

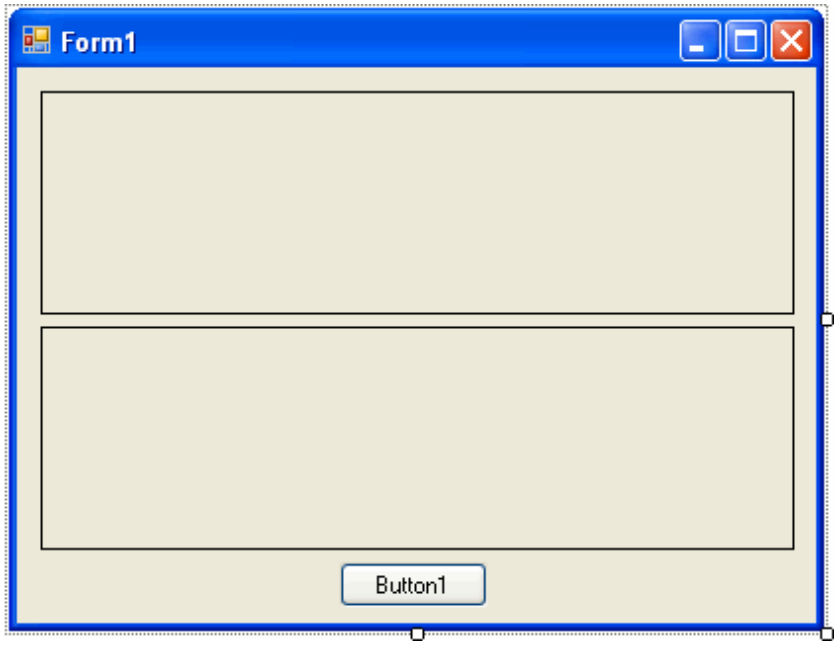
	FieldChange events, **DataObjects for .NET** handles this situation correctly. Only after all changes are done and handled, AfterChanges event is triggered.
FirstChange	Fired when a first change is made to the row (a field value changed) after BeginEdit.
UpdateRow	This event is not fired in a client application, unless it is a direct client, that is a 2-tier application updating the database directly from client, see Tutorial 3: Creating Distributed 3-Tier Applications. In a 3-tier deployment, it is fired only on the server, when a modified row is committed to the database.

6. Compile the Northwind project. Now the data library can be used in a client application.
7. Create a new Windows Application project.
8. Select **Project | Add Reference**:
    1. In the **Add Reference** dialog box, click the **Browse** tab.
    2. Locate and select **Northwind.dll** and click **OK**. The Northwind.dll assembly is added to your client project. The Northwind DLL should be located in the bin folder in the Northwind project directory.

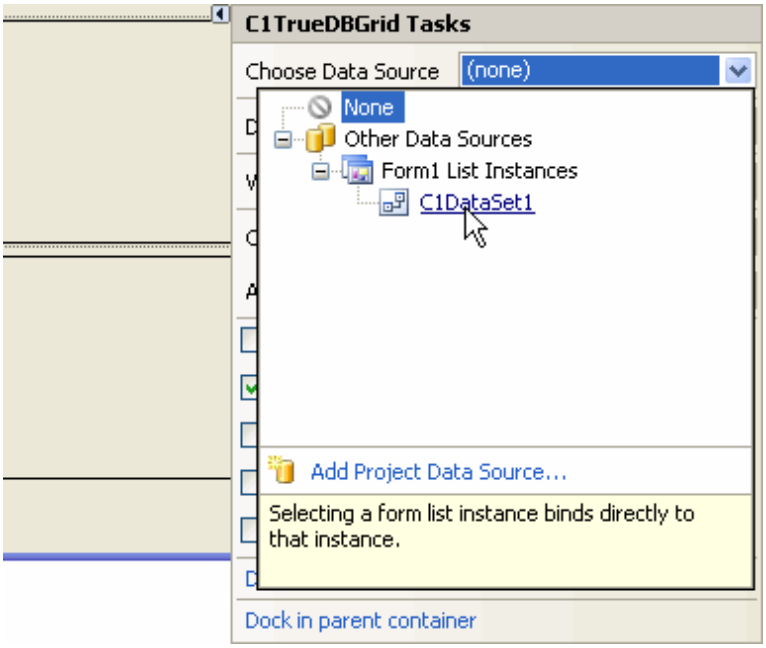


9. Place the following components on the form as shown in the figure:

Number of Components	Name	Namespace
1 C1DataSet	C1DataSet1	C1.Data.C1DataSet
2 C1TrueDBGrid	C1TrueDBGrid1 C1TrueDBGrid2	C1.Win.C1TrueDBGrid.C1TrueDBGrid
1 command Button	Button1	System.Windows.Forms.Button



10. Set the Button1.**Text** property to **Commit Changes**.
11. Select the **C1DataSet1** component and set the following properties:
    1. Enter *Northwind* in the DataLibrary property
    2. Select *ProductsOrders*from the drop-down list of the DataSetDef property.
    These settings connect **C1DataSet1** to the *ProductsOrders* data set of the data library. Now we can use the **C1DataSet1** component as a data source for data-bound GUI component grids.
12. Open the **C1TrueDBGrid Tasks menu** for each **C1TrueDBGrid** and select **C1DataSet1** under **Choose DataSource**.



13. Set the **DataMember** properties of **C1TrueDBGrid1** and **C1TrueDBGrid2** to **CustOrdersDetails** and **ProductOrderDetailsCust**, respectively. Click **Yes** to replace the existing column layout.
14. Double-click Button1 to create the **Button1_Click** event. In order to be able to send data modifications to the database, add the following code to the **Button1_Click** event:

To write code in Visual Basic

Visual Basic
`C1DataSet1.Update()`

To write code in C#

```
C#
c1DataSet1.Update();
```

15. As in Tutorial 1: Creating a Data Schema, add the following code to create the **c1DataSet1_BeforeFill** event handler to restrict (filter) the data set:

To write code in Visual Basic

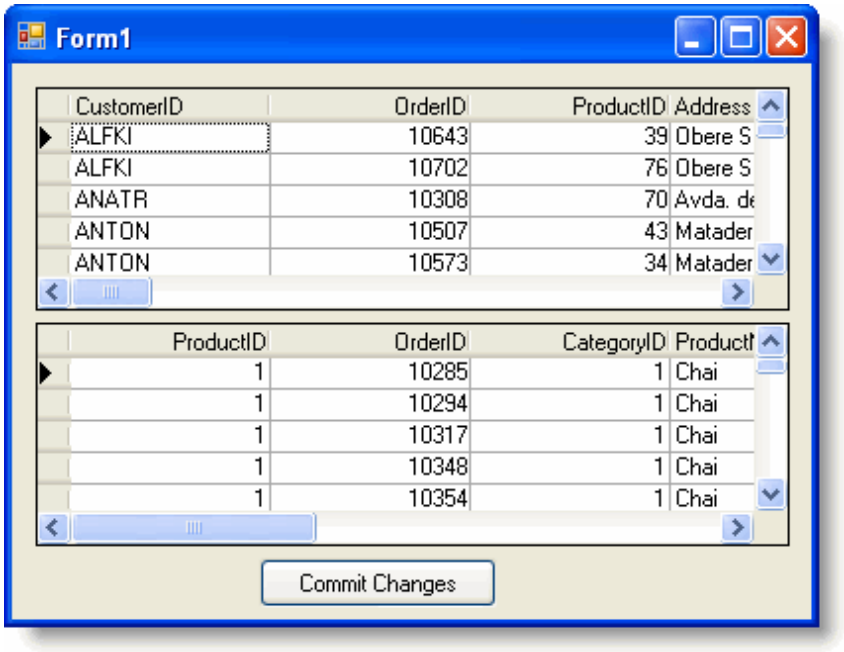
```
Visual Basic
Private Sub C1DataSet1_BeforeFill(ByVal sender As Object, ByVal e As
C1.Data.FillEventArgs) Handles C1DataSet1.BeforeFill
    Dim dataSetDef As C1.Data.SchemaObjects.DataSetDef
dataSetDef = e.DataSet.Schema.DataSetDefs("ProductsOrders")
    e.Filter.Add(New
C1.Data.FilterCondition(dataSetDef.TableViews("ProductsOrderDetailsCust"), "
[CategoryID] = 1"))
    e.Filter.Add(New
C1.Data.FilterCondition(dataSetDef.TableViews("CustOrdersDetails"), "[CategoryID]
= 1"))
End Sub
```

To write code in C#

```
C#
private void c1DataSet1_BeforeFill(object sender, C1.Data.FillEventArgs e)
{
    C1.Data.SchemaObjects.DataSetDef dataSetDef =
e.DataSet.Schema.DataSetDefs["ProductsOrders"];
    e.Filter.Add(new C1.Data.FilterCondition(dataSetDef.TableViews
   ["ProductsOrderDetailsCust"], "[CategoryID] = 1"));
    e.Filter.Add(new C1.Data.FilterCondition(dataSetDef.TableViews
   ["CustOrdersDetails"], "[CategoryID] = 1"));
}
```

Run the program and observe the following:

- The two grids show data according to the schema defined in the data library. So, when developing a client application, provided that you already have a data library, all you have to do is connect to it as to a data source. No data-oriented code or business logic code is necessary in a client GUI application. This provides a clear separation between your data and business logic and your GUI (presentation) code. You can also reuse the business logic-providing data libraries in multiple GUI front-end applications without having to modify or even know the centralized business logic code.



- Try entering a negative number in the *Quantity* column in both grids. You will see that the business logic code ensuring that the entered *Quantity* value is positive works in both table views, **CustOrdersDetails** and **ProductOrderDetailsCust**, containing the **Order Details** table.
- Change the *Quantity* value in the two grids and you will see that the business rule changing **Products**.*UnitsInStock* when the *Quantity* value changes is enforced under any circumstances where the **Products** table is involved, which in this case is in two different table views, **CustOrdersDetails** and **ProductOrderDetailsCust**. First find a row in the top and bottom grid containing the same *ProductID*; you can do this easily by sorting both grids by *ProductID*. Then change the *Quantity* value in both rows. The *UnitsOnOrder* column changes as well.

Tutorial 3: Creating Distributed 3-Tier Applications

Although Visual Studio makes great strides towards facilitating development of distributed applications, it falls short of completely automating it. You still need to add many items to your application to make it a distributed, Web application. You need to create Web services, write server-side code to retrieve and update data on the server, and so on. **DataObjects for .NET** fills this gap, making it incredibly easy to create distributed applications.

No additional steps are required to use a **DataObjects for .NET** data library in a distributed, Web environment. You develop your data and business logic code, as shown in Tutorial 2: Defining Business Logic, in exactly the same way, regardless of whether you use it in a classic two-tier client-server environment (as we did in Tutorial 2: Defining Business Logic), or in a Web distributed environment. It becomes simply a matter of deployment. In this tutorial we will do exactly this, use the Northwind data library as it was built in Tutorial 2: Defining Business Logic, in a distributed environment.

In a distributed scenario, you have your data library assembly deployed on the server and on the client. Deploying it on the client, you can use the .NET assembly download facilities, so you will not need a special client installation procedure; the data library and other parts of your application will be automatically downloaded from the server and, if necessary, automatically upgraded. **DataObjects for .NET** manages the data library code, so that necessary parts of your business logic code execute on the server and other parts execute on the client. All this is done transparently to the developer, so your business logic code must be concerned with its proper task, business logic, and does not need to organize client-server interaction or even be aware of it.

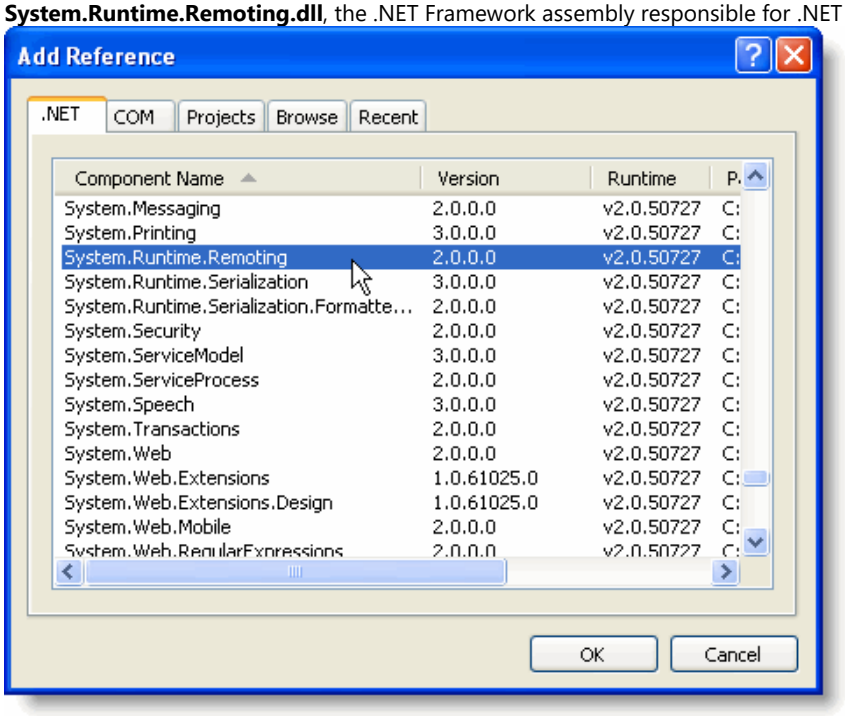
In short, **DataObjects for .NET** makes developing distributed, Web applications very simple. You just create an application as you always do, regardless of whether it is distributed, and **DataObjects for .NET** makes it distributed for you and manages all server-client interaction according to the actual deployment environment.

Server-side **DataObjects for .NET** data libraries work according to the Web services architecture; the server is stateless, and they do not maintain live database connections per user. So the distributed application support in **DataObjects for .NET** is as effective and scalable as with manually written server-side code. The only difference is that, of course, **DataObjects for .NET** does it for you.

An additional advantage of the **DataObjects for .NET** distributed model is that you can use any of the multiple deployment scenarios and options supported by .NET Remoting. You can deploy your data library on the server as a Web service, an IIS application, or you can create your own server hosting the data library assembly.

In this tutorial, we have chosen the option of hosting the data library in a special host application (server), instead of hosting it in IIS (Microsoft Internet Information Server), to make the tutorial independent of IIS. Using **DataObjects for .NET** over the Web as a Web service is even easier than the process demonstrated in this tutorial. All you need to do is publish your data library as an IIS application, creating a virtual directory hosting your data library assembly.

1. Create a new Windows Application project and name it *ThreeTierServer*.
2. Select **Project | Add Reference** and select **System.Runtime.Remoting** in the **Add Reference** window to add a reference to the **System.Runtime.Remoting.dll**, the .NET Framework assembly responsible for .NET remoting, to the project.



3. Select **Project | Add Reference** and browse to locate the compiled **Northwind.dll** assembly, built in Tutorial 2: Defining Business Logic. This adds a reference to the assembly to the project.
4. Add the following statement at the top of the form in **Code** view to import the System.Runtime.Remothing.Channels namespace:

To write code in Visual Basic

Visual Basic

```
Imports System.Runtime.Remoting.Channels
```

To write code in C#

C#

```
using System.Runtime.Remoting.Channels;
```

5. Add the following code to the **Form_Load** event:

To write code in Visual Basic

Visual Basic

```
Dim serverProv As BinaryServerFormatterSinkProvider
serverProv = New BinaryServerFormatterSinkProvider
serverProv.TypeFilterLevel = System.Runtime.Serialization.Formatters.TypeFilterLevel.Full
Dim clientProv As BinaryClientFormatterSinkProvider = New BinaryClientFormatterSinkProvider
Dim props As IDictionary = New Hashtable
props("port") = 8000
Dim chan As System.Runtime.Remoting.Channels.Tcp.TcpChannel = New System.Runtime.Remoting._
                                    Channels.Tcp.TcpChannel(props, clientProv,
serverProv)
```

```
ChannelServices.RegisterChannel(chan, False)
System.Runtime.Remoting.RemotingConfiguration.RegisterWellKnownServiceType(Type.GetType_
                                        ("Northwind.RemoteService, Northwind"),
"ThreeTierServer.soap", _

System.Runtime.Remoting.WellKnownObjectMode.Singleton)
```

To write code in C#

C#

```
BinaryServerFormatterSinkProvider serverProv;
serverProv = new BinaryServerFormatterSinkProvider();
serverProv.TypeFilterLevel = System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
BinaryClientFormatterSinkProvider clientProv = new BinaryClientFormatterSinkProvider();
IDictionary props = new Hashtable();
props["port"] = 8000;
System.Runtime.Remoting.Channels.Tcp.TcpChannel chan = new
System.Runtime.Remoting.Channels._
                                        Tcp.TcpChannel(props, clientProv, serverProv);
ChannelServices.RegisterChannel(chan, false);
System.Runtime.Remoting.RemotingConfiguration.RegisterWellKnownServiceType(Type.GetType("_
                                        Northwind.RemoteService, Northwind"),
"ThreeTierServer.soap", System._
                                        Runtime.Remoting.WellKnownObjectMode.Singleton);
```

Here **ThreeTierServer.soap** is the name we have chosen to identify this data library in its clients, the URI (Unified Resource Identifier). You can choose any unique identifier you like, provided you use the same ID in the client in the DataLibraryUrl property that specifies the connection.
This code registers **DataObjects for .NET** remoting service with .NET remoting framework. In this tutorial, we use TCP/IP channel, not the more popular HTTP channel that you would normally use in a Web application, because we want the tutorial to work on machines with no Web server running.
Instead of this code, you can use a configuration file for remoting registration. If you are deploying your data library in IIS, a configuration file is the only option. A configuration file, *web.config*for IIS deployment, may look like the following example:

```
<configuration>
   <system.runtime.remoting>
       <application> <service>
           <wellknown mode="Singleton"
                type="Northwind.RemoteService, Northwind" objectUri="ThreeTierServer.soap" />
           </service>
       </application>
  </system.runtime.remoting>
</configuration>
```

Now that our Northwind data library is deployed on a server, we can use it in a client application. This client application can reside on the same machine, or it can be thousands of miles away, it is just a matter of setting a single property in the client application specifying the connection URL. A client application for distributed use is created in exactly the same way as a regular client application; in fact, it is a regular client application, such as the one in Tutorial 2: Defining Business Logic. The only difference is that we set the connection property, DataLibraryUrl.

6. Copy the client project from Tutorial 2: Defining Business Logic. This will serve as our client application.
7. Open the form and select the **C1DataSet1** component.
8. Set C1DataSet1.DataLibraryUrl = **tcp://localhost:8000/ThreeTierServer.soap**. This connects it to the server.

Run the Server Application, Then Run the Client Application and Observe the Following:

Run the ThreeTierServer application first and make sure it is running while the client application is running.

The client application behaves exactly as the application we built in Tutorial 2: Defining Business Logic. This is **DataObjects for .NET**'s transparent distributed application development at work; with no code changes we have deployed our application in a distributed environment. All database access code is executed on the server, and business logic is executed on the client without a single line of

code added for that purpose to the client application. The server can be anywhere on the Internet, it is just a matter of changing the URL from localhost to an Internet address, for example, to c1DataSet1.DataLibraryUrl = http://www.mycompany.com/ThreeTierServer.soap.

Tutorial 4: Virtual Mode: Dealing with Large Datasets

Visual Studio and its underlying data engine, ADO.NET, support only one mode: disconnected access to data. An ADO.NET data set is always pre-fetched in its entirety from the server to the client. Pre-fetching large data sets over the wire creates two serious problems:

- Even distributed Web applications often include large data sets, such as a list of available products and other data originated in database tables with thousands of records or more. It is very inefficient and in many cases impossible to transfer such data from the server to the client in its entirety. This forces developers to produce makeshift solutions, such as asking the user to enter a few initial letters of the product name before showing the list of products, severely reducing the quality of end-user experience, application performance and scalability.
- The absence of large data set support makes it impossible to develop classic client-server and desktop database applications in Visual Studio (without **DataObjects for .NET**). This is a serious drawback since many developers need to develop such applications. These applications traditionally require access to large data sets. A popular belief that you only need large data sets if you have not designed your application correctly is a misconception due to the lack of tools supporting the right data access modes. Another popular misconception is that the disconnected model, also loosely referred to as *Web application*, *3-tier application*, and so on, necessarily means pre-fetching all the data from the server to the client at once and that any other approach is the old *live connection per user* approach that is not scalable.

DataObjects for .NET fills this gap, offering a solution to the problem of large data sets. It proves the misconceptions mentioned above wrong. It gives you the tool to achieve the best of both worlds: to have data access that is both disconnected, or no live connection is maintained on the server for particular users, and is therefore scalable, and at the same time, unlimited in data size.

To use a large data set in **DataObjects for .NET** is as easy as setting the DataAccessMode property to **Virtual Mode**.

Perhaps you are skeptical and think that it might work in a demo but never in the real world with huge tables, and so on. Well, how about a table of **2.7 million rows**? This is what is shown in this tutorial, and you will see this data appearing in a **TrueDBGrid** with no delay, and it can be scrolled without noticeable delays.

Although it is not demonstrated in this tutorial, you can use virtual data access mode in distributed Web-based applications as well, simply by setting the DataAccessMode property to *Virtual*. See Tutorial 3: Creating Distributed 3-Tier Applications for more information.

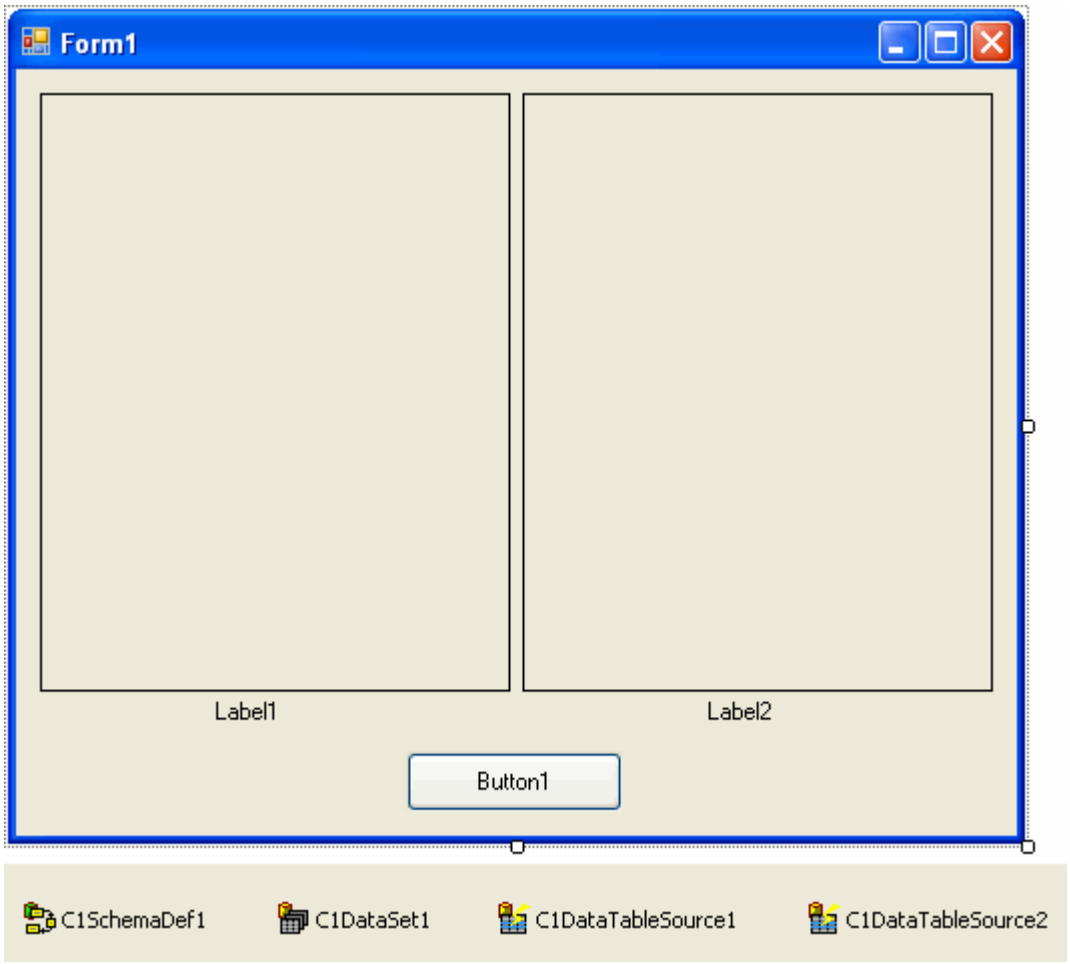
We will begin by creating a new Windows Application project. In this tutorial we will create two forms.

To create Form1:

1. Place the following components on the first form as shown in the figure.

Number of Components	Name	Namespace
1 C1SchemaDef	C1SchemaDef1	C1.Data.C1SchemaDef
1 C1DataSet	C1DataSet1	C1.Data.C1DataSet
2 C1DataTableSource	C1DataTableSource1 C1DataTableSource2	C1.Data.C1DataTableSource
2 C1TrueDBGrid	C1TrueDBGrid1 C1TrueDBGrid2	C1.Win.C1TrueDBGrid.C1TrueDBGrid

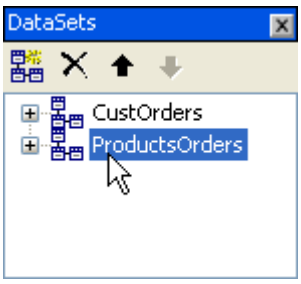
2 Label	Label1 Label2	System.Windows.Forms.Label
1 command Button	Button1	System.Windows.Forms.Button



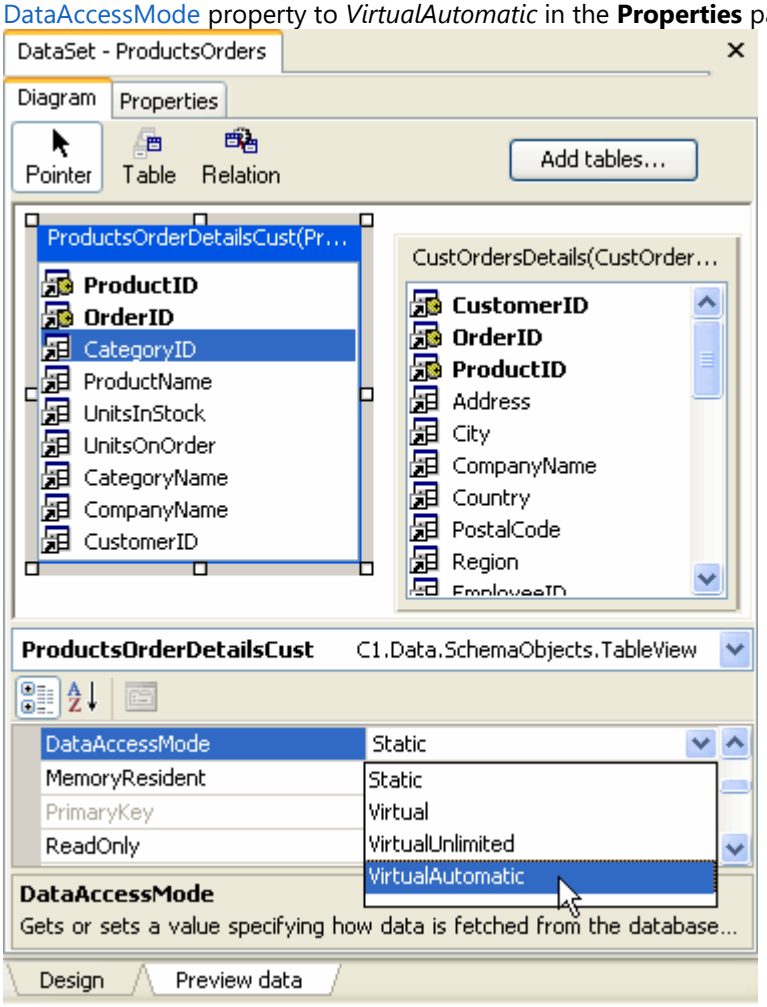
2. Set the **Text** property of each label to **Fetching data**.
3. Set the **Text** property of **Button1** to **Huge Table**.
4. Select the **C1SchemaDef1** control, use the smart tags to expand the **C1SchemaDef Tasks** menu and select **Schema Designer**.

The **Schema Designer** opens, and the **Import Wizard** appears, but you can click **Cancel** to close the wizard.

5. Select **File | Open** in the **Schema Designer** menu and open the schema file that was saved in Tutorial 1: Creating a Data Schema. The schema appears in the designer
6. To enable virtual mode, we must set the DataAccessMode property for the table view objects that we need to be accessed "virtually". "Virtual", as opposed to "static", means data is fetched in increments, on demand, as opposed to fetching everything at once:
    1. In the **View** menu, make sure that **DataSets** is checked.
    2. Double-click the **ProductsOrders** dataset in the **DataSets** window to open it.



3. In the **DataSet Editor**, click the title bar of the **ProductsOrderDetailsCust** table view and set the DataAccessMode property to *VirtualAutomatic* in the **Properties** panel.

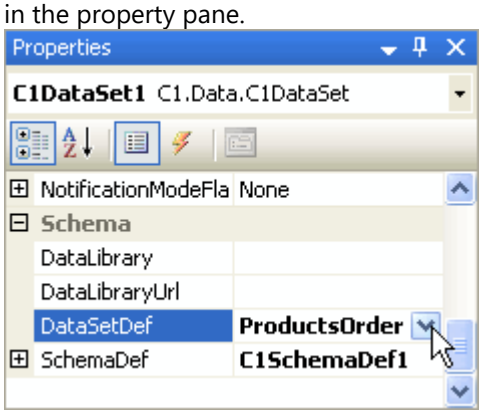


4. Click the title bar of the **CustOrdersDetails** table view and set the DataAccessMode property to **VirtualAutomatic**.
   DataAccessMode = *VirtualAutomatic* means that the fetch is performed in increments, or segments, asynchronously in background mode while the application becomes available to the user and interactive immediately after the first segment is fetched. The user has interactive access not just to the data already fetched in background, but to the whole table, including its physical end, so the user sees data in its entirety, as if all data were transferred to the client, although the background fetch may not yet be complete. To enable this transparency, **DataObjects for .NET** performs the fetch on demand, when the user requests data not yet fetched from the server, in addition to the background fetch.

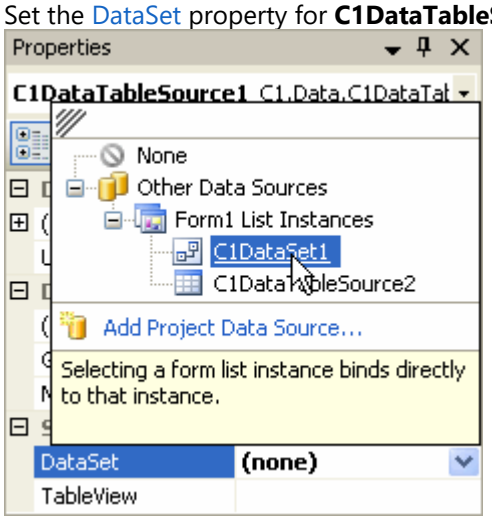
   DataAccessMode = *VirtualAutomatic* is appropriate for large tables that are big enough to make it undesirable to fetch all data at startup time (doing so would incur a long delay before the application becomes interactive), but not too big, so they can still fit in client memory. Later in this tutorial we will also use DataAccessMode = *Virtual*, which is intended for very large tables, so large that they do not fit in memory.

5. Close the **Schema Designer** and click **Yes** to save the schema.

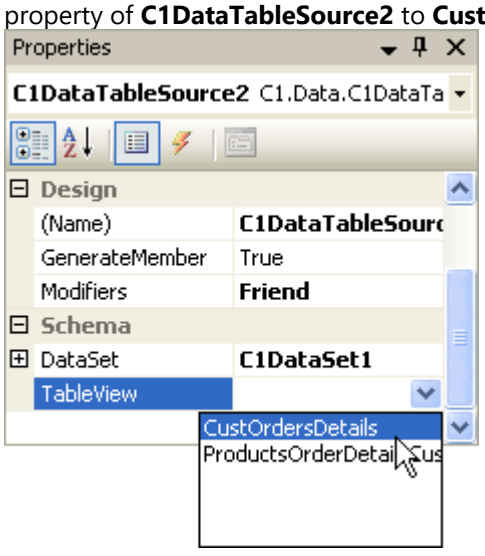
7. Set the C1DataSet1's SchemaDef property to **C1SchemaDef1** and the DataSetDef property to **ProductsOrders** in the property pane.



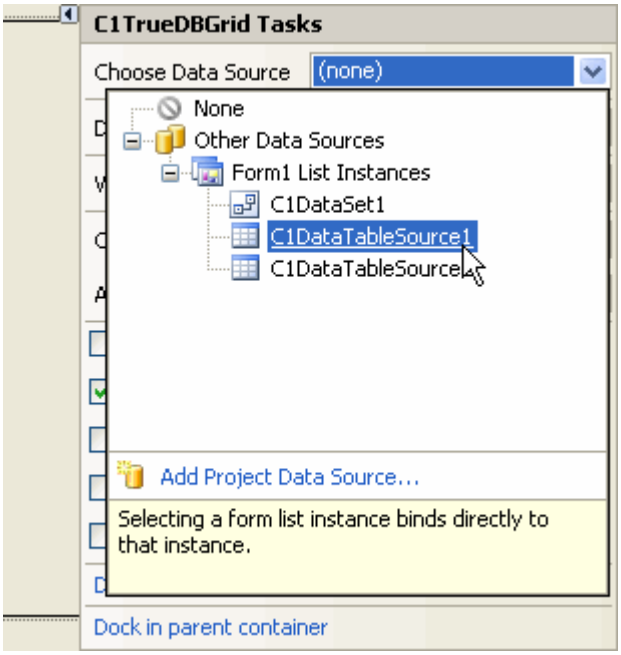
8. Set the DataSet property for **C1DataTableSource1** and **C1DataTableSource2** to **C1DataSet1**.



9. Set the TableView property of **C1DataTableSource1** to **ProductsOrderDetailsCust** and set the TableView property of **C1DataTableSource2** to **CustOrdersDetails**.



10. Bind the **C1TrueDBGrid** components to the **C1DataTableSource** components:
    1. Set C1TrueDBGrid1's **DataSource** to **C1DataTableSource1**.



2. Set C1TrueDBGrid2's **DataSource** to **C1DataTableSource2**.

Note that we bind the grids to C1DataTableSource components rather than directly to a C1DataSet component. This is essential for virtual mode; it works only if you use C1DataTableSource as your data source.

11. Add the following code to create a **C1DataTableSource1_AsyncFetchComplete** event handler:

To write code in Visual Basic

Visual Basic

```
Private Sub C1DataTableSource1_AsyncFetchComplete(ByVal sender As Object, ByVal
e As System.EventArgs) Handles C1DataTableSource1.AsyncFetchComplete
    Label1.Text = "Fetch complete. Record count: " +
C1DataTableSource1.DataTable.Rows.Count.ToString()
End Sub
```

To write code in C#

C#

```
private void c1DataTableSource1_AsyncFetchComplete(object sender,
System.EventArgs e)
{
    label1.Text = "Fetch complete. Record count: " +
c1DataTableSource1.DataTable.Rows.Count.ToString();
}
```

12. Add the following code to create a **C1DataTableSource2_AsyncFetchComplete** event handler:

To write code in Visual Basic

Visual Basic

```
Private Sub C1DataTableSource2_AsyncFetchComplete(ByVal sender As Object, ByVal
e As System.EventArgs) Handles C1DataTableSource2.AsyncFetchComplete
    Label2.Text = "Fetch complete. Record count: " +
C1DataTableSource2.DataTable.Rows.Count.ToString()
```

```
End Sub
```

To write code in C#

C#

```
private void c1DataTableSource2_AsyncFetchComplete(object sender,
System.EventArgs e)
{
    label2.Text = "Fetch complete. Record count: " +
c1DataTableSource2.DataTable.Rows.Count.ToString();
}
```

This completes the Form1 set up, where we used DataAccessMode = *VirtualAutomatic*.

Now let's add a second form to the project where we show a very large table with 2.7 million rows. For such huge data amounts we must use DataAccessMode = *Virtual*.

In this part of the tutorial we need a SQL Server connection. Up until now a Microsoft Access sample database was enough. MS Access, being a desktop database, does not provide sufficient query optimization, so it cannot be effectively used in **DataObjects for .NET** virtual mode. We will use the SQL Server sample Northwind database included in the standard SQL Server installation.

Note: If you are running the pre-built tutorial projects included in **DataObjects for .NET** installation, you will need to change the connection string in this tutorial to point to your SQL Server instance.

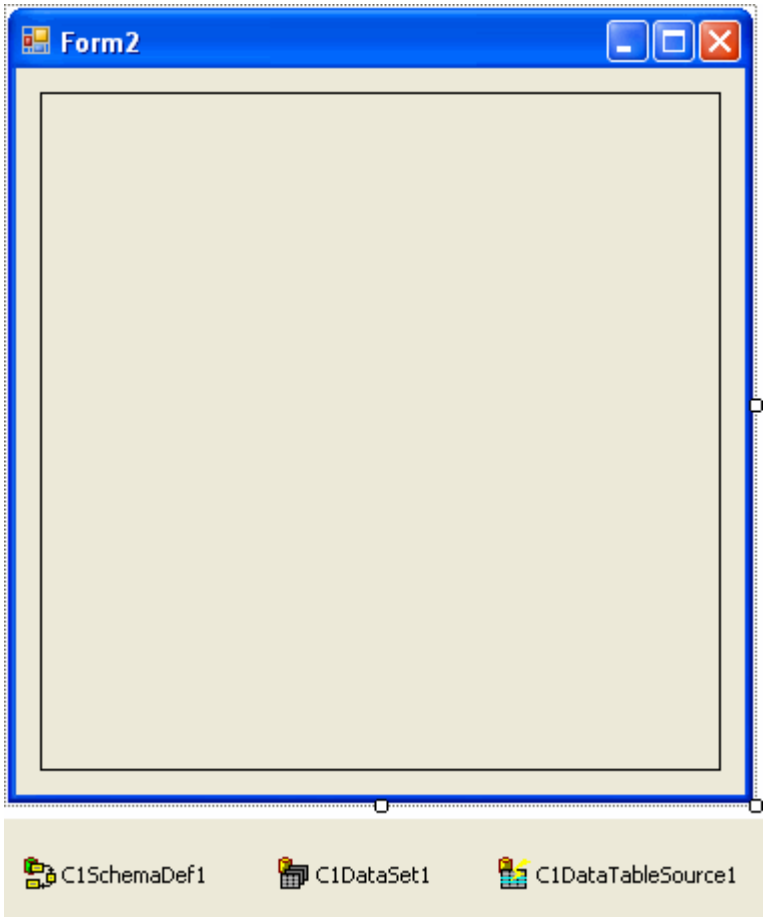
Obviously, we do not want to physically fill 2.7 million rows in your SQL Server database, so we will simulate a huge table by creating a composite table with **self-joins**, repeating the same tables, **Order Details** and **Products**, several times to create the desired effect:

```
Order Details →(∞-1) Products →(1-∞) Order Details →(∞-1) Products →(1-∞) Order
Details
```

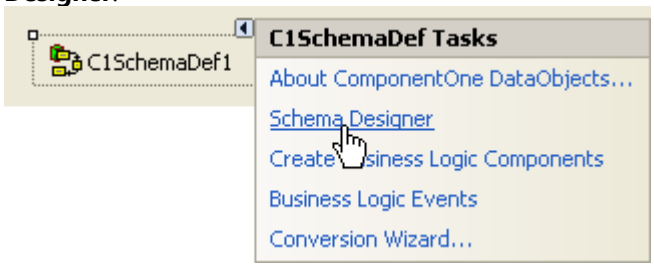
To create Form2:

1. Select **Project | Add Windows Form**. The **Add New Item** dialog box appears.
    1. Choose **Windows Form** from the **Templates** pane.
    2. Name the form **Form2** and click **Add**.
2. Place the following components on the form as shown in the figure.

Number of Components	Name	Namespace
1 C1SchemaDef	c1SchemaDef1	C1.Data.C1SchemaDef
1 C1DataSet	c1DataSet1	C1.Data.C1DataSet
1 C1DataTableSource	C1DataTableSource1	C1.Data.C1DataTableSource
1 C1TrueDBGrid	c1TrueDBGrid1	C1.Win.C1TrueDBGrid.C1TrueDBGrid

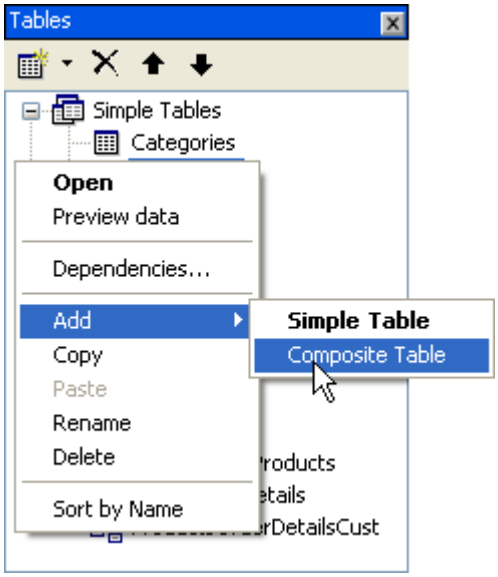


3. As in other tutorials, we start with the schema created in Tutorial 1: Creating a Data Schema. Select the **C1SchemaDef1** control, click the smart tag to open the **C1SchemaDef Tasks** menu, and select **Schema Designer**.

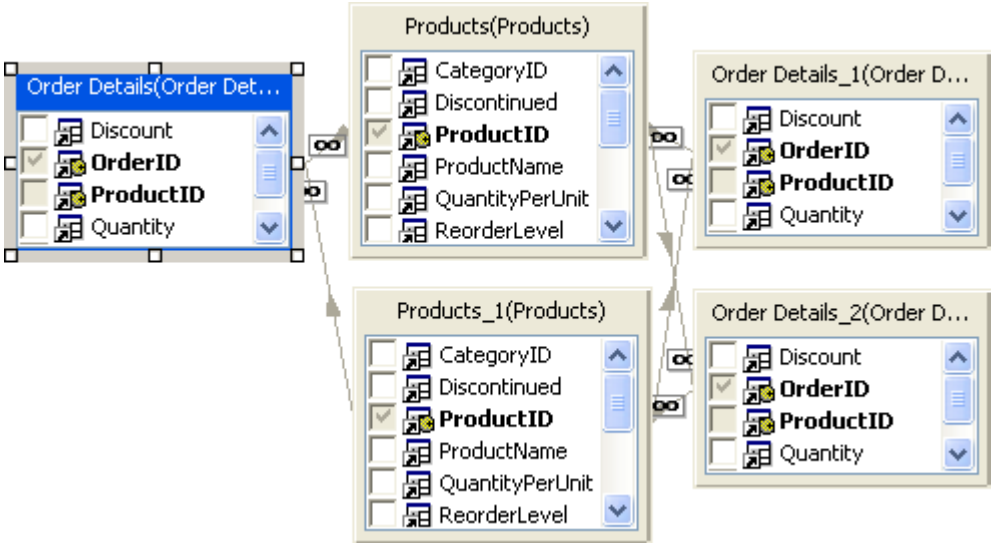


The **Schema Designer** opens, and the **Import Wizard** appears, but you can click **Cancel** to close the wizard.

4. Select **File | Open** in the **Schema Designer** menu and open the schema file that was saved in Tutorial 1: Creating a Data Schema. The schema will appear in the designer.
5. Right-click the **Tables** window and select **Add | Composite table** from the context menu.

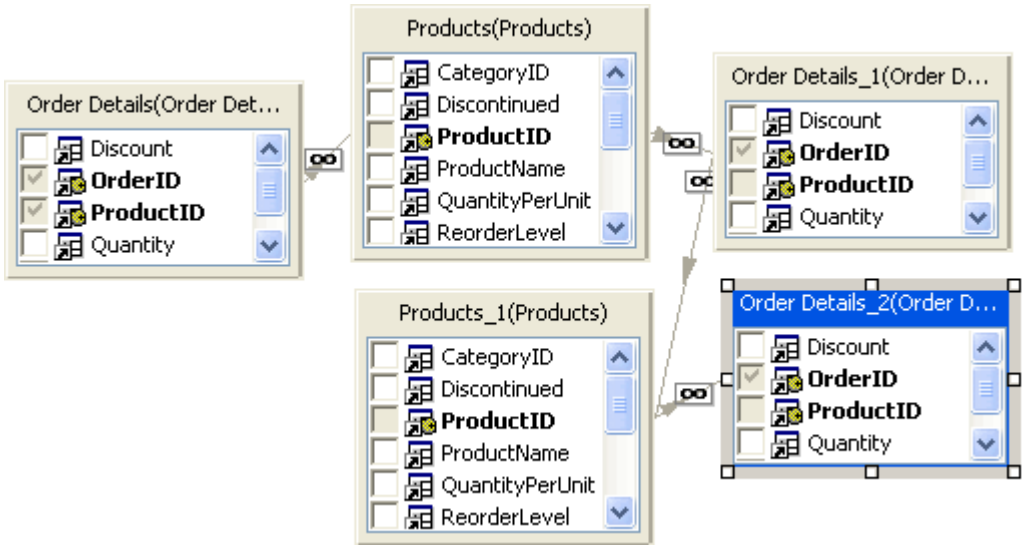


6. In the newly added node, change the default name *CompositeTable* to *HugeTable*.
7. Select the **Diagram** tab in the **Composite Table Editor** and drag-and-drop the following tables from the **Tables** window to the **Composite Table Editor**, arranging them as in the image below:
    - **Order Details** – drag and drop this table three times.
    - **Products** – drag and drop this table two times.



8. In the resulting diagram, delete the following redundant relations by selecting the relation and pressing the **Delete** key (or right-click the relation and select **Remove** from the context menu):
    - **Products – Order Details_2**
    - **Products_1 – Order Details**
9. Invert the first (**Products – Order Details**) and third (**Products_1 – Order Details_1**) relations by selecting the relation and choosing **Invert** from the context menu, so that the resulting diagram becomes:

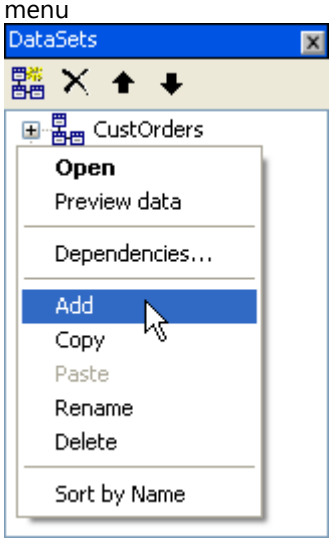
```
Order Details →(∞-1) Products →(1-∞) Order Details →(∞-1) Products →(1-∞) Order
Details
```



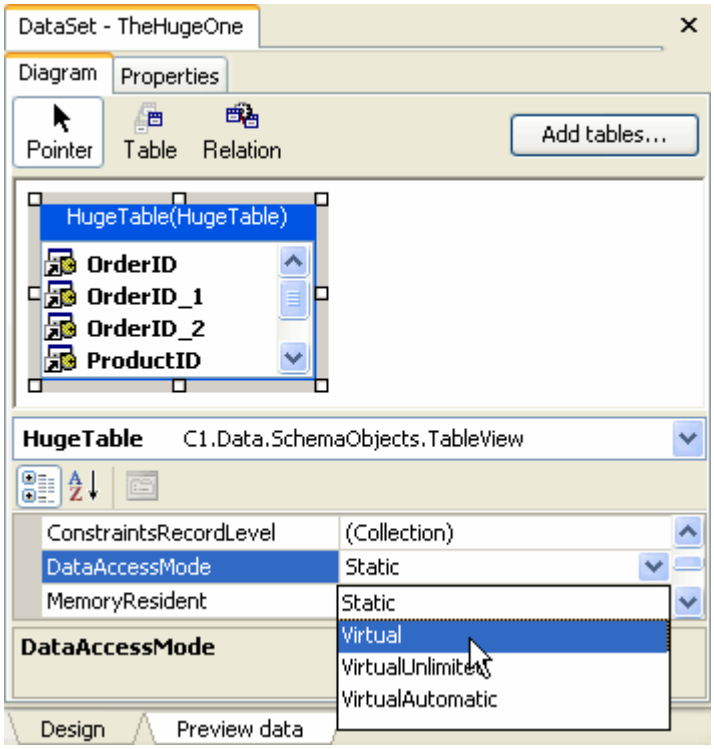
10. Define composite table fields by checking the check boxes for the following fields:

Table	Field
Order Details	OrderID ProductID
Products	ProductName
Order Details_1	OrderID
Products_1	ProductName
Order Details_2	OrderID

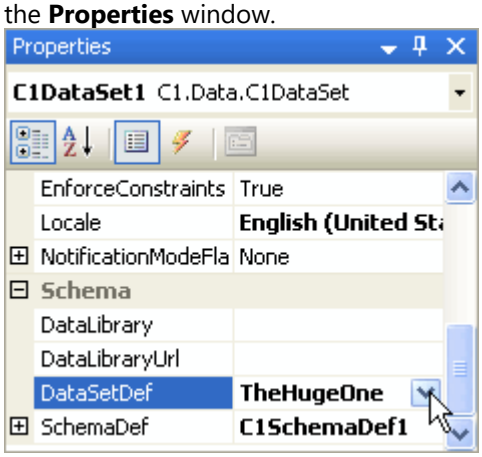
11. If necessary, select **View | DataSets** and then right-click the **DataSets** window and select **Add** in the context menu



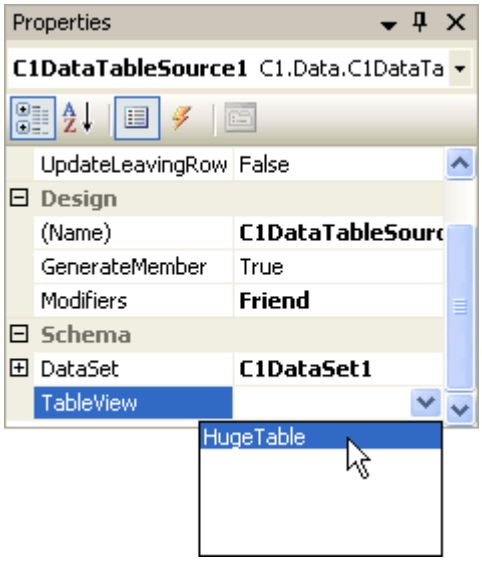
12. For the newly created data set node, change the default name *DataSet* to *TheHugeOne*.
13. Drag the **HugeTable** node from the **Tables** window and drop it in the **DataSet Editor**. This creates a data set consisting of a single table, *HugeTable*.
14. Set the DataAccessMode property of the **HugeTable** table view to *Virtual*.



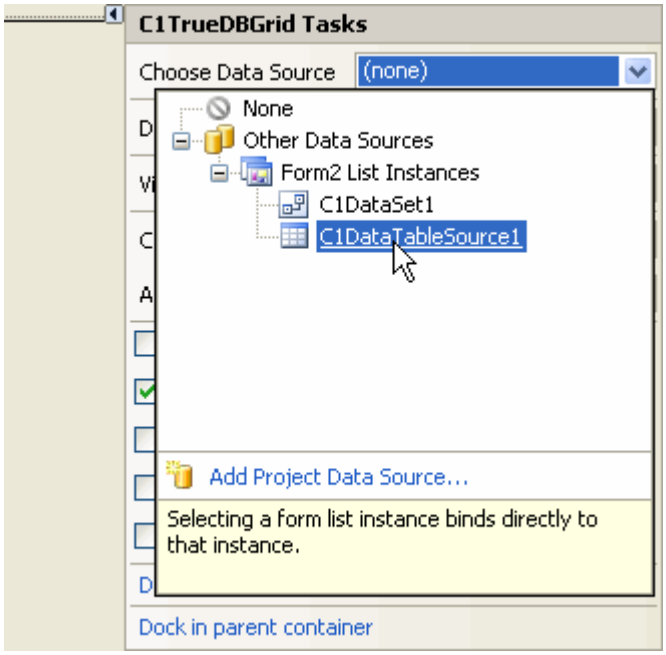
15. Now we must change the database connection from MS Access to SQL Server in order to use the SQL Server sample Northwind database included in the standard SQL Server installation.
    1. If necessary, select **View | Connections** and then double-click the existing **Connection** node in the **Connections** window. The **Connection Editor** opens.
    2. Click the ellipsis button to the right of the Connection string to open the standard **OLE DB Data Link Properties** dialog box.
    3. Click the **Provider** tab and select **Microsoft OLE DB Provider for SQL Server**.
    4. Click the **Connection** tab and select your SQL Server server name; enter your login information, if necessary; and select the **Northwind** database.
    5. Click **OK** to close the **Data Link Properties** dialog box.
    6. Close the **Schema Designer** and click **Yes** to save the schema.
16. Set the C1DataSet1's SchemaDef property to **c1SchemaDef1** and the DataSetDef property to **TheHugeOne** in the **Properties** window.



17. Set the C1DataTableSource's DataSet property to **C1DataSet1** and the TableView property to **HugeTable**.



18. Set the C1TrueDBGrid1's **DataSource** to **C1DataTableSource1** to bind the grid to the **C1DataTableSource**.



Note that we bind the grid to the **C1DataTableSource** components rather than directly to the **C1DataSet** component. This is essential for virtual mode; it works only if you use **C1DataTableSource** as your data source.

19. To activate Form2, switch to Form1 and double-click **Button1** to switch to code view and add the **Button1_Click** event. Add the following code to the **Button1_Click** event in Form1:

To write code in Visual Basic

Visual Basic

```
Dim form2 As Form2
form2 = New Form2()
form2.ShowDialog(Me)
```

To write code in C#

C#

```
Form2 form2 = new Form2();
form2.ShowDialog(this);
```

Run the program and observe the following:

- The first form demonstrates DataAccessMode =*VirtualAutomatic*, where **DataObjects for .NET** continues to fetch data in the background while the user can interact with the grid as though the data were fetched completely to the client. The labels below the grids show the status of the background fetch: they show "Fetching data..." until the fetch is complete. While the label still shows "Fetching data...", try going to the last row by dragging the scrollbar thumb to the bottom. After a brief delay you will see the last row (CustomerID="WOLZA", OrderID=11044). This is the actual last row of the table, and **DataObjects for .NET** positions to the end of the table (as well as to any other desired position) correctly, although the background fetch is still in progress and has not yet reached the last row. Unlike asynchronous fetch features in other tools, **DataObjects for .NET** implements it transparently to the user; the user does not need to be aware that background fetch is incomplete.
- When **DataObjects for .NET** finishes fetching all rows in the background, it automatically switches to the regular in-memory mode, so the scrollbar is now exactly positioned according to the current row number. While the fetch is in progress, the scrollbar position is approximate, because the total row count and absolute row numbers are not yet known.
- Next press the "Huge table" button and enjoy the view of a 2.7 million rows table!
- Try scrolling forward fast by continuously pressing the PageDown key in quick succession. You will notice that after a few keystrokes it pauses for a brief moment. This occurs when **DataObjects for .NET** refreshes its cache, fetching a new portion of data from the server. Now try scrolling forward a bit slower, taking a short pause after each time you press PageDown. You will notice that if you do it this way, the delays disappear; the grid reacts immediately to every keystroke. This is a result of one of the optimization techniques employed by **DataObjects for .NET**: if necessary, it fetches additional segments of data during periods of inactivity. Since users rarely scroll through data without at least taking a look, in most cases, this technique creates the impression of continuous smooth scrolling without pauses and delays.
- Try going to the last row by dragging the scrollbar thumb to the bottom. After a brief delay you will see the last row (OrderID=11077, ProductID= 77,...). **DataObjects for .NET** only paused for a brief moment, which is necessary to fetch the segment of rows in the end of the table; it did not fetch the whole enormous number of rows as other tools do when last record is requested.

DataObjects for .NET Express Edition

DataObjects Express Edition makes it exceptionally easy to work with data in .NET applications. It shields developers from the complexities of ADO.NET, makes data bound forms as easy to build as it used to be in previous versions of Microsoft Visual Studio. However, **DataObjects for .NET Express** is not just about ease of use; it also adds many power features absent in standard ADO.NET:

- **DataObjects for .NET Express** fully supports **multi-table rowsets** (*composite tables*) automatically enforcing data relations without manual coding. For example, changing a *CustomerID* field will automatically change the corresponding *CustomerName* field in the same row, although it is stored in a separate table.
- With an innovative **virtual mode** technology, **DataObjects for .NET Express** allows you to use **large datasets** in .NET Windows Forms applications, the feature that is not supported in Visual Studio and ADO.NET without **DataObjects for .NET**.
- **DataObjects for .NET Express** completely automates **database updates**. There is no need to use ADO.NET DataAdapter or other special components. Database updates are performed without manual coding. **DataObjects for .NET Express** can update the database even when multiple and interrelated changes have been made to multiple tables.
- Setting a single property, UpdateLeavingRow, you can make **DataObjects for .NET Express** *update the database immediately* after the user changes a row. This optional feature is commonly used in desktop and classic client-server applications. Standard ADO.NET does not support this feature.
- **DataObjects for .NET Express** supports an extensive set of **events** enabling full programmatic customization.
- **DataObjects for .NET Express** is a special edition of **DataObjects**. The **Express Edition** is geared toward ease of use and optimized for desktop and client-server applications. The main goal of Express Edition is to provide a data framework for .NET that is very easy to use, highly intuitive and requires minimal effort to master. If you need even more power, consider using **DataObjects Enterprise Edition** supporting additional power features:
- **DataObjects for .NET Enterprise Edition** uses the standard **business object** paradigm to allow you to develop business logic components (data libraries) and reuse them in multiple client projects. It provides **clear separation of business and data logic** from the presentation (GUI) layer.
- **C1DataObjects Enterprise Edition** allows you to create a centralized and reusable repository of data schema and business logic (data libraries) used in applications throughout the enterprise.
- **C1DataObjects Enterprise Edition** completely automates the task of developing **distributed 3-tier Web-based applications**. No special server-based code is necessary, and making your application distributed becomes a simple matter of deployment configuration.

DataObjects for .NET Express and **C1DataObjects Enterprise Edition** are not mutually-exclusive, they can even be used together in the same application. They have common runtime core functionality and object model. The difference is primarily in design time: developers use a special **Schema Designer** in **DataObjects for .NET** whereas **DataObjects for .NET Express** is a suite of three simple components with built-in design-time support.

C1ExpressTable: Working with Simple and Composite Tables

C1ExpressTable is the main component of **DataObjects for .NET Express**. It defines a *table*, a *rowset*. Data-aware controls, such as ComponentOne, Microsoft or third-party grid controls, can bind to a C1ExpressTable component as their DataSource. A C1ExpressTable component can be used either as a standalone data control or attached to a C1ExpressConnection component. In the latter case, all C1ExpressTable components attached to a C1ExpressConnection share the same database connection and form a *data set*, see C1ExpressConnection: Combining Tables into Data Sets.

Connecting to Database and Working with Data

You can connect to a database and begin setting properties for your table.

1. Connect to a database:
   - A standalone C1ExpressTable component is connected to a database by setting its ConnectionString property.
   - A C1ExpressTable component attached to a C1ExpressConnection (its ConnectionComponent property set) does not need a separate database connection; it uses one of C1ExpressConnection.
2. After connecting to a database, set the DbTableName property choosing a database table from a combo box or creating Composite Tables.
   It is recommended to use the DbTableName property to bind a C1ExpressTable to a database table, but you can also use the SelectCommandText property to specify a SQL statement or a stored procedure (the latter if the SelectCommandType property is set to StoredProcedure). Using SelectCommandText is almost as simple as using DbTableName, but you also need to create a DataAdapter component for this C1ExpressTable if you need it to be modifiable, have update functionality (select **Create DataAdapter** from the context menu). Also, SQL-based C1ExpressTable components (those that use SelectCommandText instead of DbTableName) cannot be used in composite tables and in virtual mode, and their FillFilter property is ignored.
3. The last step is defining fields using the **Fields Editor** that opens when you press the button for the Fields property. For simple tables, **DataObjects for .NET Express** retrieves the fields collection from the database table. For a composite table, you must add fields from underlying database tables to your composite table using the **Fields Editor**. You can customize the Fields collection deleting and rearranging the fields, adding calculated fields and setting various field properties.

Once a C1ExpressTable component is bound to a database table(s), it can be used as a data source in data-aware GUI controls. Just select the component in the DataSource property combo box of a data-aware control. Note, that if you need master-detail data binding, you must use a C1ExpressConnection component as your DataSource and select appropriate DataMember values. The string representing a **DataObjects for .NET Express** table in **DataMember** is determined by the TableName property.

The FillOnRequest property (belonging to a C1ExpressConnection component) determines whether C1ExpressTable components are filled with data automatically at start-up (when data-bound controls request data from it). Setting FillOnRequest to **False**, you can fill them with data later, programmatically, calling the Fill method (C1ExpressTable.C1ExpressConnection.Fill).

Unless you specify a filter condition in the FillFilter property, C1ExpressTable will fetch all existing table data unrestricted. To restrict fetch, specify a SQL filter condition in the FillFilter property. In this condition, use field names in brackets, for example,

```
C1ExpressTable1.FillFilter = "[CategoryID] = 1"
```

Setting the FillSort property you can specify the order of fetched rows, for example,

```
C1ExpressTable1.FillSort = "OrderID,CustomerID"
```

By default, rows are sorted by primary key.

By default, **DataObjects for .NET Express** works in disconnected mode, just like standard ADO.NET: data is pre-fetched from the database in its entirety. In this mode, you cannot bring large datasets, with hundreds of thousands of records and more, to the client. However, **DataObjects for .NET Express** also supports so called *virtual mode*, where data size is unlimited. With its innovative virtual mode technology, **DataObjects for .NET** (both editions) allows you to use large datasets, unlimited in size, in .NET WinForms applications. This feature is not supported in Visual Studio .NET and ADO.NET without **DataObjects for .NET**. See Tutorial 4: Virtual Mode: Dealing with Large Datasetsfor an example of a rowset containing 2.7 million rows that is available on the client with limited memory consumption and startup time of about three seconds. See Tutorial 4: Virtual Mode: Dealing with Large Datasets in **DataObjects for .NET** documentation for details on virtual mode. In **DataObjects for .NET Express**, virtual mode is enabled by setting the DataAccessMode property of a C1ExpressTable component.

The AllowAddNew, AllowDelete, and ReadOnly properties determine whether the user can modify table data. By default, user modifications are cached on the client and not sent back to the database. The database is updated,

modifications written to the database, when the Update method is called from code. **DataObjects for .NET Express** also supports an *automatic update* mode. In this mode, a modified row is automatically updated to the database when the end user leaves that row, when another row becomes current. To enable automatic update mode, set the UpdateLeavingRow property to **True**.

Using Composite Tables

In many cases, simple database tables are not enough; you need a rowset combining data from several database tables. For example, you may need *CustomerID* and *CustomerName* fields where *CustomerID* belongs to a database table *Orders* and *CustomerName* belongs to a database table *Customers*. It is customary to use SQL statements for this purpose. However, direct use of multi-table SQL statements causes many problems, the data cannot be easily updated, the resulting rowset does not preserve the structure and relations used to obtain it, and so on, not to mention inherent complexities of using SQL.

DataObjects for .NET Express automates this task, as many others. You can create a composite table, specifying database tables it consists of and joins between those tables. Fetching and updating data, **DataObjects for .NET Express** automatically generates necessary SQL statements, transparently to the developer. Working with data, **DataObjects for .NET Express** maintains the structure specified in composite table definitions. For example, changing *CustomerID* will automatically change the related field *CustomerName*, without any manual coding.

A **C1ExpressTable** component represents either a simple database table or a composite table. To specify a composite table, select **Composite** in the DbTableName property combo box or select **Composite Table Editor** from the context menu.

In the **Composite Table Editor**, add database tables constituting the composite table. Constituent database tables form a hierarchical structure (a tree). Filling a composite table with data can be represented as performing nested loops over this hierarchy. Each node, a constituent database table, is attached to its parent either by 1-M (one-to-many) or M-1 relation. A one-to many relation represents a child table with independent key, as in Customers à(1-M) Orders. A many-to-one relation represents a main table and a lookup table, as in Orders à(M-1) Customers. The difference is that in 1-M relation there are many child rows for a single parent row, whereas in an M-1 relation there only one child row for each parent row. The constituent table hierarchy satisfies a "non-branching" restriction: only one 1-M-child is allowed for each parent (M-1-relations are not restricted and can branch). Each node of the hierarchy, each constituent database table, except the first one, must be joined to its parent with one or more joins – equalities between a parent and a child field. An example of composite table structure is as follows:

```
Orders à (1-M) OrderDetails à (M-1) Products à (M-1) Categories
```

Corresponding SQL statement automatically generated by **DataObjects for .NET Express** (simplified):

```
SELECT * FROM ((Orders INNER JOIN OrderDetails ON Orders.OrderID =
OrderDetails.OrderID) INNER JOIN Products ON (OrderDetails.ProductID =
Products.ProductID))
INNER JOIN Categories ON (Products.CategoryID = Categories.CategoryID)
```

By default, M-1 joins in a composite table define a referential integrity (foreign key) constraint. So, for example, in a composite table *CustomersOrders* it is not allowed to enter *CustomerID* that is not present in the *Customers* table. If you want to change this behavior, define a relation in the **C1ExpressConnection** component with the same joins as used in the composite table and uncheck the **Enforce constraints** check box. See Defining Relations for more details.

Defining Fields

For simple tables, **DataObjects for .NET Express** retrieves the fields collection from the database table. For a composite table, you must select fields from underlying database tables and add them to your composite table using the **Fields Editor**. You can customize the Fields collection deleting and rearranging the fields, adding calculated fields and setting various field properties.

A field has a **Name** that must be unique in the table. A field can be renamed by renaming the field node in the Fields

list. The name is used to identify the field as the name of the column exposed to the users, and as the default display name (caption) of the column.

The DataType property determines the field .NET type, and NativeDbType – its native database or OLE DB type. NativeDbType is used only for updating values in the database, and can be set to *Any (-1)*, in which case its value is effectively ignored. When in doubt, use*Any (-1)* as the default value forNativeDbType.

DB Fields and Calculated Fields

A *DB field* represents a database table field as determined by its DbFieldName property.

A *calculated field* is not based on a database field. Its value is determined by one or more calculations (FieldCalculationInfo objects) or assigned in code. Each calculation (FieldCalculationInfo) contains an Expression, which is the expression used to obtain field values (see DataObjects for .NET Expressions for a description of **DataObjects for .NET** expression language), and two additional properties: Condition and FireEvent. Condition is an optional Boolean expression determining calculation's applicability. If Condition evaluates to **False**, the calculation expression is skipped, not evaluated. If a field has multiple calculations, **DataObjects for .NET Express** applies the first with Condition expression evaluating to **True**. If none is applicable, the field value is left unchanged. FireEvent is a Boolean property set to **False** by default. If it is set to **True**, setting the value from calculation expression triggers the same sequence of events (BeforeFieldChange, AfterFieldChange, and AfterChanges) as if the value has been modified by the end user. Field calculations are useful for DB fields as well as for calculated fields. In this case, they are usually qualified by Condition expressions. Depending on the conditions, a field's value can be derived from a value stored in the database, or it can be calculated. For instance, if field A is non-empty, field B always returns the same value as A, but if A is empty, B can be set independently of A.

Field Properties

The Field properties include the following:

Property	Description
AllowDbNull	Gets or sets a value indicating whether null or empty string values are allowed in this field. If it is set to **False**, an attempt to assign null or empty string value to this field generates an exception.
AutoIncrement	Gets or sets a value indicating whether the field automatically receives an incremented value for a new row added to the table.
AutoIncrementSeed	Gets or sets the starting value for a field with AutoIncrement other than **None**.
AutoIncrementStep	Gets or sets the increment a field with AutoIncrement other than **None**.
Constraints	Returns the collection of constraints, ConstraintInfo objects. Constraints are evaluated (tested) when the value of the field changes. For a change to be successful, all constraint expressions (Expression) must evaluate to **True**. If one of the constraints is not satisfied, an exception is thrown. The exception message is determined by ErrorDescription. Constraints with Condition expression (if non-empty) evaluating to **False**, are skipped, not tested. See also, DataObjects for .NET Expressions for a description of **DataObjects for .NET** expression language.
DataSourceReadOnly	Gets or sets the value indicating whether the field value in the database can be changed. If this property is set to **True**, the field value will not be set updating the database. It also cannot be modified unless it is done in a newly added row, before the BeforeEndAddNew event (as with ReadOnlyUnlessNew = **True**).
DefaultValue	Gets or sets the default value, in string representation, for the field in a newly

	created row.
MaxLength	Gets or sets the maximum length of a string field, in characters. If the length is unlimited, the value is **0** (default).
PrimaryKey	Determines whether the field belongs to the table's primary key. It is determined automatically by **DataObjects for .NET Express** based on the database table structure.
Precision	For numeric fields (DataType is **Numeric**, **Decimal**, or **DbTimeStamp**), this property sets or gets the maximum number of digits representing values.
ReadOnly	Gets or sets a value indicating whether the field value can be changed by the end user or from event code. If set to **True**, an attempt to change the field throws an exception.
ReadOnlyUnlessNew	Gets or sets a value indicating whether the field value can be changed after the row has been added to the table (after AfterEndAddNew event). If set to **True**, an attempt to change the field throws an exception unless it is done in a newly added row, before the BeforeEndAddNew event.
Scale	For numeric fields (DataType is **Numeric**, **Decimal**, or **DbTimeStamp**), this property sets or gets the scale of numeric values, that is, how many digits to the right of the decimal point are used to represent values.
Unique	Gets or sets a value indicating whether the values of this field in each row must be unique. If it is set to **True**, an attempt to assign a duplicate value to this field generates an exception.

Programmatic Access to Data

To access data in code, use properties and methods of the C1.Data.C1DataTable object returned by the **DataTable** property. As with all programmatic classes in **DataObjects for .NET Express**, C1DataTable object model is based on the ADO.NET DataTable object model, so you will find its methods and properties familiar if you already know ADO.NET.

Here is a brief syntax description for most common programmatic tasks in accessing data in code (in Visual Basic, replace indexing brackets "[index]" with parentheses "(index)"):

- C1ExpressTable.DataTable.Rows.Count – number of rows in the table.
- C1ExpressTable.DataTable.AddNew() – adds a new row to the table.
- C1ExpressTable.DataTable.Rows[index].Delete() – deletes a row from the table.
- C1ExpressTable.DataTable.Rows[index] – a data row, a C1DataRow object.

To get/set field value in a row, (C1DataRow object), use row[field_name] or row[field_index], or, with full access path, C1ExpressTable.DataTable.Rows[row_index] [field_index].

- C1DataRow.GetChildRow(s) – gets child rows with respect to a master-detail relation.
- C1DataRow.GetParentRow(s) – gets parent rows with respect to a master-detail relation.
- C1DataRow.Modified – returns **true**, if the row has been modified.
- C1DataRow.RowState – returns one of the DataRowState enumeration values: **Unchanged**, **Modified**, **Added**, **Deleted**, or **Detached**.
- C1DataRow.BeginEdit/EndEdit/CancelEdit – start/end/rollback edit mode for a row.

Customizing Data Logic with Events

The **C1ExpressTable** component supports a rich set of events allowing you to customize data behavior in code. Following is a brief list of C1ExpressTable events (we omit prefixes Before and After pertaining to most events, retain the prefix only if an event occurs only Before or only After). For a full description of C1ExpressTable events see corresponding reference sections and Business Logic Events.

Event	Description
AddNew	Fired when a new (empty) row is added.
AfterChanges	Fired when all changes initiated by a field change are done and handled in code, see the **FieldChange** event.
BeginEdit	Fired when the user starts editing a row (data-bound controls start editing a row immediately after they position on it, even though no changes have been made yet).
CancelEdit	Fired when the user cancels editing a row reverting the changes made to it.
CurrentRowChanged	Fired when there are changes in the current row, whatever their cause, resulting from field change, current row change or complete data refresh. The CurrentRowChanged event is useful in scenarios such as synchronizing detail data with the master row on every change occurring in the master row.
Delete	Fired when a row is deleted.
EndAddNew	Fired when a newly added row becomes a regular row in the rowset. When a row is added, it is added empty; its primary key is unknown. A row with unknown primary key is in special transitory state, it is not a regular rowset row. Only after its primary key is set it becomes a regular (added) row, which is signaled by this event.
EndEdit	Fired when the user finishes editing a row (data-bound controls finish editing a row when they leave that row, even if no changes have been made).
Error	Fired when an exception (error condition) occurs. Gives the programmer an opportunity to handle exceptions (so normal execution can be resumed), show custom error messages and customize/localize error message texts.
FieldChange	Fired when a field value is set. Inside this event, your code can set other fields triggering recursive **FieldChange** events, **DataObjects for .NET Express** handles this situation correctly. Only after all changes are done and handled, **AfterChanges** event is triggered.
Fill	**BeforeFill** and **AfterFill** events are fired both in C1ExpressTable (for each table) and in C1ExpressConnection (once for all tables attached to it) before and after data is fetched from the database.
FirstChange	Fired when a first change is made to the row (a field value changed) after BeginEdit.
GenerateSQL	Fired before/after executing an SQL command to fetch data from the database. Can be used to customize the SQL SELECT statement used in that command.
PositionChanged	Fired when a new row becomes current. Current row is the row on which the end user is currently positioned (controlled by a CurrencyManager, see the System.Windows.Forms.CurrencyManager class in the .NET Framework documentation for more information).
UpdateError	Fired when an error occurs in database update, allowing to resolve the error

	condition. This event is fired both in C1ExpressTable and C1ExpressConnection components. For details on the database update process and how it can be customized with event code, including reconciling multi-user concurrency conflicts, see Updating the Database in **DataObjects for .NET** documentation.
UpdateRow	Fired when a modified row is committed (updated) to the database. This event is fired in a C1ExpressConnection component. For details on the database update process and how it can be customized with event code, see Updating the Database in **DataObjects for .NET** documentation.
Update	Fired before/after all modified rows are committed (updated) to the database. This event is fired in a C1ExpressConnection component.

C1ExpressConnection: Combining Tables into Data Sets

A C1ExpressConnection component is used when it is necessary to combine several tables into a single data set. C1ExpressTable components are attached to a C1ExpressConnection component by setting their ConnectionComponent property. These tables share a common database connection and a common row cache (data set). They can share the same data, for instance, if two C1ExpressTable components represent the same database table, or if one of them is a composite table including a database table that is also used in the other. In this case, if they share the same connection, changing data in one of them will be reflected in all others that include the affected rows. Updating modified data to the database is done for all tables sharing the connection as a whole, in one transaction. This is why the Update method belongs to the C1ExpressConnection component.

Conversely, two or more standalone C1ExpressTable components, not attached to a common C1ExpressConnection, use separate database connections and have completely independent copies of data (row cache).

Defining Relations

C1ExpressConnection component is also used for defining relations between tables. To open the **Relations editor**, select **Edit Relations** from the context menu or press the **Relations** property button.

To create a relation, select parent and child tables in corresponding combo boxes (if one or both of them are composite tables, you also need to select a constituent simple table in the "use fields of" combo box below). Then add one or more joins – equalities between a parent field and a child field.

Relations are used for two purposes master-detail relations and referential integrity (foreign key constraints).

Master-Detail Relations

When master-detail relations are defined between tables in a **C1ExpressConnection** component, they can be used to build data-bound master-detail forms and to navigate parent/child rows programmatically (with GetChildRow(s)/GetParentRow(s) methods). To create a master-detail relation, check the **Master-detail** check box in **Relations editor**.

Having a master-detail relation, you can bind two data-aware controls, one to the master and the other to the detail, and the detail control will follow the master, will be automatically populated with child rows of the parent row on which the master is currently positioned. In presence of master-detail relations, the **DataMember** combo box of a data-aware control shows tables participating in the master-detail hierarchy preceding with underscore, whereas the same tables regarded as standalone (not restricted by master-detail relations) are shown without underscore. For example, if you have a "Customers – Orders" relation, this is how you can bind a master and a detail grid:

To write code in Visual Basic

Visual Basic

```
ParentGrid1.DataMember = "_Customers"
ParentGrid1.DataSource = C1ExpressConnection1
ChildGrid1.DataMember = "_Customers.Customers-Orders"
ChildGrid1.DataSource = C1ExpressConnection1
```

To write code in C#

C#

```
parentGrid1.DataMember = "_Customers";
parentGrid1.DataSource = C1ExpressConnection1;
childGrid1.DataMember = "_Customers.Customers-Orders";
childGrid1.DataSource = C1ExpressConnection1;
```

Tip: If you want to add related rows to the master and detail tables at once in a C1ExpressConnection, use a connection type other than OleDb. For native SQL Server connectivity select **SQLServer** for the connection type. For native Oracle access use either **Oracle** or **MSOracle** connection type.

Enforcing Referential Integrity (Foreign Key) Constraints

For instance, in the Orders table or in a composite table *CustomersOrders* we would like to disallow assigning a value to *CustomerID* that is not present in the *Customers* table. To create a referential integrity constraint relation, check the **Enforce constraints** check box. A relation can be a master-detail relation and a referential integrity constraint simultaneously.

By default, M-1 joins in a composite table define a referential integrity constraint. So, for example, in a composite table *CustomersOrders* it is not allowed to enter *CustomerID* that is not present in the *Customers* table. If you want to change this behavior, define a relation with the same joins as used in the composite table and uncheck the **Enforce constraints** check box.

Relation Properties

Relation properties include the following:

Property	Description
GetRowsEvent	This property is **False** by default. If set to **True**, the relation becomes custom. That means that it is not based on joins, equalities between table fields, but child rows are obtained by code in the GetChildRows event. It is used when you need a more complicated algorithm than simple equality of table fields. For example, see the **CustomRelations** sample in the **Samples** directory, where a custom relation is used to represent a many-to-many relation that cannot be based on a single simple relation between tables.
DeleteCascadeRule	This property specifies what happens to child rows when their parent row is deleted. **None** (default) means that child rows are left unchanged. As a result of deleting the parent row, they become orphan rows, without parent. The **Cascade** value means that child rows are deleted. Set this property to **Cascade** when child rows belong to their parent, should not exist without it. There are also two less frequently used values: **SetNull** meaning that child rows remain with related field(s) set to **Null**, and **SetDefault** meaning that child rows remain with related field(s) set to its default value.
UpdateCascadeRule	This property determines what happens to child fields when parent field values are

	changed. If it is set to **Cascade** (default), the child fields are changed correspondingly. If it is set to **None**, the child fields are left unchanged. If it is set to **SetNull**, the child fields are set to **Null**. If it is set to **SetDefault**, the child fields are set to their default values.

C1ExpressView: Filtering, Sorting and Working with Tables in Other Forms

C1ExpressView is a component that can serve as a data source for data-aware controls, along with C1ExpressTable and C1ExpressConnection. It provides a view of a C1ExpressTable. This view can include only rows satisfying certain filter conditions, if such conditions are specified. This view can also sort the rows in a certain order. All these views are independent from one another, so you can show the same data filtered and sorted according to different conditions. Also, each C1ExpressView, being an independent data source, maintains its own current row, so you can use several views with a single C1ExpressTable if you need several representations with independent current row.

C1ExpressView also serves to allow binding of data-aware controls in one form to C1ExpressTable components residing in another form, see Working with Tables in Other Forms.

To attach a C1ExpressView to a C1ExpressTable component, set the ExpressTable property (or ExpressTableName if you need to attach it to a C1ExpressTable residing in another form).

To sort C1ExpressView rows, set the Sort property to the sort field name. To sort by multiple fields use coma as the delimiter:

To filter C1ExpressView rows, restrict the C1ExpressView rowset to only those rows that satisfy a certain condition, set the RowFilter property to a filter expression. For example,

To write code in Visual Basic

```
Visual Basic
C1ExpressView.RowFilter = "City = 'London'"
```

To write code in C#

```
C#
c1ExpressView.RowFilter = "City = 'London'";
```

See DataObjects for .NET Expressions for a description of **DataObjects for .NET** expression language.

C1ExpressView also allows to filter by row state, showing, for example, only added rows, or only modified rows, or deleted rows (note that this is the only way to access deleted rows), or original rows (original rows are the rows with their respective field values as they were fetched from the database, before any user modifications made to them). To filter rows by row state, use the RowFilter property which can have one of the following values (default: **Current**):

Value	Description
Added	New rows.
CurrentRows	Current rows including unchanged, new, and modified rows.
Deleted	Deleted rows.
ModifiedCurrent	Current rows excluding new and unchanged rows.
ModifiedOriginal	Original rows excluding deleted and unchanged rows.
OriginalRows	Original rows including deleted and unchanged rows.

Unchanged	Unchanged rows.

Working with Tables in Other Forms

Using C1ExpressView you can bind data-aware controls to the data of a C1ExpressTable residing in a different form. To attach a C1ExpressView component to a C1ExpressTable component from a different form, open the form containing the C1ExpressTable component and select the component name in the ExpressTableName combo box. The ExpressTableName combo box contains all C1ExpressTable components in the forms currently open in the environment. Having selected an ExpressTableName value, you will see that the ExpressTable property has changed correspondingly. After that, you can bind data-aware controls to the C1ExpressView component.

DataObjects for .NET Express Design-Time Support

DataObjects for .NET Express provides customized context menus, smart tags, and a designer that offers rich design-time support and simplifies working with the object model. The following sections describe how to use **DataObjects for .NET Express**' design-time environment to configure the **DataObjects for .NET Express** components.

Tasks Menus

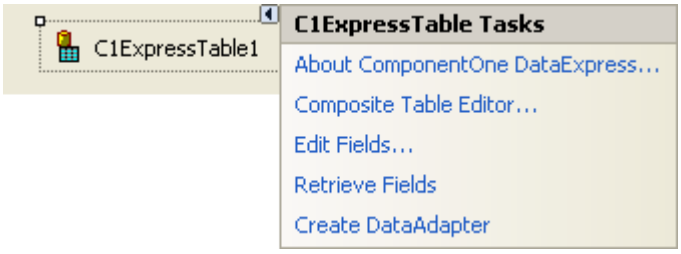
A smart tag represents a short-cut tasks menu that provides the most commonly used properties in each component. You can invoke each component's tasks menu by clicking on the smart tag (▶) in the upper-right corner of the component.

Properties Window

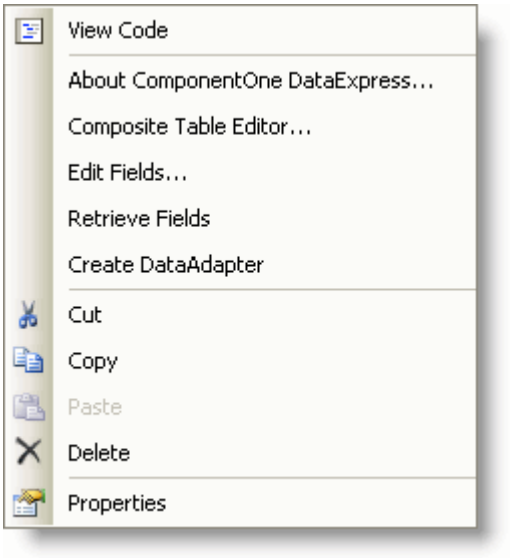
You can also easily configure **DataObjects for .NET Express** at design time using the Properties window in Visual Studio. You can access the Properties window by right-clicking the control and selecting **Properties**.

C1ExpressTable Tasks and Context Menus

You can access the **C1ExpressTable Tasks** menu by clicking the smart tag in the upper-right corner of the C1ExpressTable component.



You can access the C1ExpressTable context menu by right-clicking the C1ExpressTable component.



About DataExpress

Clicking **About** displays the **DataObjects for .NET Express**' **About** dialog box, which is helpful in finding the build number of the component.

Composite Table Editor

Clicking **Composite Table Editor** opens the **Composite Table Editor** dialog box which allows you to specify a composite table. In the **Composite Table Editor**, you can add database tables constituting the composite table.

Edit Fields

Clicking **Edit Fields** opens the **Fields editor** where you can add and delete fields.

Retrieve Fields

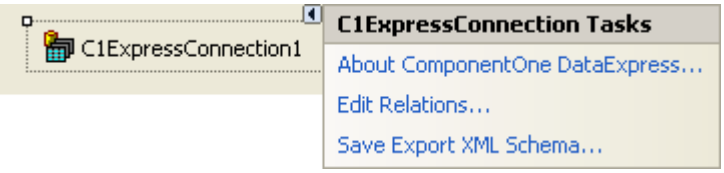
Clicking **Retrieve Fields** will retrieve table fields from the database. If you want to restore the collection of table view fields to its initial state, select the table view, and select **Retrieve Fields** from the context menu.

Create DataAdapter

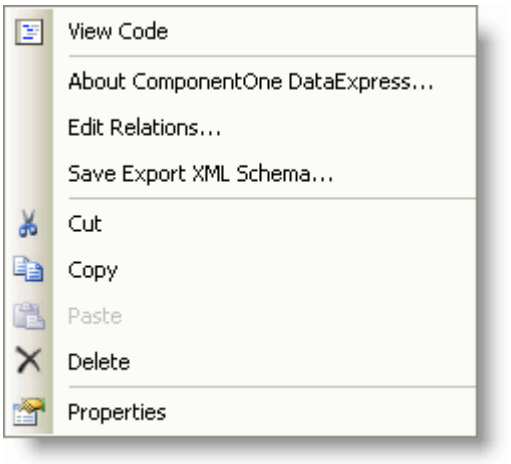
Clicking **Create DataAdapter** creates a **DataAdapter** in a C1ExpressTable component associated with the table. The **DataAdapter** component will then perform both fetch and update without custom code (but you can customize the default fetch and update behavior in event code if needed).

C1ExpressConnection Tasks and Context Menus

You can access the **C1ExpressConnection Tasks** menu by clicking the smart tag in the upper-right corner of the C1ExpressConnection component.



You can access the C1ExpressConnection context menu by right-clicking the C1ExpressConnection component.



About DataExpress

Clicking **About** displays the **DataObjects for .NET Express**' **About** dialog box, which is helpful in finding the build number of the component.

Edit Relations

Clicking **Edit Relations** opens the **Relations editor** where you can define relations between tables. To create a relation, select parent and child tables in corresponding combo boxes (if one or both of them are composite tables, you also need to select a constituent simple table in the "use fields of" combo box below). Then add one or more joins – equalities between a parent field and a child field.

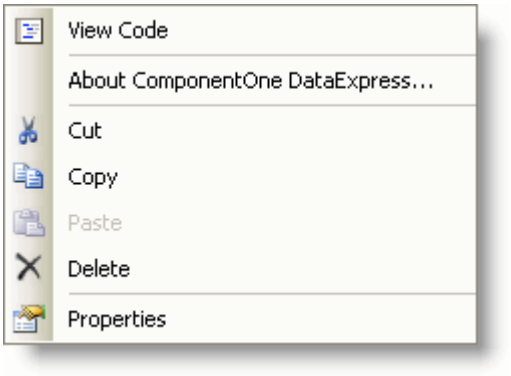
Save Export XML Schema

Clicking **Save Export XML Schema** opens the **Save schema to a file** dialog box which you can use to export data from a C1ExpressConnection to XML. See Exporting Data from a C1DataSet to XML for more information.

C1ExpressView Tasks and Context Menus

You can access the **C1ExpressView Tasks** menu by clicking the smart tag in the upper-right corner of the C1ExpressView component.

You can access the C1ExpressView context menu by right-clicking the C1ExpressView component.



About DataObects

Clicking **About** displays the **DataObjects for .NET Express**' **About** dialog box, which is helpful in finding the build number of the component.

Notes for Users of DataObjects for .NET Enterprise Edition

If you are familiar with **DataObjects Enterprise Edition**, it may be helpful to understand the relationship between **DataObjects for .NET** and **DataObjects for .NET Express**.

DataObjects for .NET Express is a simplified version of **DataObjects for .NET**. It makes working with data very simple and straightforward and it is easier to master than **C1DataObject Enterprise Edition**. **DataObjects for .NET Express** is intended for small to medium projects where full power of **DataObjects for .NET** is not required.

DataObjects for .NET Express resides in a separate assembly, C1.Data.Express.2.dll that uses the **DataObjects for .NET** assembly C1.Data.2.dll. **DataObjects for .NET Express** added three components to **DataObjects for .NET**: C1ExpressTable, C1ExpressConnection, and C1ExpressView.

DataObjects for .NET Express always works in direct client mode (see Direct Client for more information), it does not support 3-tier configuration and data libraries.

DataObjects for .NET Express does not use the **Schema Designer** and does not require creating C1DataSet components on user forms. A C1DataSet object and a corresponding schema is generated automatically for each C1ExpressConnection component and can be accessed through its DataSet property:

For a C1ExpressConnection component with attached C1ExpressTable components:

To write code in Visual Basic

Visual Basic
`C1ExpressConnection.DataSet.Schema`

To write code in C#

C#
`c1ExpressConnection.DataSet.Schema;`

For a standalone C1ExpressConnection component:

To write code in Visual Basic

Visual Basic
`C1ExpressTable.ExpressConnection.DataSet.Schema`

To write code in C#

C#

```
c1ExpressTable.ExpressConnection.DataSet.Schema;
```

DataObjects for .NET Express Tutorials

If you are running the pre-built tutorial projects included in **DataObjects for .NET Express** installation, please be aware that the projects have the sample database location hardcoded in the connection string.

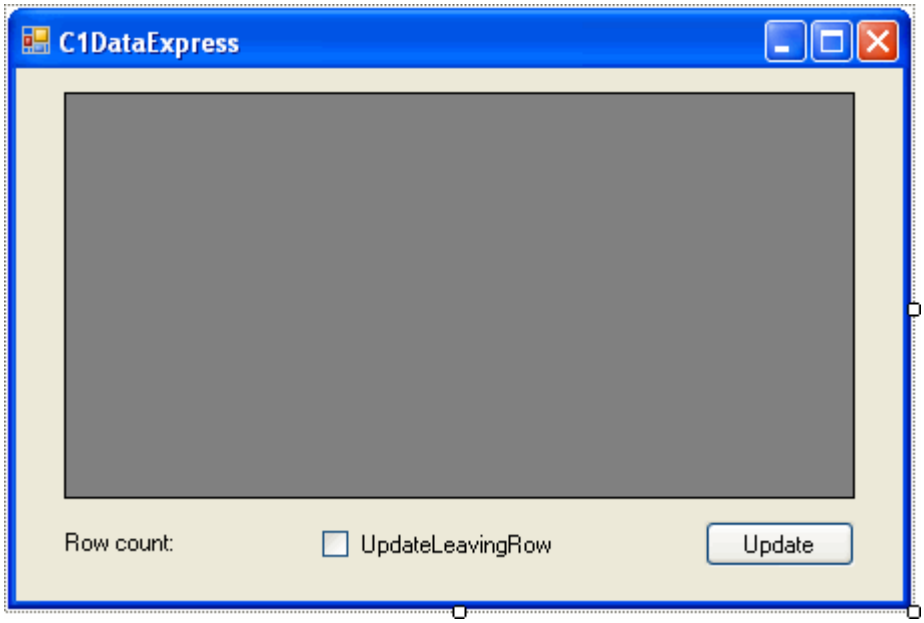
If you have the Northwind database (standard MS Access sample database included in Visual Studio) installed in a different location, you can either change the connection string in tutorial projects or copy the **C1NWind.mdb** file to the required location.

Tutorial 1: Binding to a Simple Table

In this tutorial, you will see how easy it is to bind to simple database table data using **DataObjects for .NET Express Edition**. Complete the following steps:

1. Create a new .NET 2.0 Windows Application project.1. Place the following components on the form as shown in the figure.

Number of Components	Name	Namespace
1 C1ExpressTable	C1ExpressTable1	C1.Data.Express.C1ExpressTable
1 DataGridView	DataGridView1	System.Windows.Forms.DataGridView
1 Label	Label1	System.Windows.Forms.Label
1 CheckBox	CheckBox1	System.Windows.Forms.CheckBox
1 command Button	Button1	System.Windows.Forms.Button

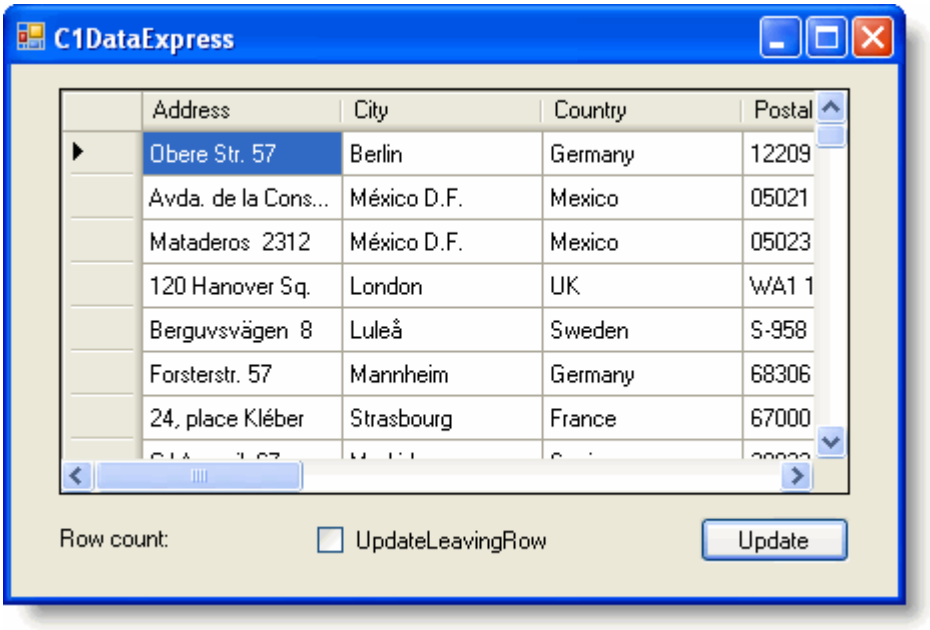


In this tutorial we intentionally use Microsoft DataGridView, not a ComponentOne grid control. It is meant to

show that **DataObjects for .NET Express** can serve as a data source to any data-bound GUI controls adhering to .NET data binding specification. For all other tutorials will use **True DBGrid**. Naturally, we recommend **C1TrueDBGrid** and other presentation controls, both because they are the best and because they are most closely integrated with **DataObjects for .NET Express**. Most **DataObjects for .NET Express** features, all features that rely only on standard .NET data binding, will work with any third-party data-bound control. However, one important feature, Virtual Mode, requires that you use a ComponentOne data bound grid (if you need a grid at all).

2. Set the **Text** properties of the **CheckBox** and **Button** to the following:
   - CheckBox1.Text = **UpdateLeavingRow**
   - Label1.Text = **Row count:**
   - Button1.Text = **Update**
3. Select the **C1ExpressTable1** component.
4. Click the drop-down arrow next to the ConnectionString property in the **Properties** window and select **<New Connection...>**. The **Add Connection** dialog box opens.
5. Select the provider, the database and other necessary connection properties in that dialog box. In this tutorial, we use the standard MS Access Northwind sample database (C1NWind.mdb).
   1. Click the **Change** button, if necessary, and select **Microsoft Access Database File**. The **.NET Framework Data Provider for OLE DB** is selected under **Data provider**.
   2. Under Database file name, browse to locate the C1NWind.mdb. The database is installed in the Common folder of the ComponentOne Samples directory.
   3. Click **OK** to close the **Add Connection** dialog box.
6. Click the drop-down arrow next to the DbTableName property and select **Customers** from the database table list. Click **Yes** to retrieve the fields.
7. Select the **DataGridView1** control, click the drop-down arrow next to the **DataSource** property in the **Properties** window and. select **C1ExpressTable1**. This binds the grid to the C1ExpressTable, and the **Customers** fields appear in the grid.

Run the program and observe the following:



- The grid shows the Customers data. Creating a basic data-bound form with **C1Express** has been as easy as setting two properties in a single C1ExpressTable component and setting the **DataSource** property of a data-bound control.

Close the program and return to the design time environment. Next you'll access **DataObjects for .NET Express** data

programmatically in code.

Continue by completing the following steps:

1. Create an event handler, **C1ExpressTable1_AfterFill**, and enter the following code:

   To write code in Visual Basic

   ```
   Visual Basic
   Label1.Text = "Row count: " + c1ExpressTable1.DataTable.Rows.Count.ToString()
   ```

   To write code in C#

   ```
   C#
   label1.Text = "Row count: " + c1ExpressTable1.DataTable.Rows.Count.ToString();
   ```

   The AfterFill event occurs after the table is filled with data. The code shows the number of rows in the table in a label control.
2. Create an event handler, **Button1_Click**, and enter the following code:

   To write code in Visual Basic

   ```
   Visual Basic
   C1ExpressTable1.ExpressConnection.Update()
   ```

   To write code in C#

   ```
   C#
   c1ExpressTable1.ExpressConnection.Update();
   ```

   This code sends changes made by the end user to the database.
3. Create an event handler, **CheckBox1_CheckStateChanged**, and enter the following code:

   To write code in Visual Basic

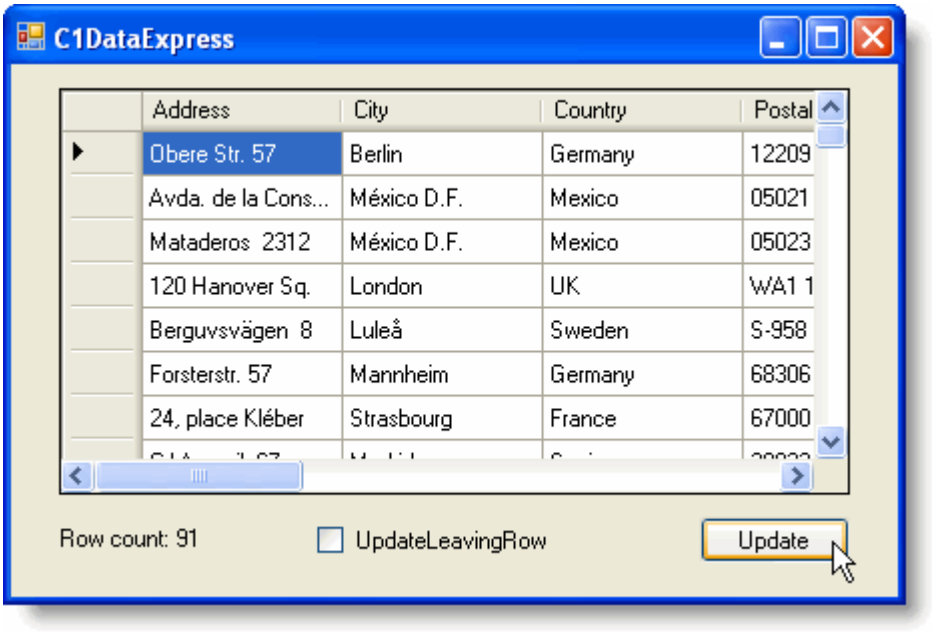
   ```
   Visual Basic
   C1ExpressTable1.UpdateLeavingRow = checkBox1.Checked
   ```

   To write code in C#

   ```
   C#
   c1ExpressTable1.UpdateLeavingRow = checkBox1.Checked;
   ```

   This code toggles the UpdateLeavingRow property value when the check box is checked or unchecked.

Run the program and observe the following:



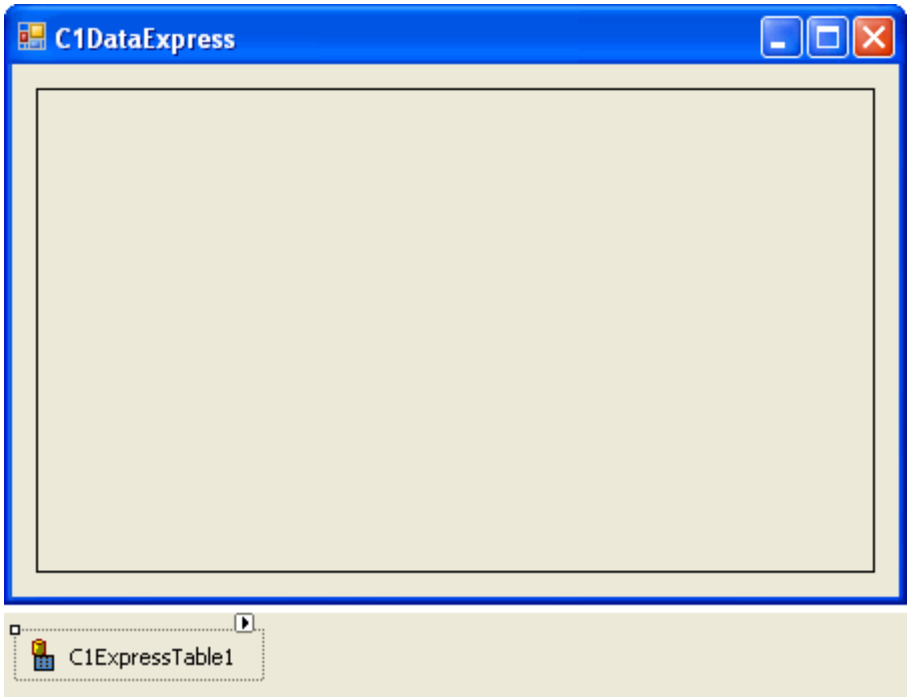
- The label now details the number of rows in the table.
- If the check box is unchecked, changes made to data in the grid are not saved in the database unless you press the **Update** button, at which point the changes are saved. You can observe this by closing and re-opening the program.
- If the check box is checked, changes are automatically saved in the database when you modify a row and leave it for another row either with the keyboard or with the mouse.

Tutorial 2: Creating a Composite Table

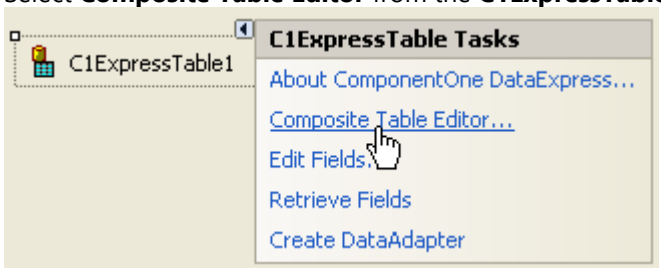
In this tutorial, you will learn how to work with composite tables in **DataObjects for .NET Express**. Complete the following steps:

1. Create a new .NET 2.0 Windows Application project.
2. Place the following components on the form and arrange them as shown in the image below:

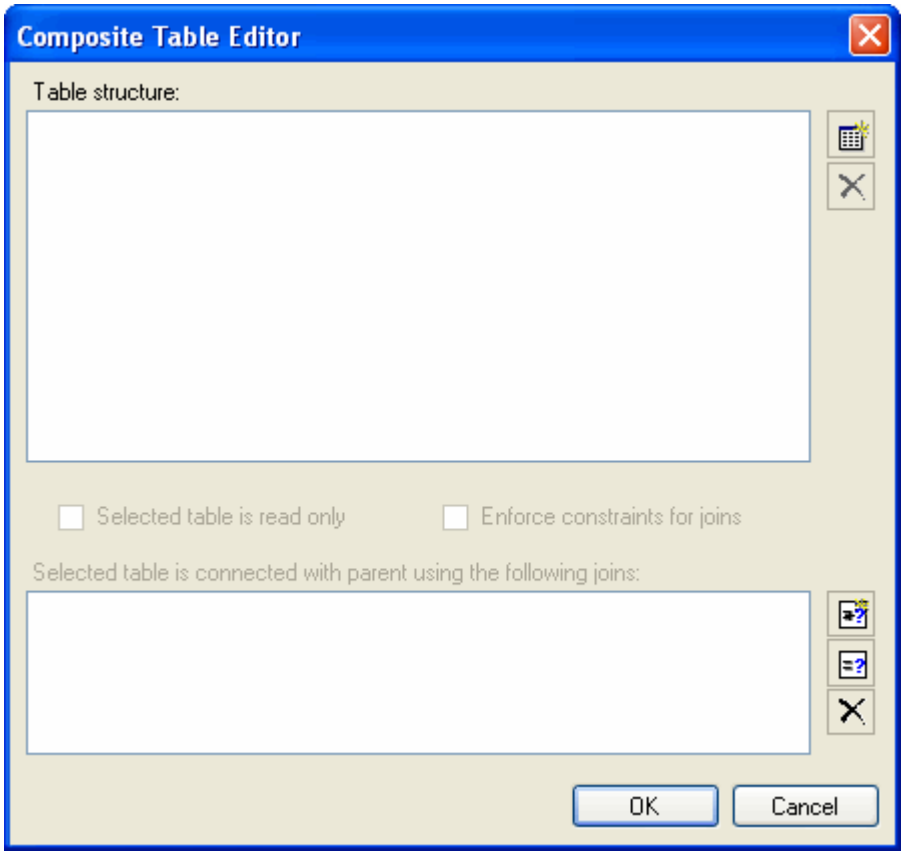
Number of Components	Name	Namespace
1 C1ExpressTable	C1ExpressTable1	C1.Data.Express.C1ExpressTable
1 C1TrueDBGrid	C1TrueDBGrid1	C1.Win.C1TrueDBGrid.C1TrueDBGrid

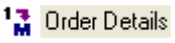


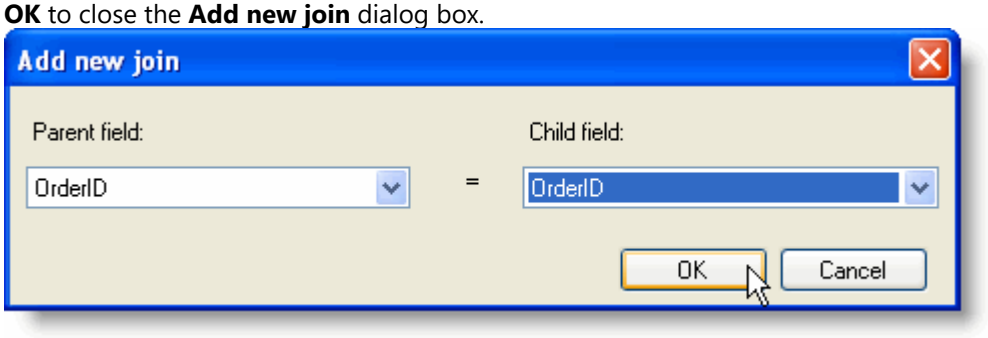
3. Select the **C1ExpressTable1** component.
4. Click the drop-down arrow next to the ConnectionString property in the **Properties** window and select **<New Connection...>**. The **Add Connection** dialog box opens.
5. Select the provider, the database and other necessary connection properties in that dialog box. In this tutorial, we use the standard MS Access Northwind sample database (C1NWind.mdb)
    1. Click the **Change** button, if necessary, and select **Microsoft Access Database File**. The **.NET Framework Data Provider for OLE DB** is selected under **Data provider**.
    2. Under **Database file name**, browse to locate the **C1NWind.mdb**. The database is installed in the Common folder of the **ComponentOne Samples** directory
    3. Click **OK** to close the **Add Connection** dialog box.
6. Select **Composite Table Editor** from the **C1ExpressTable Tasks** menu.



Alternatively, you can click the drop-down arrow next to the DbTableName property and select **<Composite...>** from the database table list. The **Composite Table Editor** appears.



7. In the **Composite Table Editor**, click the **Add table** button  . In the **Add table** dialog box, select the **Orders** table and click **OK**.
8. Open the **Add table** dialog box again and select **Order Details**. Notice that the 1-M relation radio button is selected.
9. Click **OK**. Notice that the **Order Details** table has been added to the composite table attached to its parent with a 1-M, or one-to-many relation  .
10. Click the **Add join** button  and select **OrderID** in the **Parent** field and **OrderID** in the **Child** field. Then click **OK** to close the **Add new join** dialog box.



The following join connects the **Order Details** table to its parent **Orders**:

```
Orders.OrderID = Order Details.OrderID
```

11. Click the **Add table** button again and select **Products**.
12. Select the **using many-to-one (M-1)** relation radio button and click **OK**. Notice that the **Products** table has been added to the composite table attached to its parent with an M-1, or many-to-one relation  .
13. Click the **Add join** button  and select **ProductID** in the **Parent** field and **ProductID** in the **Child** field. Then click **OK** to close the **Add new join** dialog box.

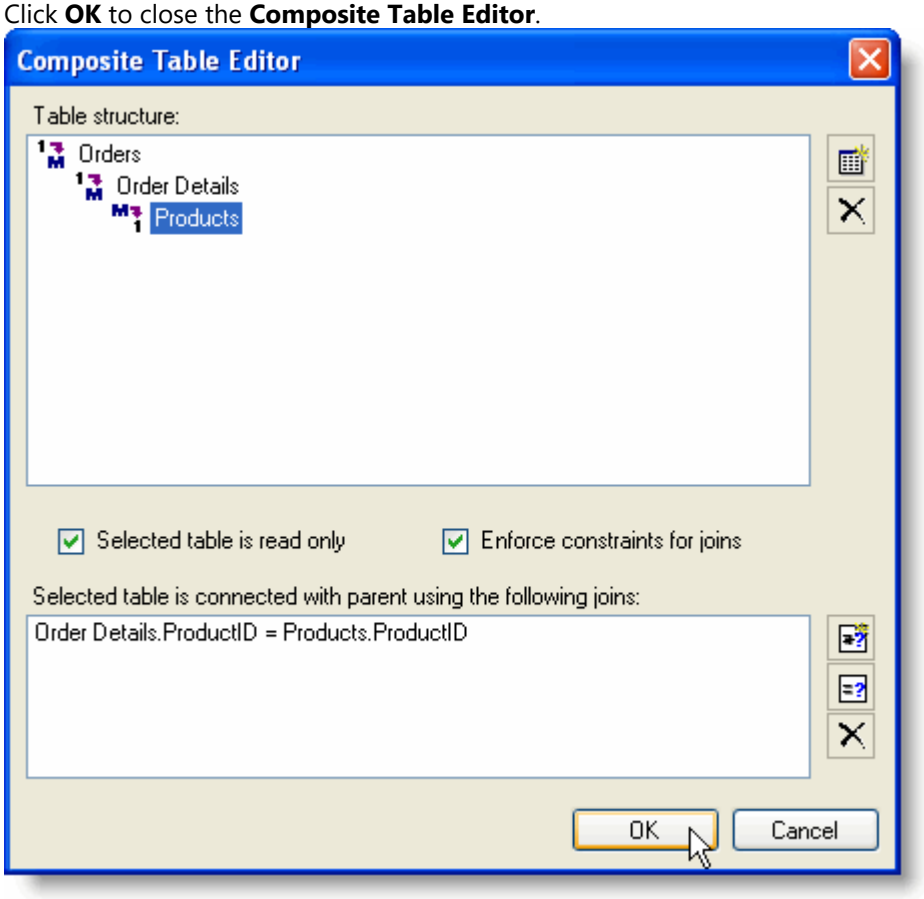
The following join connects the **Products** table to its parent **Orders**:

```
Order Details.ProductID = Products.ProductID
```

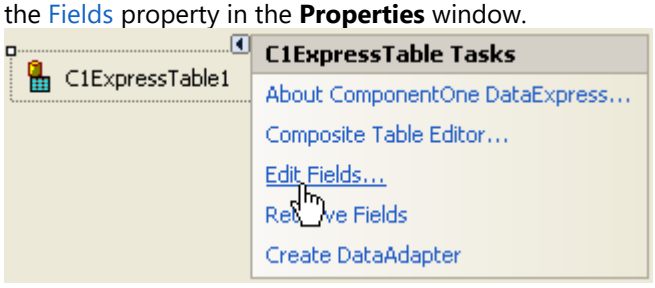
The overall table structure looks like this:

```
Orders (1->M) Order Details (M->1) Products
```

14. Click **OK** to close the **Composite Table Editor**.

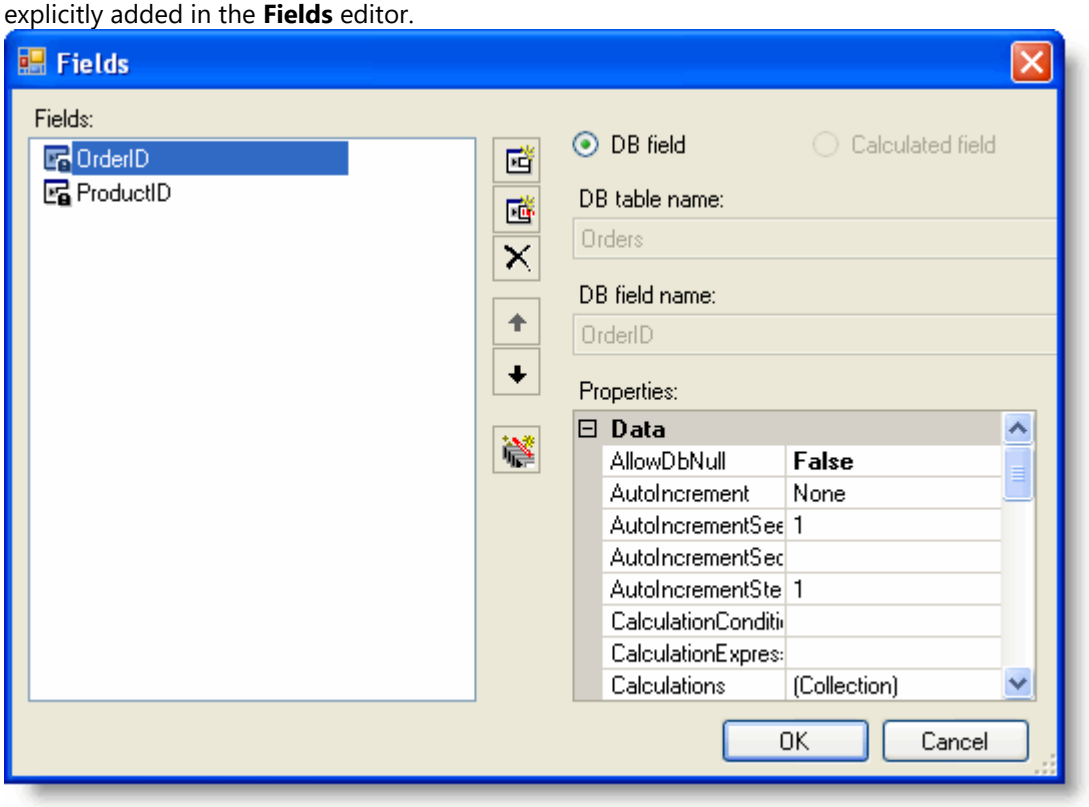


15. Select **Edit Fields** from the **C1ExpressTable Tasks** menu. Alternatively, you can click the **ellipsis** button next to the Fields property in the **Properties** window.

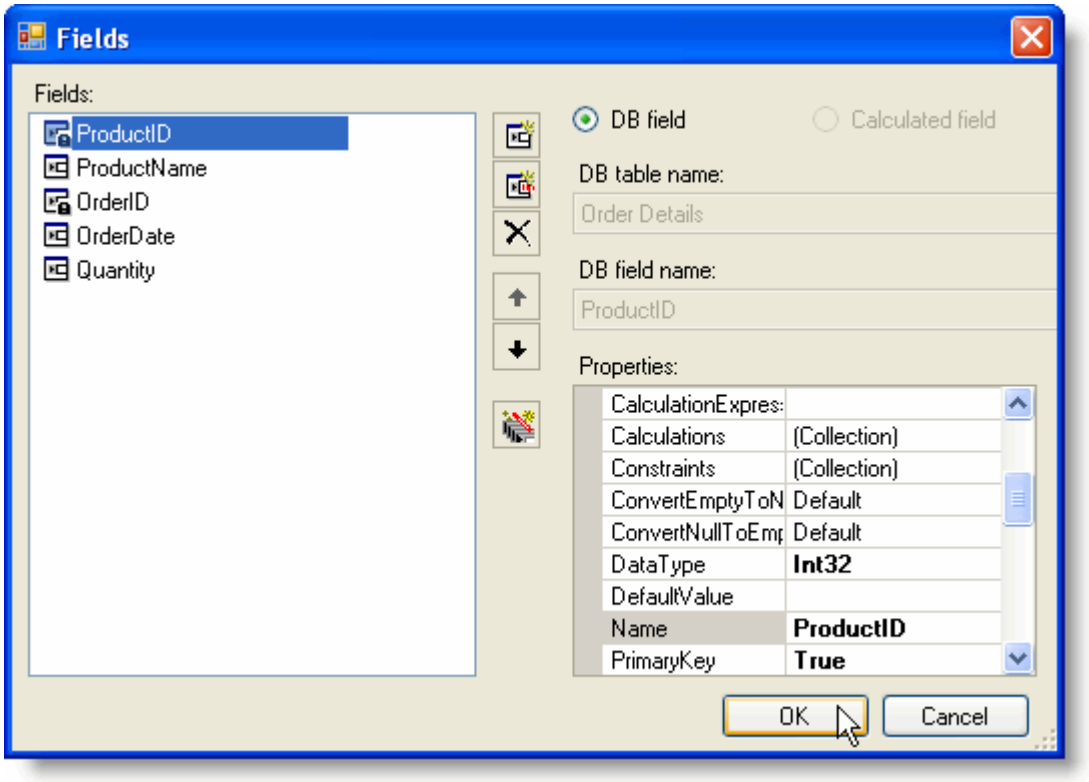


The **Fields** editor appears. Only two fields appear in the editor: **OrderID** and **ProductID**. These are primary key fields of the constituent DB tables; they are always present in the composite table fields. Other fields must be

explicitly added in the **Fields** editor.

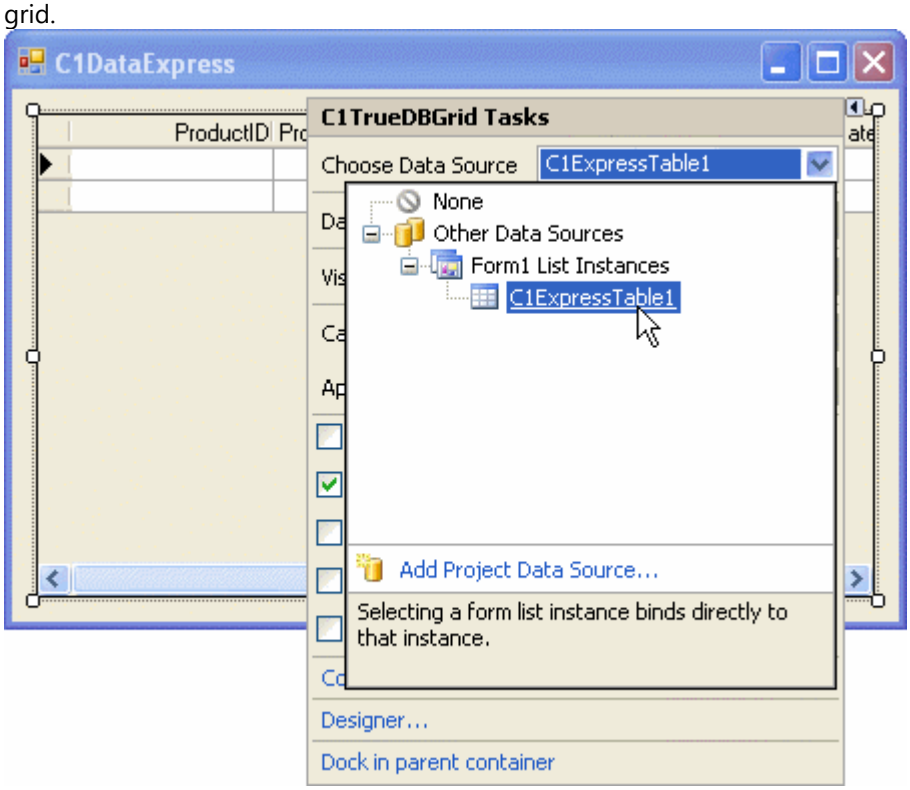


16. Add the following fields using the **Add DB field** button : **Orders.OrderDate**, **Products.ProductName**, **Order Details.Quantity**.
17. Rearrange the order of fields to your liking using the **Move up** and **Move down** buttons and click **OK** to close the **Fields** editor.

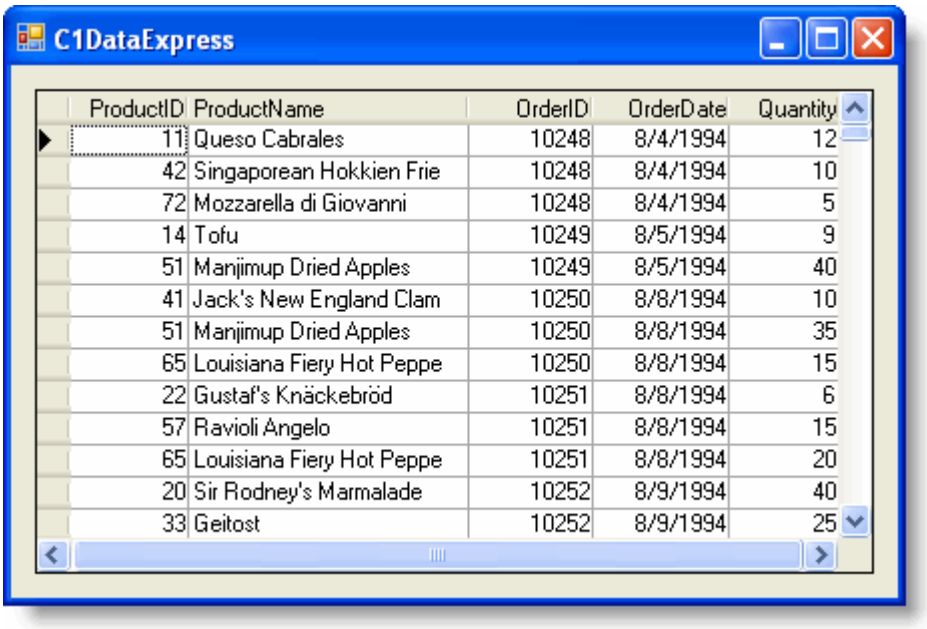


18. Select the **C1TrueDBGrid** control, open the **C1TrueDBGrid Tasks** window, and set the **Data Source** to **C1ExpressTable1**. This binds the grid to the **C1ExpressTable**, and the composite table fields appear in the

grid.



Run the program and observe the following:



ProductID	ProductName	OrderID	OrderDate	Quantity
11	Queso Cabrales	10248	8/4/1994	12
42	Singaporean Hokkien Frie	10248	8/4/1994	10
72	Mozzarella di Giovanni	10248	8/4/1994	5
14	Tofu	10249	8/5/1994	9
51	Manjimup Dried Apples	10249	8/5/1994	40
41	Jack's New England Clam	10250	8/8/1994	10
51	Manjimup Dried Apples	10250	8/8/1994	35
65	Louisiana Fiery Hot Peppe	10250	8/8/1994	15
22	Gustaf's Knäckebröd	10251	8/8/1994	6
57	Ravioli Angelo	10251	8/8/1994	15
65	Louisiana Fiery Hot Peppe	10251	8/8/1994	20
20	Sir Rodney's Marmalade	10252	8/9/1994	40
33	Geitost	10252	8/9/1994	25

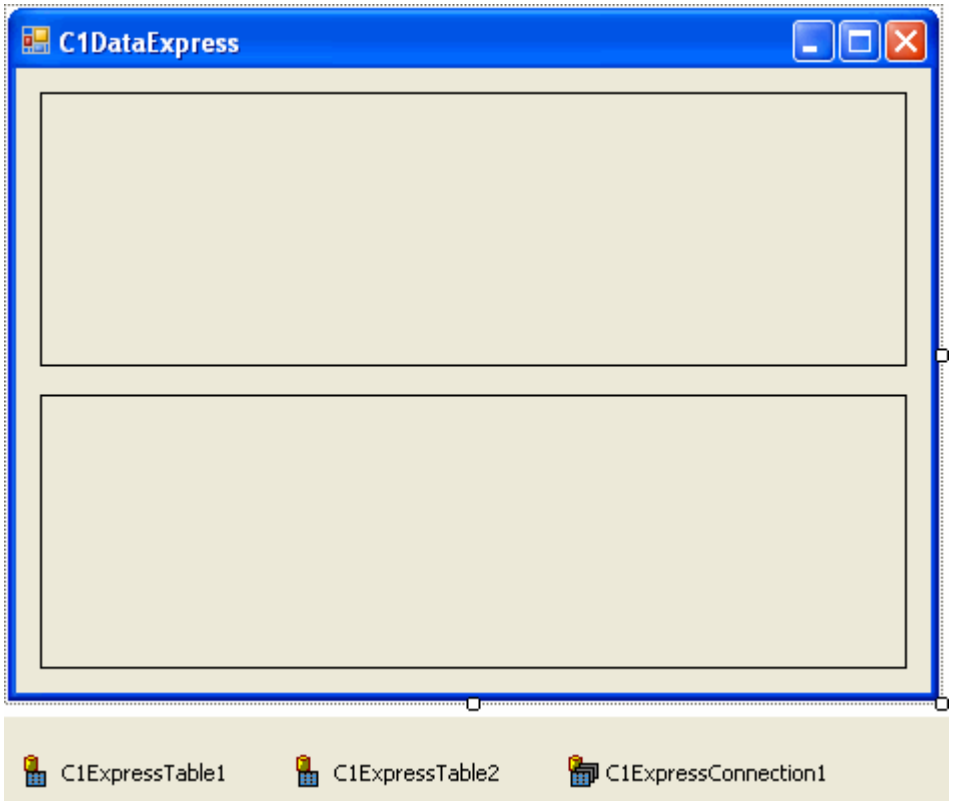
- The grid shows the composite table rows containing fields from all three constituent DB tables.
- Change a value in the *ProductID* column and observe that the corresponding value of the *ProductName* column changes accordingly. **DataObjects for .NET Express** not only populates composite tables automatically (generating SQL statements behind the scenes), but unlike other data frameworks, including ADO.NET, it remains "structure-aware", preserves the structure of the composite table throughout end user modifications.

Tutorial 3: C1ExpressConnection and Master-Detail Relations

In this tutorial, you will see how to combine multiple tables into a data set, how to specify master-detail relations and to build a master-detail data-bound form. Complete the following steps:

1. Create a new .NET 2.0 Windows Application project.
2. Place the following components on the form as shown in the figure.

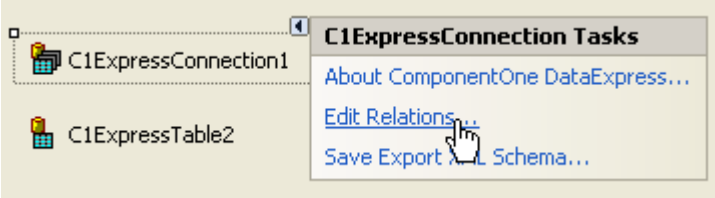
Number of Components	Name	Namespace
2 C1ExpressTable	C1ExpressTable1 C1ExpressTable2	C1.Data.Express.C1ExpressTable
1 C1ExpressConnection	C1ExpressConnection1	C1.Data.Express.C1ExpressConnection
2 C1TrueDBGrid	C1TrueDBGrid1 C1TrueDBGrid2	C1.Win.C1TrueDBGrid.C1TrueDBGrid



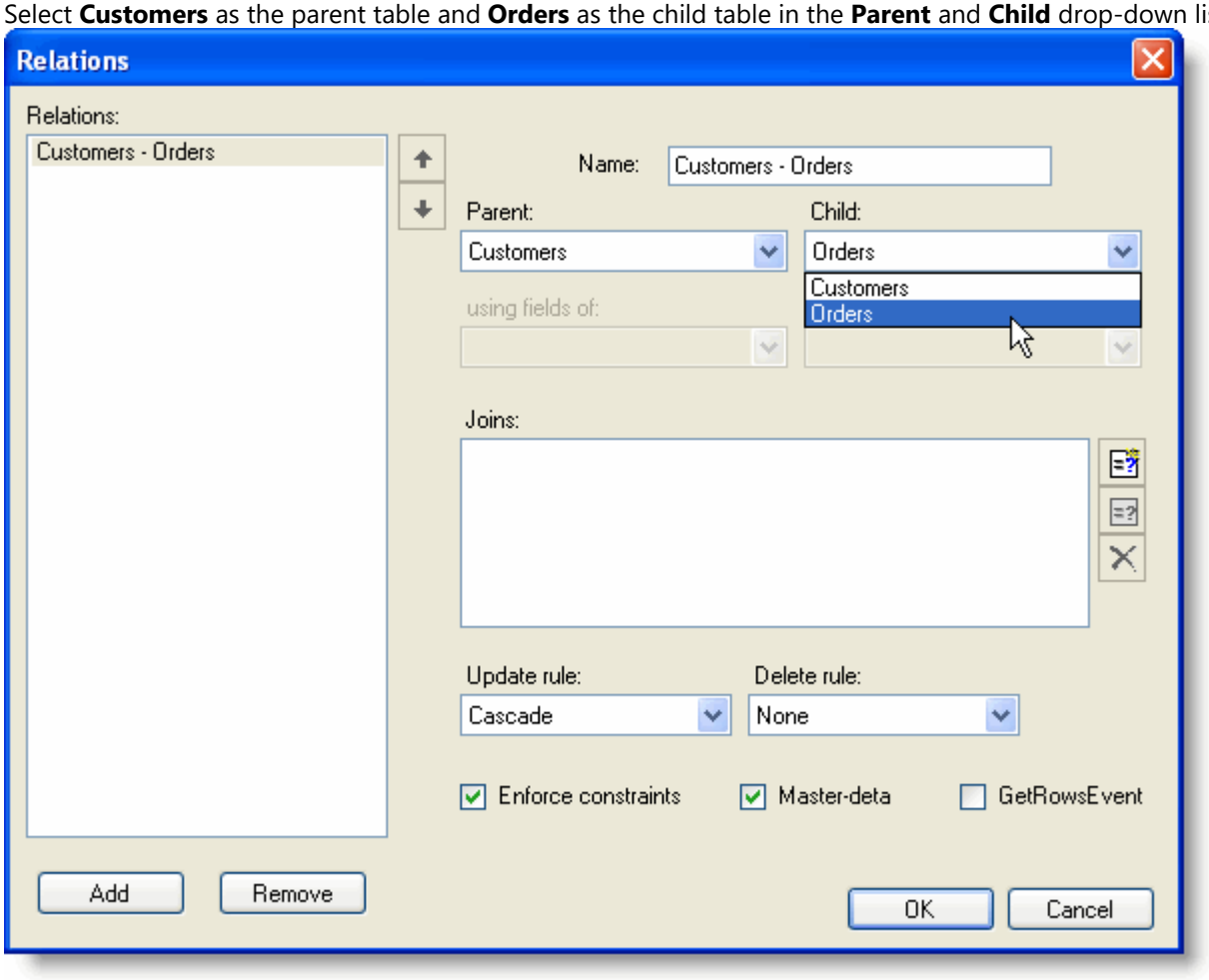
3. Click the drop-down arrow next to the ConnectionString property in the Properties window and select **<New Connection...>**. The **Add Connection** dialog box opens.
4. Select the provider, the database and other necessary connection properties in that dialog box. In this tutorial, we use the standard MS Access Northwind sample database (C1NWind.mdb).
    1. Click the **Change** button, if necessary, and select **Microsoft Access Database File**. The **.NET Framework Data Provider for OLE DB** is selected under **Data provider**.
    2. Under **Database file name**, browse to locate the **C1NWind.mdb**. The database is installed in the **Common** folder in the **ComponentOne Samples** directory.
    3. Click **OK** to close the **Add Connection** dialog box.
5. Select **C1ExpressTable1** and set the ConnectionComponent property to **C1ExpressConnection1** in the

Properties window.

6. Set C1ExpressTable1's DbTableName property to **Customers** and click **Yes** when asked to retrieve fields. **C1ExpressTable1** now represents the **Customers** database table.
7. Select **C1ExpressTable2** and set the ConnectionComponent property to **C1ExpressConnection1** in the **Properties** window.
8. Set C1ExpressTable2's DbTableName property to **Orders** and click **Yes** when asked to retrieve fields. **C1ExpressTable2** now represents the **Orders** database table.
9. Select **C1ExpressConnection1** and open its **Relations** editor by clicking **Edit Relations** under **C1ExpressConnection Tasks**.



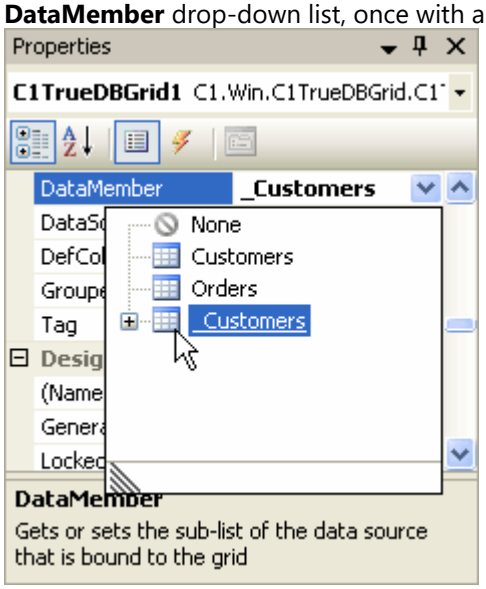
10. In the **Relations** editor, click the **Add** button to create a new relation.
11. Select **Customers** as the parent table and **Orders** as the child table in the **Parent** and **Child** drop-down lists.



12. Press the **Add join** button  to open the **Add new join** dialog box.
13. Select **CustomerID** in both the **Parent** and **Child** field drop-down lists and click **OK**.

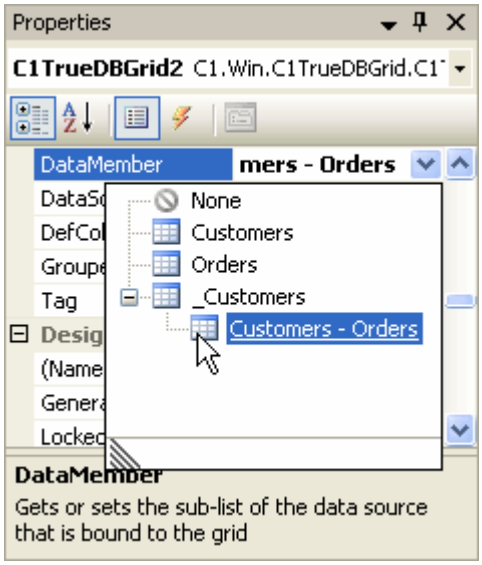
The following join connects the **Orders** table to its parent table **Customers**:
Customers.CustomerID = Orders.CustomerID

14. Click **OK** to close the **Relations** editor.
15. Select **C1TrueDBGrid1** and set its **DataSource** property to **C1ExpressConnection1** in the **Properties** window.
16. Set the **DataMember** property to **_Customers** to bind the first grid to the master (parent) table, **Customers**. Click **Yes** to replace the existing column layout. Note that the **Customers** table appears twice in the **DataMember** drop-down list, once with a leading underscore and once without it.

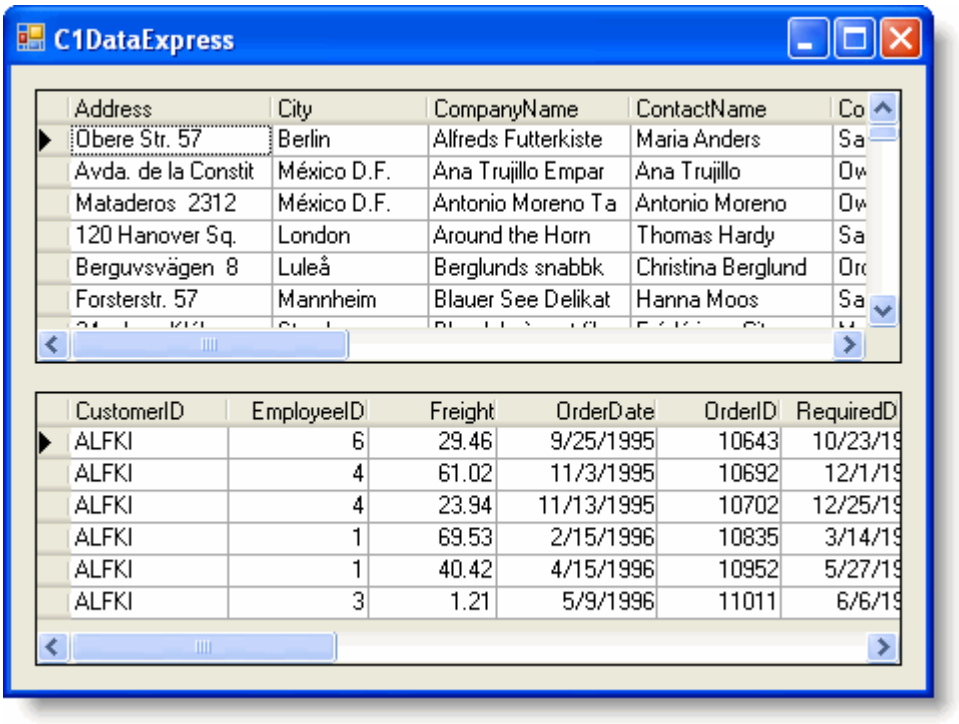


The instance without a leading underscore represents a standalone **Customers** table that is not a part of master-detail hierarchy. The instance with a leading underscore represents the same **Customers** table as the root of a master-detail hierarchy. It is also indicated by the presence of the + sign, expanding it you will see the *Customers – Orders* relation representing the child (detail) node of the hierarchy.

17. Select the **C1TrueDBGrid2**and set its **DataSource** property to **C1ExpressConection1** in the **Properties** window.
18. Set the **DataMember** property to **_Customers, Customer – Orders**. This binds the second grid to the detail (child) table, **Orders**. Click **Yes** to replace the existing column layout.



Run the program and observe the following:



The two grids show *Customers – Orders* data in a master-detail hierarchy. When you select a row in the **Customers** grid, the **Orders** grid is populated with order information for the selected customer.

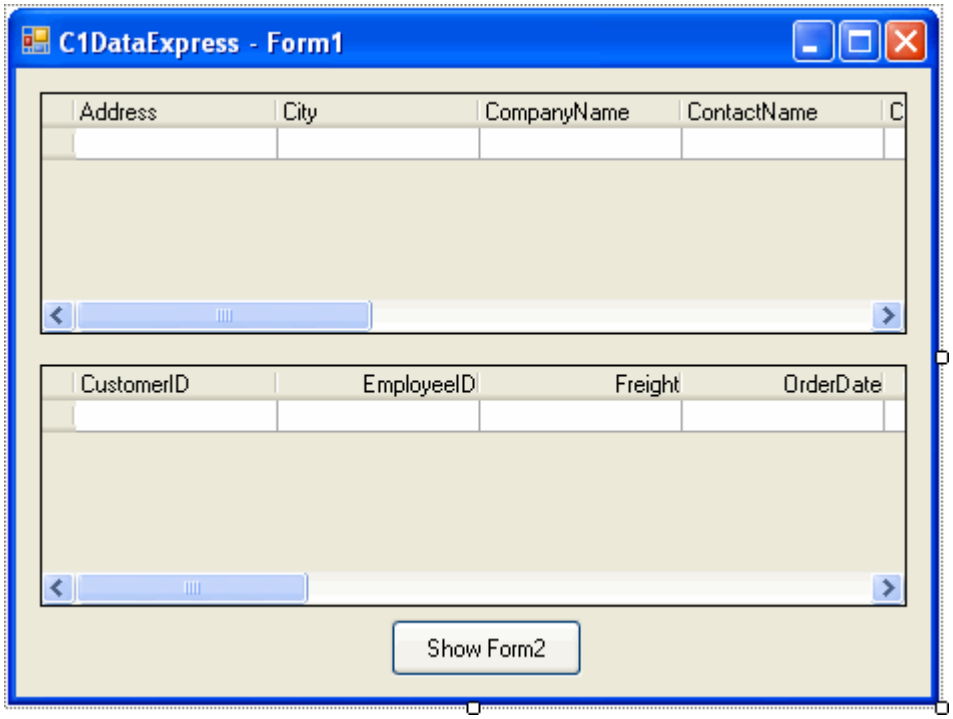
Save your project, note that Tutorial 4 uses the project created in this Tutorial.

Tutorial 4: Using C1ExpressView Component

In this tutorial, you will see how to use the C1ExpressView component to filter and sort table data and how to bind data-aware controls to table data that is defined in a different form. Complete the following steps:

1. Add a command button to **Form1** in the project built in Tutorial 3: C1ExpressConnection and Master-Detail

Relations.



2. Set the **Text** property of **Button1** to **Show Form2**.
3. Add the following code to the **Button1_Click** event:

To write code in Visual Basic

```
Visual Basic
Dim form As New Form2()
form.ShowDialog()
```

To write code in C#

```
C#
Form2 form = new Form2();
form.ShowDialog();
```

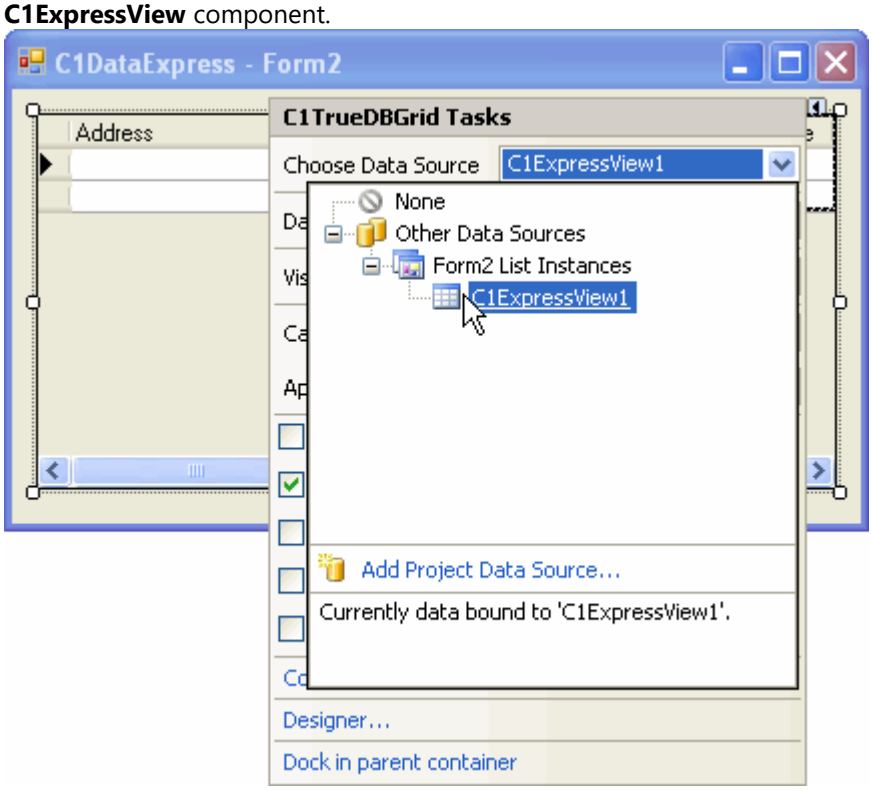
4. Add a new form, Form2 to the project and place the following components on the form:

Number of Components	Name	Namespace
1 C1ExpressView	C1ExpressView1	C1.Data.Express.C1ExpressView
1 C1TrueDBGrid	C1TrueDBGrid1	C1.Win.C1TrueDBGrid.C1TrueDBGrid

5. Select the **C1ExpressView1** component, and in the Properties window click the drop-down arrow next to the ExpressTableName property.
6. Make sure that **Form1** is open in the Visual Studio design time environment and select **Form1.C1ExpressTable1**. The ExpressTableName property shows the list of **C1ExpressTable** controls in the forms that are currently open.
7. Enter the following values for **C1ExpressView1** properties:

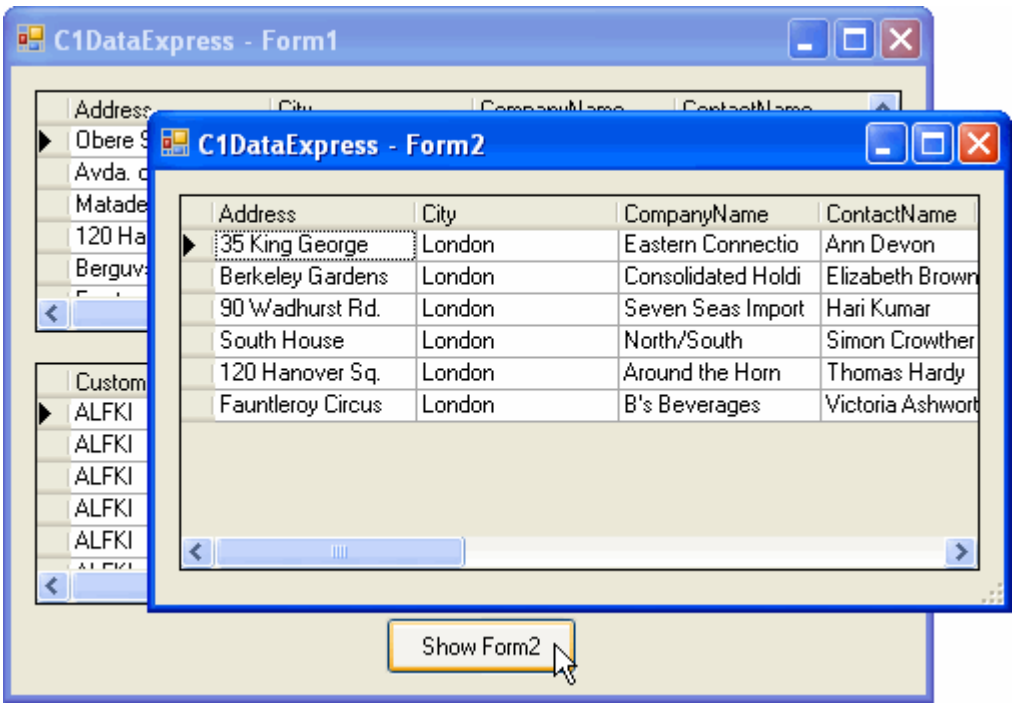
Property	Value
RowFilter	City = 'London'
Sort	ContactName

8. Select the **C1TrueDBGrid1** control and open the **C1TrueDBGrid Tasks** menu.
9. Click the **Choose Data Source** drop-down arrow and select **C1ExpressView1**. This binds the grid to the **C1ExpressView** component.



Run the program and observe the following:

When you press the **Show Form2** button, the form appears with the grid filled with filtered and sorted data from the **Customers** table: it includes only customers from **London** and is sorted by **ContactName**.



Notice that setting Sort is only one (programmatic) of the two possible ways to sort table data; the end user can also sort it interactively in the grid clicking on a column header.

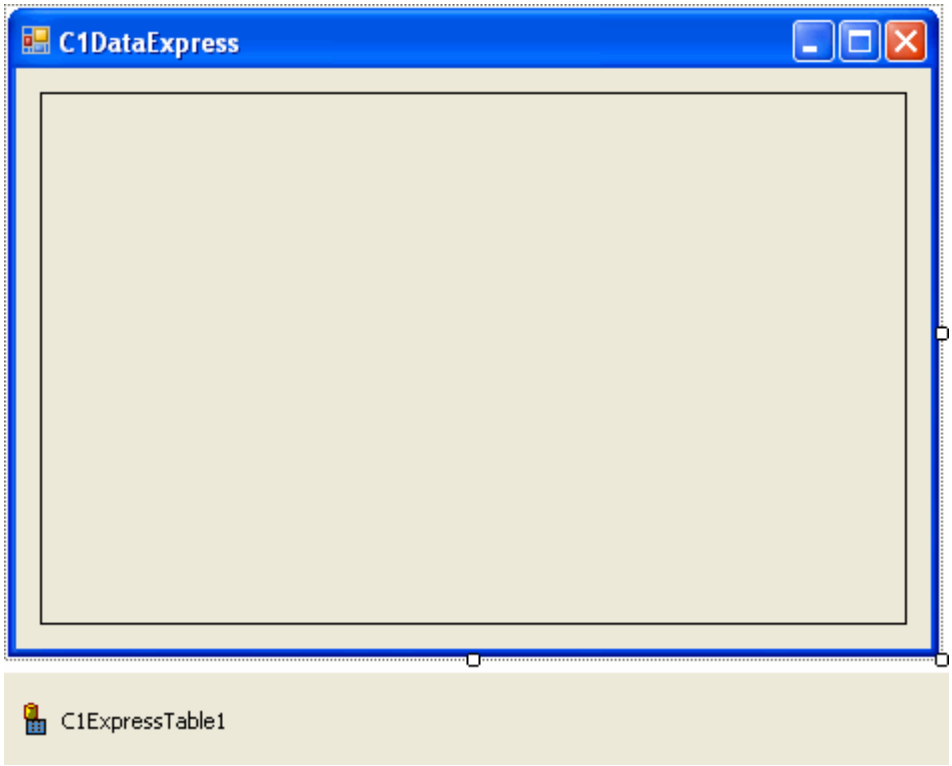
Tutorial 5: Customizing Data Behavior with Events

In this tutorial, you will learn how to use C1ExpressTable events to customize data behavior in code.

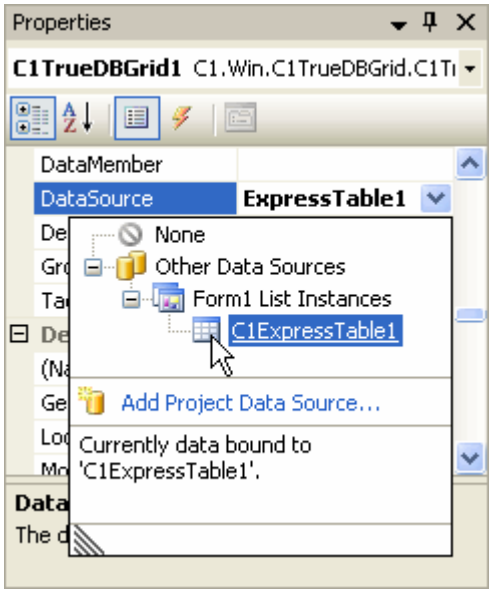
Create a new .NET 2.0 Windows Application project.

1. Place the following components on the form as shown in the image below:

Number of Components	Name	Namespace
1 C1ExpressTable	C1ExpressTable1	C1.Data.Express.C1ExpressTable
1 C1TrueDBGrid	C1TrueDBGrid1	C1.Win.C1TrueDBGrid.C1TrueDBGrid



2. Select the **C1ExpressTable1** component.
3. Click the drop-down arrow next to the ConnectionString property in the Properties window and select **<New Connection...>**. The **Add Connection** dialog box opens.
4. Select the provider, the database and other necessary connection properties in that dialog box. In this tutorial, we use the standard MS Access Northwind sample database (C1NWind.mdb).
    1. Click the **Change** button, if necessary, and select **Microsoft Access Database File**. The **.NET Framework Data Provider for OLE DB** is selected under **Data provider**.
    2. Under **Database file name**, browse to locate the **C1NWind.mdb**. The database is installed in the **Common** folder of the **ComponentOne Samples** directory.
    3. Click **OK** to close the **Add Connection** dialog box.
5. Click the drop-down arrow next to the DbTableName property and select **Order Details** from the database table list. Click **Yes** to retrieve the fields.
6. Select the **C1TrueDBGrid1**and set its **DataSource** property to **C1ExpressTable1** in the **Properties** window.



7. Add the following event handlers, **C1ExpressTable1_BeforeFieldChange** and **C1ExpressTable1_BeforeEndEdit**, and their code for **C1ExpressTable1**:

To write code in Visual Basic

Visual Basic

```
Private Sub C1ExpressTable1_BeforeFieldChange _
(ByVal sender As Object, ByVal e As  C1.Data.FieldChangeEventArgs) Handles _
C1ExpressTable1.BeforeFieldChange
If e.Field.Name = "Quantity" Then
  If e.NewValue < 1 Then
          e.NewValue = 1
          End If
ElseIf e.Field.Name = "Discount" Then
  If e.NewValue < 0 Or e.NewValue > 1 Then
          Throw New ApplicationException("Discount must be between 0 and 1")
  End If
End If
End Sub

Private Sub C1ExpressTable1_BeforeEndEdit _
(ByVal sender As Object, ByVal e As C1.Data.RowChangeEventArgs) Handles _
C1ExpressTable1.BeforeEndEdit
If e.Row("Quantity") * e.Row("UnitPrice") > 100000 Then
  Throw New ApplicationException("Too expensive")
End If
End Sub
```

To write code in C#

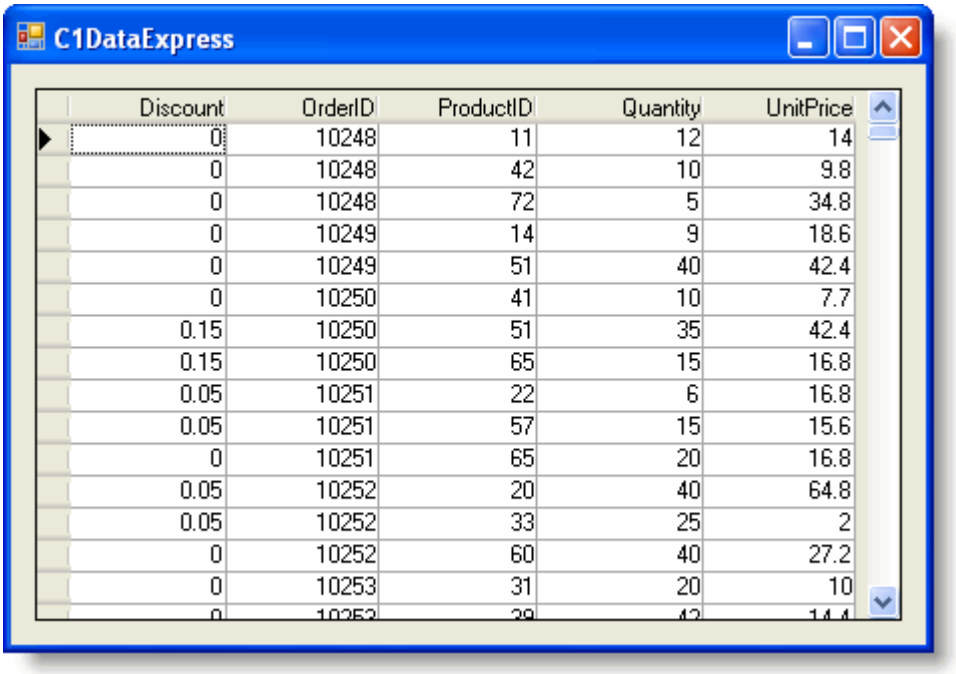
C#

```
private void c1ExpressTable1_BeforeFieldChange
(object sender, C1.Data.FieldChangeEventArgs e)
{
if (e.Field.Name == "Quantity")
```

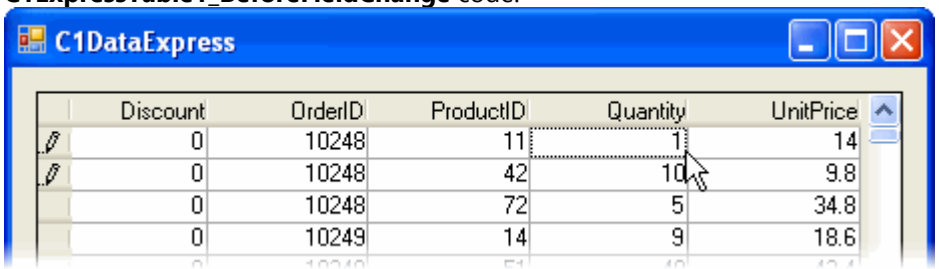
```
{
if ((short)e.NewValue < 1)
e.NewValue = (short)1;
}
else if (e.Field.Name == "Discount")
{
if ((float)e.NewValue < 0 || (float)e.NewValue > 1)
throw new ApplicationException("Discount must be between 0 and 1");
}
}

private void c1ExpressTable1_BeforeEndEdit
(object sender, C1.Data.RowChangeEventArgs e)
{
if ((short)e.Row["Quantity"] * (decimal)e.Row["UnitPrice"] > 100000)
throw new ApplicationException("Too expensive");
}
```

Run the program and observe the following:

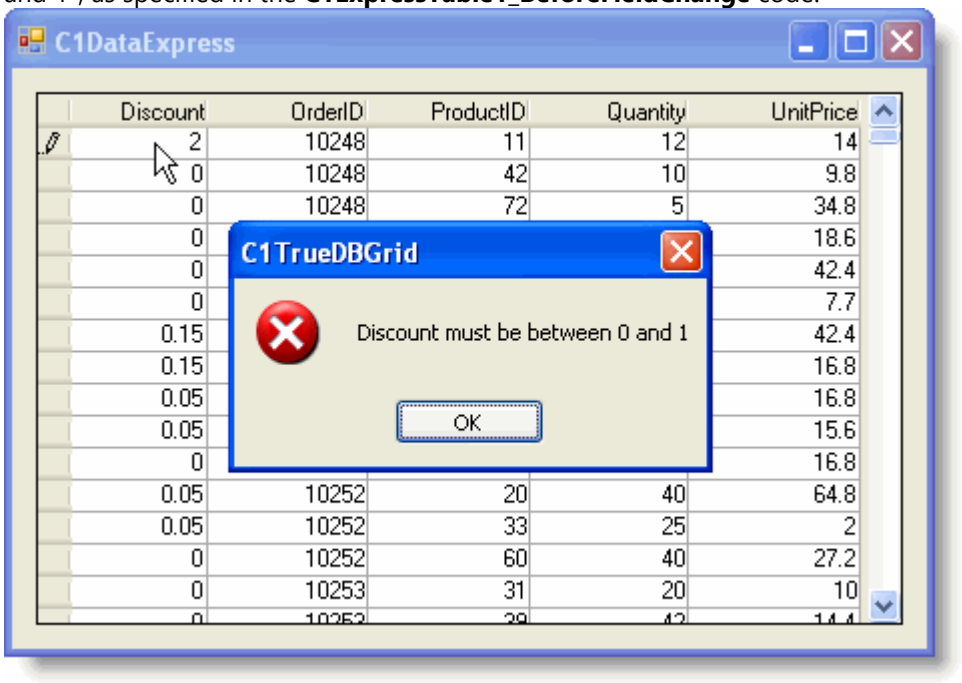


- Type **0** in *Quantity* column and press **Enter**. The value will be silently changed to 1, as specified in the **C1ExpressTable1_BeforeFieldChange** code.

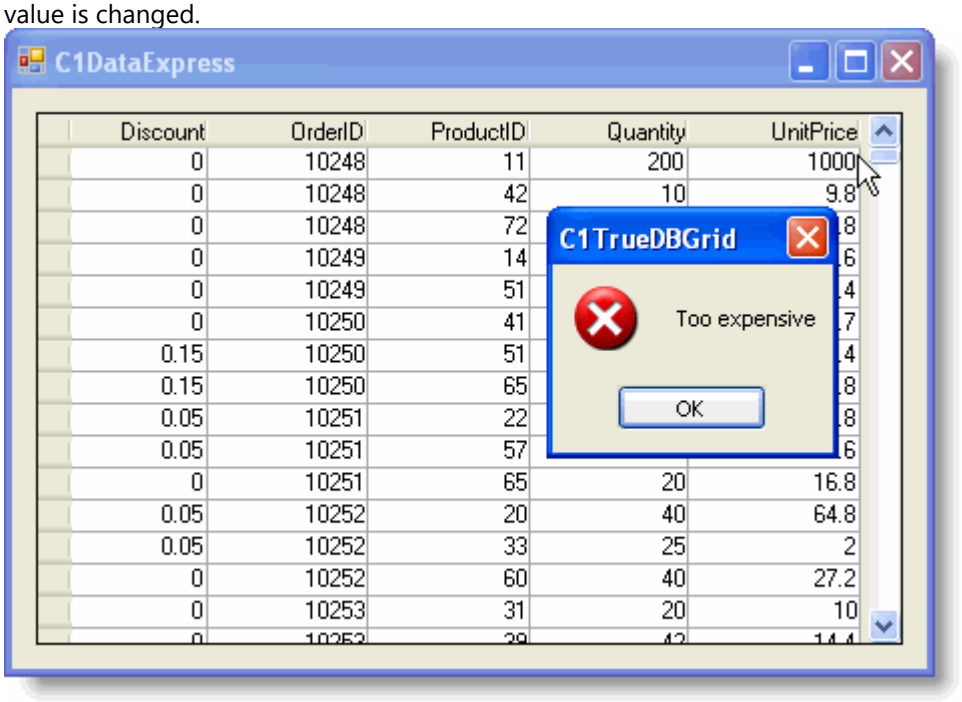


- Type **2** in *Discount* column and press **Enter**. This will result in an error message "Discount must be between 0

and 1", as specified in the **C1ExpressTable1_BeforeFieldChange** code.



- Type **200** in *Quantity* column and **1000** in *UnitPrice* column, then try to leave the row (for example, click another row in the grid). This results in an error message "Too expensive", as specified in the **C1ExpressTable1_BeforeEndEdit** code. **BeforeEndEdit** is not fired immediately when you change a field value; it is fired when you finish editing a whole row. This is why the error message appears when you leave the row as opposed to the error message in **BeforeFieldChange** above that is fired immediately before a field value is changed.



DataObjects for .NET 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.

The samples illustrate various features of **DataObjects for .NET**. However, the samples do not cover all **DataObjects for .NET** features, only those that are not sufficiently represented in **DataObjects for .NET** Tutorials.

All samples use the standard Microsoft Access sample database Northwind, **C1NWind.mdb**.

> **Note:** Please be aware that the sample projects have the database location (to the installation directory) hard coded in the connection string.

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\WinForms

The following table lists available Visual Basic and C# samples:

Sample	Description
ADOStorage	Enables you to work with the same data using **DataObjects for .NET** and ADO.NET, interchangeably. This functionality is very important in **DataObjects for .NET**, because it allows you to: • Do everything with your data that you can do with ADO.NET, including things that are not supported in **DataObjects for .NET** (for example, DataSet.Merge). • Export/import your data from/to **DataObjects for .NET** in ADO.NET DataSet format. This sample uses the C1DataSet and C1SchemaDef controls.
ADOStorage(DX)	Demonstrates data interchange between **DataObjects for .NET Express** and ADO.NET. This sample uses the C1ExpressTable and C1ExpressConnection controls.
AutoincrementMasterDetail	Demonstrates **Field.AutoIncrement=ClientAndServer** in a master-detail situation. This sample uses the C1DataSet and C1SchemaDef controls.
Calculations	Shows how to use **DataObjects for .NET** expression in calculation fields. This sample uses the C1DataSet and C1DataTableSource controls.
Constraints	Shows how to use **DataObjects for .NET** expression constraints. This sample uses the C1DataSet and **C1TrueDBGrid** controls.
CrystalReportsIntegration	Shows how to create a report using **DataObjects for .NET** data with Crystal Reports. Other reporting tools can be used in a similar fashion. This sample uses the C1DataSet and C1SchemaDef controls.
CrystalReportsIntegration(DX)	Shows how to create a report using **DataObjects for .NET Express** data with Crystal Reports. Other reporting tools can be used in a similar fashion. This sample uses the C1ExpressTable and C1ExpressConnection controls.

CustomDataProvider	Demonstrate a usage of **DataObjects for .NET** with an arbitrary .NET data provider. This sample uses the C1DataSet and C1SchemaDef controls.
CustomFillUpdate	This sample demonstrates: • Adding custom business methods to a data library. • Passing C1DataSet as a parameter between client and server, different processes/machines. • Some ways to customize Fill and Update process (about other Update customization options, see Updating the Database for more information). • Managing connections and transactions on the server, including using pre-created connections/transactions and using distributed COM+ transactions (spanning multiple databases). This sample uses the C1DataSet control.
CustomRelations	Shows how to create a custom relation, the one with behavior managed by a custom code. This sample uses the C1DataSet and C1SchemaDef and C1DataSetLogic controls.
CustomRelations(DX)	This sample shows how to create relations in code, specifying the list of child rows dynamically at run time. This sample uses the C1ExpressTable and C1ExpressConnection controls.
DataLibraryInheritance (C# only)	Shows how to inherit a data library from another data library. This sample uses the C1DataSet control.
DefaultDataView	Demonstrates the use of IsDefault property and GetRows event. This sample uses the C1DataSet and C1SchemaDef, and C1DataView controls.
DefaultDataView(DX)	This sample demonstrates the use of IsDefault property and GetRows event. This sample uses the C1ExpressTable, C1ExpressConnection, and C1ExpressView controls.
DynamicConnections	Demonstrates how to set ConnectionString and other Connection properties for a data set at run time. This sample uses the C1DataSet and C1SchemaDef controls.
DynamicExpress(DX)	This sample demonstrates how to set ConnectionString, DbTableName, and other properties at run time in **DataObjects for .NET Express**. This sample uses the C1ExpressTable and C1ExpressConnection controls.
Programmatic	This sample demonstrates creating **DataObjects for .NET** schema objects at run time. This sample uses the **C1TrueDBGrid** control.
Relations	Demonstrates master detail hierarchies based on various kinds of relations. This sample uses the C1DataSet and **C1TrueDBGrid** controls.
ResolvingConcurrencyConflicts	Shows how to resolve concurrency conflicts in database update. This sample uses the C1DataSet, C1DataView, and **C1TrueDBGrid** controls.
ResyncFromDatabase	This sample shows how to synchronize rows with the current state of the database. This sample uses the C1DataSet and C1SchemaDef controls.
SQLBasedTables	This sample shows how to specify custom SQL statements for filling and

	updating **DataObjects for .NET** tables. This sample uses the C1DataSet and **C1TrueDBGrid** controls.
SQLBasedTablesEasy	Demonstrates how to create a table based on a SQL SELECT statement or a stored procedure. This sample uses the C1DataSet C1SchemaDef, and C1TableLogic controls.
SQLBasedTablesEasy(DX)	Demonstrates how to create a table based on a SQL SELECT statement or a stored procedure. This sample uses the C1ExpressTable control.
UnboundTables	This sample shows various techniques that can be useful in creating unbound tables. This sample uses the C1DataSet and **C1TrueDBGrid** controls.
UsingDataView	This sample demonstrates filtering and sorting features of the C1DataView component. This sample uses the C1DataSet, C1SchemaDef, and C1DataView controls.
VirtualSort	This sample shows how to specify sort order in virtual mode table views and change it at run time. This sample uses the C1DataSet, C1SchemaDef, C1DataTableSource, and C1DataView controls.
WorkingWithData	Shows how to access **DataObjects for .NET** data programmatically. This sample uses the C1DataSet control.

DataObjects for .NET Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio, and know how to use bound controls in general. By following the topics outlined in the task-based help, you will be able to create projects demonstrating a variety of **DataObjects for .NET** features and get a good sense of what **DataObjects for .NET** can do.

Each task-based help topic also assumes that you have created a new .NET 2.0 project.

Avoiding a Memory Leak

It is possible you may experience a memory leak when using **DataObjects for .NET** and **True DBGrid for WinForms**. You may lose information, or the memory usage of your program may increase.

In order to avoid this problem, try adding the following code to your project:

To write code in Visual Basic

Visual Basic

```
Dim f As MyForm = New MyForm
Try
 f.ShowDialog
Finally
 CType(f, IDisposable).Dispose()
End Try
f = Nothing

OR

Using f As New MyForm
 f.ShowDialog
End Using
```

To write code in C#

C#

```
using ( MyForm f = new MyForm() );
{
    f.ShowDialog();
}

OR

MyForm f = new MyForm();
using ( f )
{
    f.ShowDialog();
}
f = null;
```

Tip: To avoid memory leaks try to avoid unnecessary memory allocation – for example avoid recreating an

> editing child form every time a user edits a row.

Changing the Connection String

In code, clear the **DataSource** property and reassign the **ConnectionString** property. In this example, we will change the connection string to the *Northwind* sample database and bind **DataGrid** to a C1ExpressTable control. Add the following code to your project (changing the connection string to add the full directory where the Nwind database is installed):

To write code in Visual Basic

Visual Basic

```
DataGrid1.DataSource = Nothing

C1ExpressTable1.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
C1NWind.mdb"

C1ExpressTable1.DbTableName = "Customers"
C1ExpressTable1.ExpressConnection.Fill()
DataGrid1.DataSource = C1ExpressTable1
```

To write code in C#

C#

```
DataGrid1.DataSource = null;

C1ExpressTable1.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
C1NWind.mdb";

C1ExpressTable1.DbTableName = "Customers";
C1ExpressTable1.ExpressConnection.Fill();
DataGrid1.DataSource = C1ExpressTable1;
```

Creating a Composite Table Programmatically

The following code is an example of how to create a composite table at run time in code. See Composite Tables and Using Composite Tables for additional information. Note that you might need change to the connection string to add the full directory where the Nwind database is installed.

To write code in Visual Basic

Visual Basic

```
'Connection string
    Private connectionString As String = "Provider=Microsoft.Jet.OLEDB.4.0;" + _
     "Data Source=C:\Users\<User Name>\Documents\ComponentOne
Samples\Common\C1NWind.mdb;" + _
     "Persist Security Info=False"

Private Sub btnFillCategoriesProducts_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnFillCategoriesProducts.Click
```

```
        Dim schemaDef As C1.Data.C1SchemaDef
        Dim connection As C1.Data.SchemaObjects.C1OleDbConnection
        Dim dbTableCategories As C1.Data.SchemaObjects.DbTable
        Dim dbTableProducts As C1.Data.SchemaObjects.DbTable
        Dim simpleRelation As C1.Data.SchemaObjects.SimpleRelation
        Dim joinCondition As C1.Data.SchemaObjects.JoinCondition
        Dim compositeTable As C1.Data.SchemaObjects.CompositeTable
        Dim tableViewCategories As C1.Data.SchemaObjects.TableView
        Dim tableViewProducts As C1.Data.SchemaObjects.TableView
        Dim viewRelation As C1.Data.SchemaObjects.ViewRelation
        Dim dataSetDef As C1.Data.SchemaObjects.DataSetDef
        Dim tableView As C1.Data.SchemaObjects.TableView
        Dim dataSet As C1.Data.C1DataSet

        'Create Schema (C1SchemaDef) object
        schemaDef = New C1.Data.C1SchemaDef()

        'Create a connection
        connection = New C1.Data.SchemaObjects.C1OleDbConnection(schemaDef.Schema)
        connection.ConnectionString = connectionString
        connection.Name = "Connection"

        'Create simple table Categories
        dbTableCategories = New C1.Data.SchemaObjects.DbTable(schemaDef.Schema)
        dbTableCategories.DbTableName = "Categories"
        dbTableCategories.Name = dbTableCategories.DbTableName
        dbTableCategories.Connection = connection
        CreateCategoriesFields(dbTableCategories)

        'Create simple table Products
        dbTableProducts = New C1.Data.SchemaObjects.DbTable(schemaDef.Schema)
        dbTableProducts.DbTableName = "Products"
        dbTableProducts.Name = dbTableProducts.DbTableName
        dbTableProducts.Connection = connection
        CreateProductsFields(dbTableProducts)

        'Establish a simple relation
        simpleRelation = New C1.Data.SchemaObjects.SimpleRelation(dbTableCategories,
dbTableProducts)
        simpleRelation.Name = "Categories - Products"
        joinCondition = New C1.Data.SchemaObjects.JoinCondition(simpleRelation.Joins)
        joinCondition.ParentField = dbTableCategories.Fields("CategoryID")
        joinCondition.ChildField = dbTableProducts.Fields("CategoryID")

        'Create a composite table
        compositeTable = New C1.Data.SchemaObjects.CompositeTable(schemaDef.Schema)
        compositeTable.Name = "CategoriesProducts"

        'Create TableView for Categories table
        tableViewCategories = New
C1.Data.SchemaObjects.CompositeDefView(compositeTable.CompositeTableDef,
```

```
dbTableCategories)
        tableViewCategories.Name = "Categories"
        tableViewCategories.RetrieveFields()

        'Create TableView for Products table
        tableViewProducts = New
C1.Data.SchemaObjects.CompositeDefView(compositeTable.CompositeTableDef,
dbTableProducts)
        tableViewProducts.Name = "Products"
        tableViewProducts.RetrieveFields()

        'Create ViewRelation between the TableViews
        viewRelation = New
C1.Data.SchemaObjects.CompositeDefRelation(compositeTable.CompositeTableDef,
simpleRelation)
        viewRelation.Name = simpleRelation.Name
        viewRelation.Parent = tableViewProducts
        viewRelation.Child = tableViewCategories

        'Create fields in the composite table (ProductID field is already here)
        AddCompositeField("ProductID", compositeTable, tableViewProducts,
dbTableProducts)
        AddCompositeField("CategoryID", compositeTable, tableViewProducts,
dbTableProducts)
        AddCompositeField("ProductName", compositeTable, tableViewProducts,
dbTableProducts)
        AddCompositeField("CategoryName", compositeTable, tableViewCategories,
dbTableCategories)
        AddCompositeField("Discontinued", compositeTable, tableViewProducts,
dbTableProducts)
        AddCompositeField("UnitPrice", compositeTable, tableViewProducts,
dbTableProducts)
        AddCompositeField("QuantityPerUnit", compositeTable, tableViewProducts,
dbTableProducts)

        'Create DataSetDef object
        dataSetDef = New C1.Data.SchemaObjects.DataSetDef(schemaDef.Schema)
        dataSetDef.Name = "DataSet"

        'Create TableView for the whole composite table
        tableView = New C1.Data.SchemaObjects.TableView(dataSetDef, compositeTable)
        tableView.Name = "CategoriesProducts"
        tableView.RetrieveFields()

        'Create C1DataSet to bind to
        dataSet = New C1.Data.C1DataSet()
        dataSet.SchemaDef = schemaDef
        dataSet.DataSetDef = "DataSet"
        dataSet.Fill(True)

        'Unbind the grid from current data source
```

```
        gridBound.DataMember = ""
        gridBound.DataSource = Nothing
        gridBound.ClearFields()

        'Bind to CategoriesProducts composite table
        gridBound.DataMember = "CategoriesProducts"
        gridBound.DataSource = dataSet

        'Adjust columns width and format
        SetColumnsWidth(gridBound, New Int32() {60, 63, 155, 85, 73, 62, 110})
        gridBound.Columns("UnitPrice").NumberFormat = "Currency"
    End Sub
```

To write code in C#

C#

```
// Connection string
private readonly string connectionString = @"Provider=Microsoft.Jet.OLEDB.4.0;"+
@"Data Source=C1NWind.mdb;" + @"Persist Security Info=False";

private void btnFillCategoriesProducts_Click(object sender, System.EventArgs e)
{
    C1SchemaDef schemaDef;
    C1OleDbConnection connection;
    DbTable dbTableCategories;
    DbTable dbTableProducts;
    SimpleRelation simpleRelation;
    JoinCondition joinCondition;
    CompositeTable compositeTable;
    TableView tableViewCategories;
    TableView tableViewProducts;
    ViewRelation viewRelation;
    DataSetDef dataSetDef;
    TableView tableView;
    C1DataSet dataSet;

    // Create Schema (C1SchemaDef) object
    schemaDef = new C1SchemaDef();

    // Create a connection
    connection = new C1OleDbConnection(schemaDef.Schema);
    connection.ConnectionString = connectionString;
    connection.Name = "Connection";

    // Create simple table Categories
    dbTableCategories = new DbTable(schemaDef.Schema);
    dbTableCategories.DbTableName = "Categories";
    dbTableCategories.Name = dbTableCategories.DbTableName;
    dbTableCategories.Connection = connection;
    CreateCategoriesFields(dbTableCategories);
```

```
    // Create simple table Products
    dbTableProducts = new DbTable(schemaDef.Schema);
    dbTableProducts.DbTableName = "Products";
    dbTableProducts.Name = dbTableProducts.DbTableName;
    dbTableProducts.Connection = connection;
    CreateProductsFields(dbTableProducts);

    // Establish a simple relation
    simpleRelation = new SimpleRelation(dbTableCategories, dbTableProducts);
    simpleRelation.Name = "Categories - Products";
    joinCondition = new JoinCondition(simpleRelation.Joins);
    joinCondition.ParentField = dbTableCategories.Fields["CategoryID"];
    joinCondition.ChildField = dbTableProducts.Fields["CategoryID"];

    // Create a composite table
    compositeTable = new CompositeTable(schemaDef.Schema);
    compositeTable.Name = "CategoriesProducts";

    // Create TableView for Categories table
    tableViewCategories = new CompositeDefView(compositeTable.CompositeTableDef,
dbTableCategories);
    tableViewCategories.Name = "Categories";
    tableViewCategories.RetrieveFields();

    // Create TableView for Products table
    tableViewProducts = new CompositeDefView(compositeTable.CompositeTableDef,
dbTableProducts);
    tableViewProducts.Name = "Products";
    tableViewProducts.RetrieveFields();

    // Create ViewRelation between the TableViews
    viewRelation = new CompositeDefRelation(compositeTable.CompositeTableDef,
simpleRelation);
    viewRelation.Name = simpleRelation.Name;
    viewRelation.Parent = tableViewProducts;
    viewRelation.Child = tableViewCategories;

    // Create fields in the composite table (ProductID field is already here)
    AddCompositeField("ProductID", compositeTable, tableViewProducts,
dbTableProducts);
    AddCompositeField("CategoryID", compositeTable, tableViewProducts,
dbTableProducts);
    AddCompositeField("ProductName", compositeTable, tableViewProducts,
dbTableProducts);
    AddCompositeField("CategoryName", compositeTable, tableViewCategories,
dbTableCategories);
    AddCompositeField("Discontinued", compositeTable, tableViewProducts,
dbTableProducts);
    AddCompositeField("UnitPrice", compositeTable, tableViewProducts,
dbTableProducts);
    AddCompositeField("QuantityPerUnit", compositeTable, tableViewProducts,
```

```
dbTableProducts);

    // Create DataSetDef object
    dataSetDef = new DataSetDef(schemaDef.Schema);
    dataSetDef.Name = "DataSet";

    // Create TableView for the whole composite table
    tableView = new TableView(dataSetDef, compositeTable);
    tableView.Name = "CategoriesProducts";
    tableView.RetrieveFields();

    // Create C1DataSet to bind to
    dataSet = new C1DataSet();
    dataSet.SchemaDef = schemaDef;
    dataSet.DataSetDef = "DataSet";
    dataSet.Fill(true);

    // Unbind the grid from current data source
    gridBound.DataMember = "";
    gridBound.DataSource = null;
    gridBound.ClearFields();

    // Bind to CategoriesProducts composite table
    gridBound.DataMember = "CategoriesProducts";
    gridBound.DataSource = dataSet;

    // Adjust columns width and format
    SetColumnsWidth(gridBound, new int[] {60, 63, 155, 85, 73, 62, 110});
    gridBound.Columns["UnitPrice"].NumberFormat = "Currency";
}
```

Note: Sample Project Available For a complete sample detailing how to create a composite table programmatically, see the **Programmatic** sample available from the ComponentOne HelpCentral Sample page.

Exporting Data from a C1DataSet to XML

To export data from a C1DataSet to an .xml file, complete the following steps:

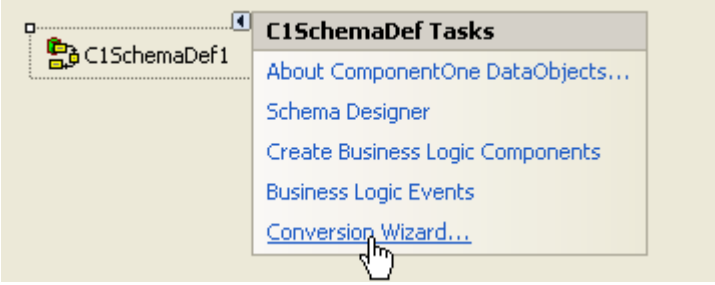
1. Select the C1DataSet component and use the smart tags to open the **C1DataSet Tasks** menu.
   

   Alternatively, you can right-click the C1DataSet component and select **Save Export XML Schema**.
2. Select **Save Export XML Schema**.
3. Enter a file name for the .xml file and click **Save**.

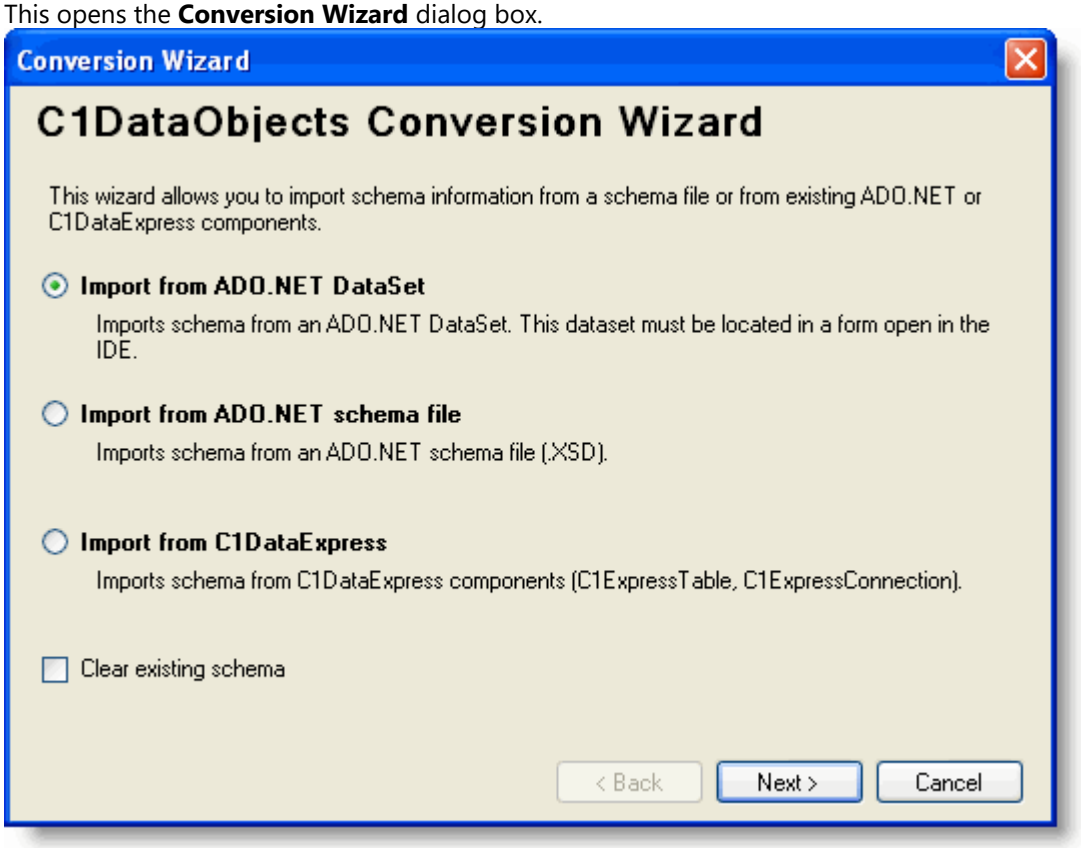
Importing Schema Information using the Conversion Wizard

This topic explains how to import schema information using the **C1DataObjects Conversion Wizard**. Complete the following steps:

1. Begin by opening the **C1SchemaDef Tasks** menu and selecting **Conversion Wizard**.



This opens the **Conversion Wizard** dialog box.



2. Select the source of the schema information to be imported. The following schema sources are available:

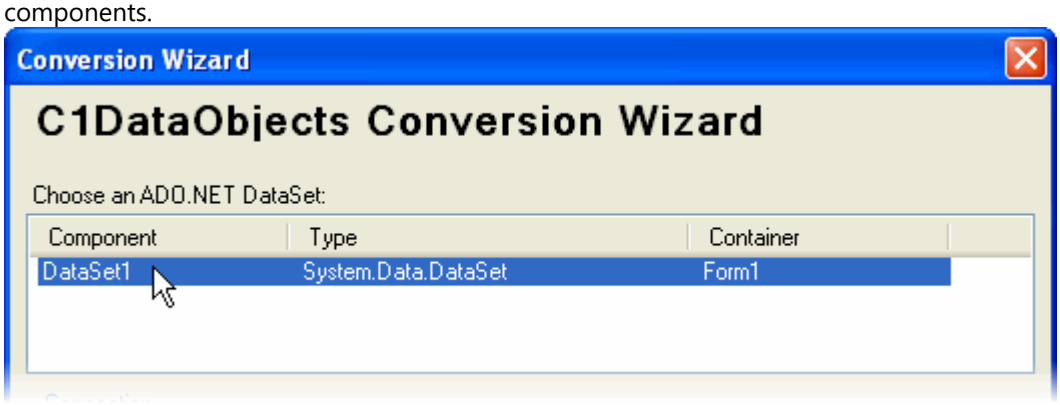
Source	Description
Import from ADO.NET DataSet	Imports schema from an ADO.NET DataSet. This dataset must be located in a form open in the IDE.
Import from ADO.NET schema file	Imports schema from an ADO.NET schema file (.XSD).
Import from DataObjects for .NET Express	Imports schema from **DataObjects for .NET Express** components (C1ExpressTable, C1ExpressConnection).

Check the **Clear existing schema** box if you want to delete the existing schema, and click **Next** to continue.

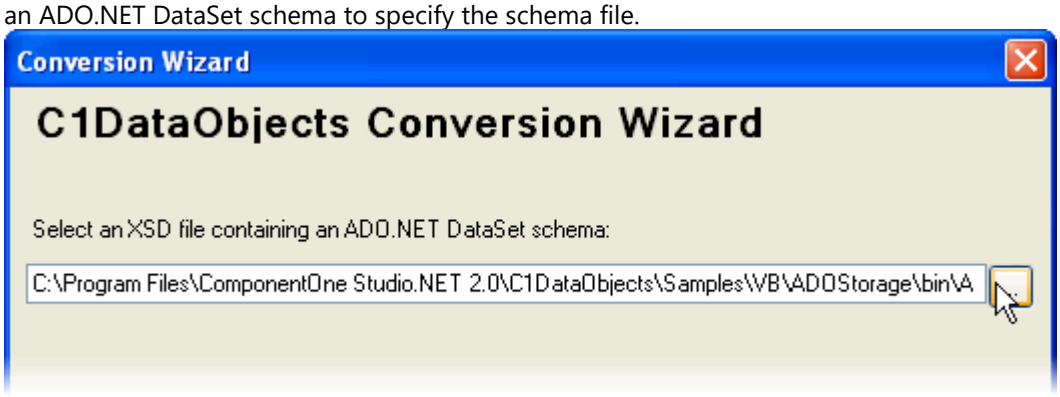
> **Note:** There must be a **Connection** object in the schema for bound tables.

3. Select the existing dataset, schema file or component with the schema information:

- If importing from an ADO.NET DataSet, choose an ADO.NET DataSet from the list of dataset components.

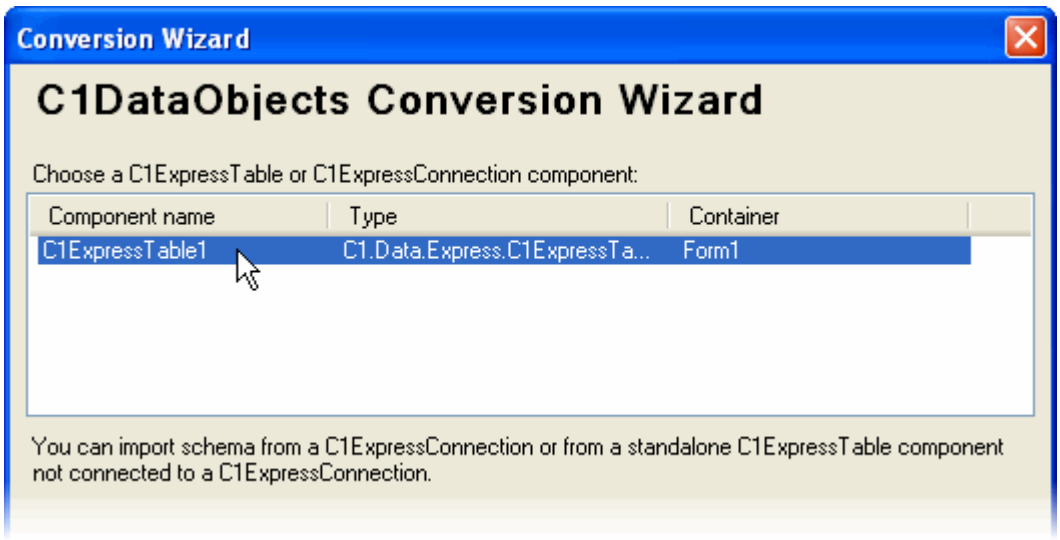


- If importing from an ADO.NET schema file, click the **ellipsis** button under Select an XSD file containing an ADO.NET DataSet schema to specify the schema file.



- If importing from **DataObjects for .NET Express** components, choose a C1ExpressConnection or C1ExpressTable component from the list of components.

> **Note:** You can import a schema from a C1ExpressConnection or from a standalone C1ExpressTable component not connected to a C1ExpressConnection.

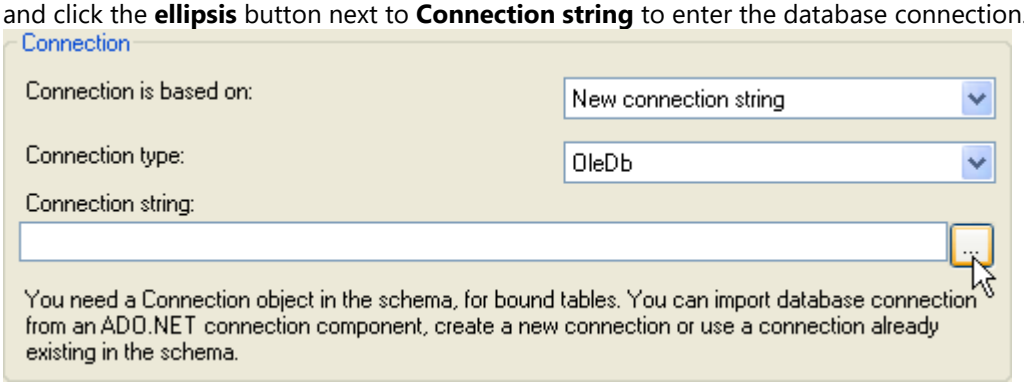


4. Define the connection and tables to be used.

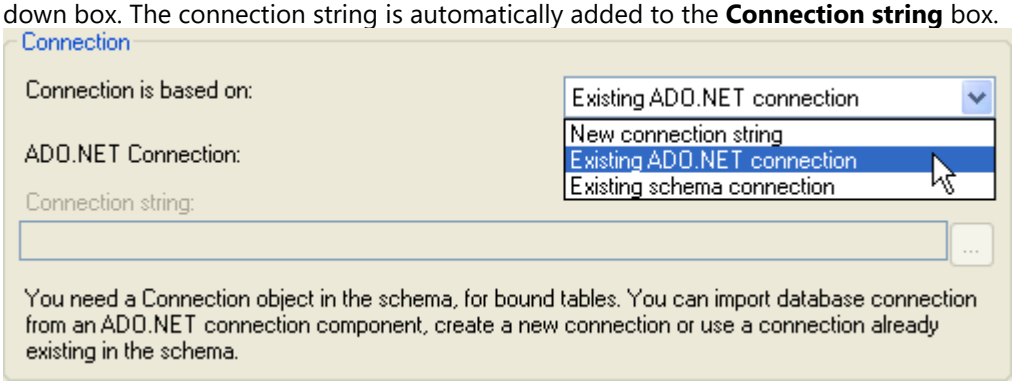
> **Note:** The **Connection** section of the **Conversion Wizard** is the same if you are importing from an ADO.NET DataSet or schema file. If importing from an **ADO.NET DataSet** or **schema file**, follow the steps below.

1. In the first drop-down box, specify whether the **Connection is based on**: a new connection string, an existing ADO.NET connection or an existing schema connection.

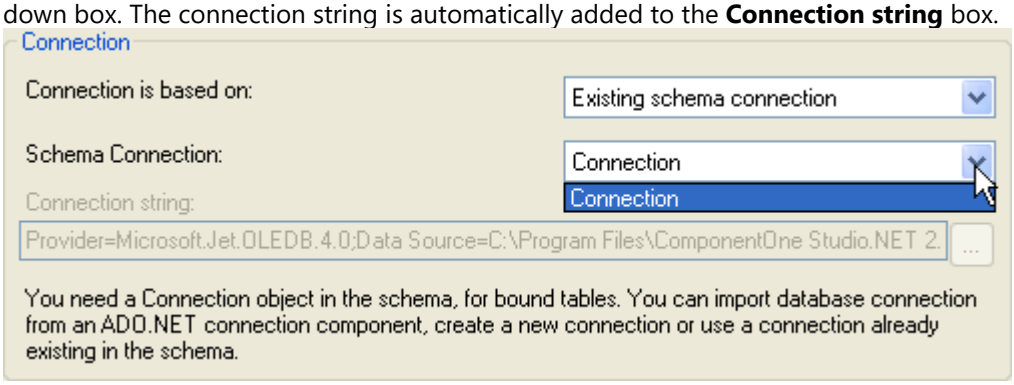
1. If you select **New connection string**, specify the **Connection type** in the next drop-down box and click the **ellipsis** button next to **Connection string** to enter the database connection.



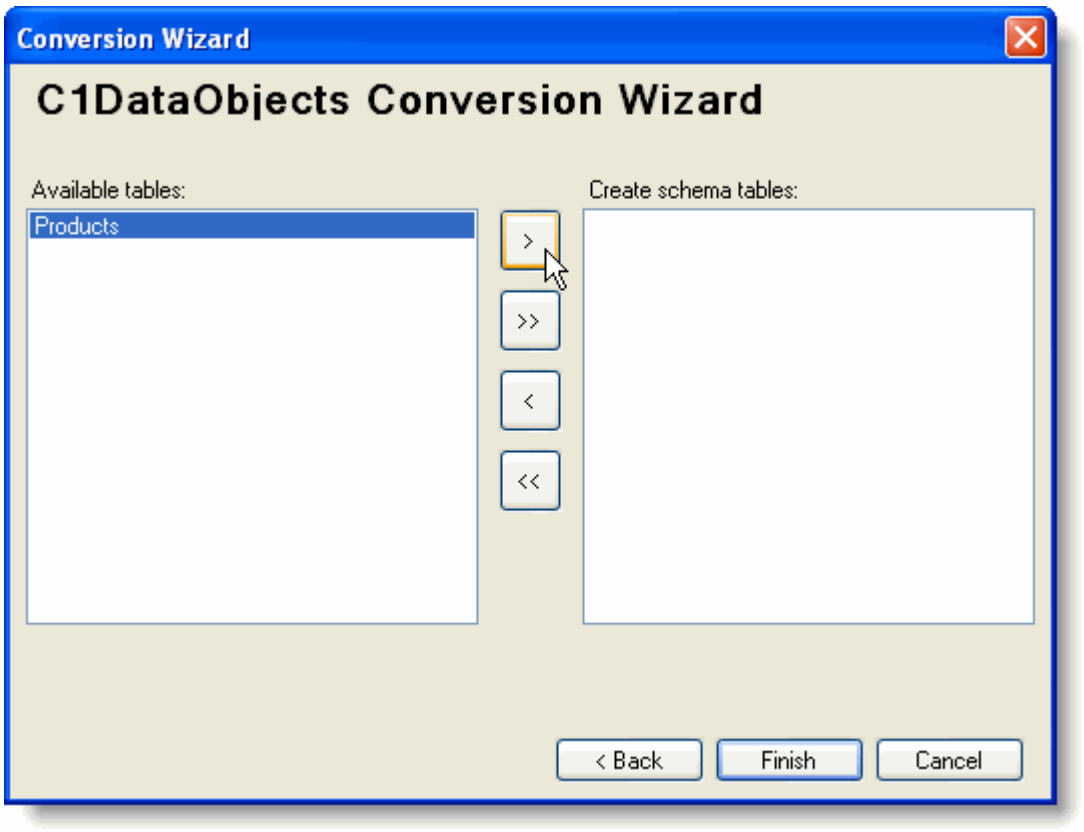
2. If you select **Existing ADO.NET connection**, specify the **ADO.NET Connection** in the next drop-down box. The connection string is automatically added to the **Connection string** box.



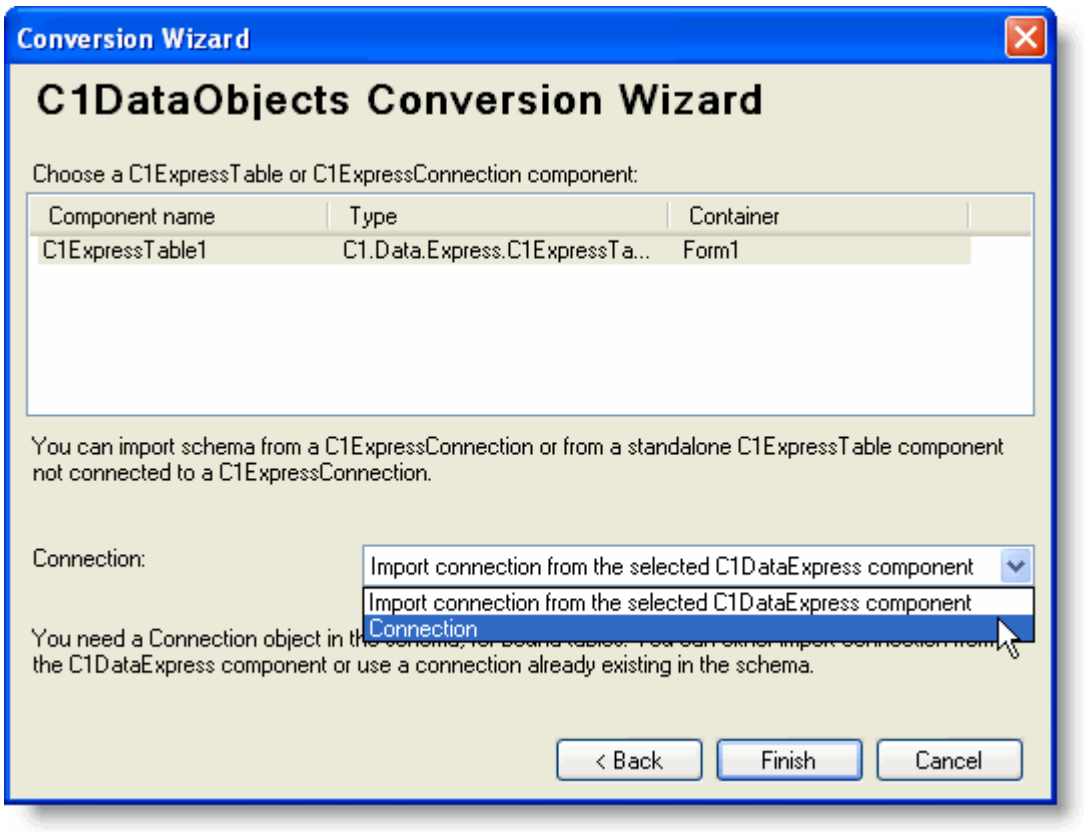
3. If you select **Existing schema connection**, select a **Schema Connection** from the next drop-down box. The connection string is automatically added to the **Connection string** box.



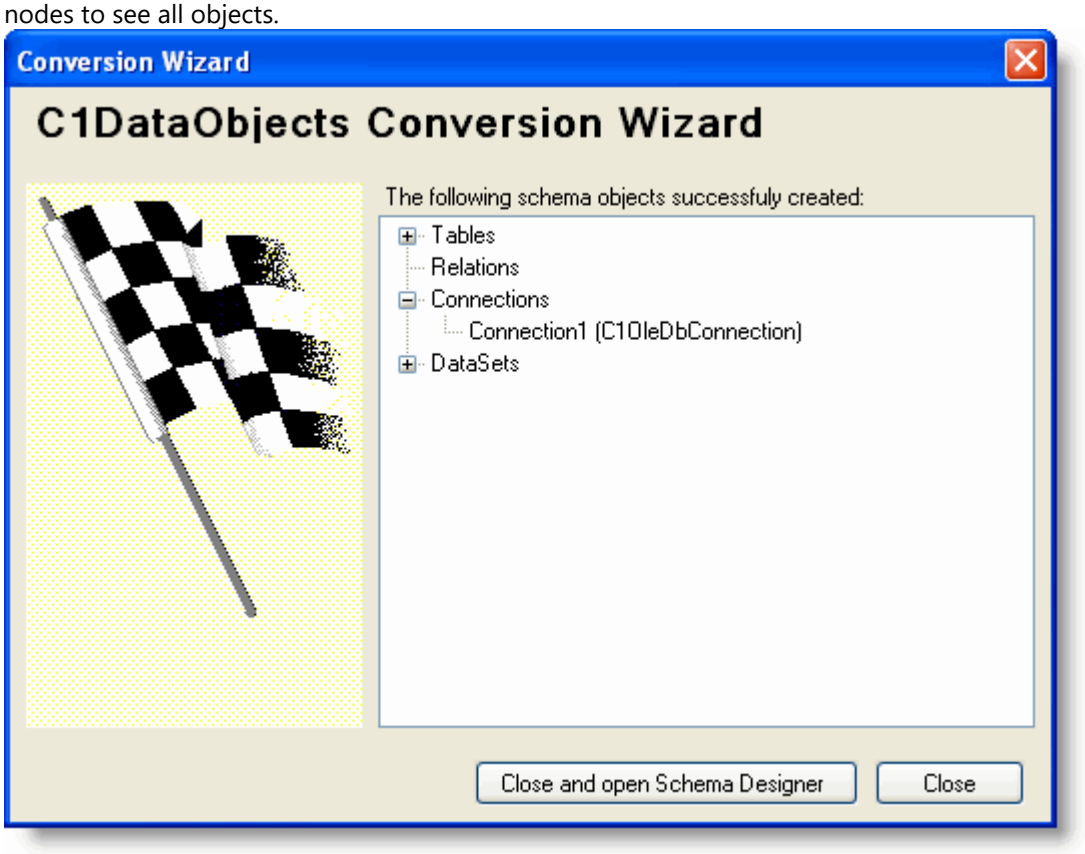
2. Click **Next** to continue.
3. Select tables from the list of **Available tables** in the database, and click the arrows to move the tables to the **Create schema tables** box.



Note: If you are importing from **DataObjects for .NET Express** components, select **Import connection from the selected C1DataExpress component** or choose one of the other existing connections from the **Connection** drop-down box.



5. Click **Finish**. The final window in the wizard shows the schema objects that have been created. Expand the nodes to see all objects.

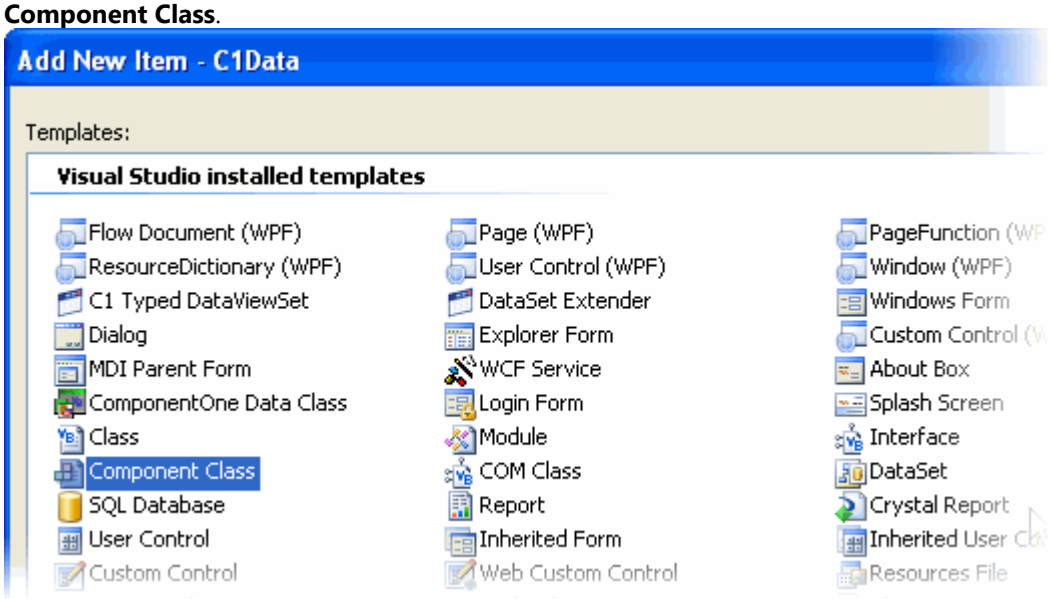


6. Click **Close and open Schema Designer** to complete the wizard.

Using the C1SchemaRef Control

To use the C1SchemaRef component to represent a C1SchemaDef component, complete the following steps:

1. Add a C1SchemaDef control to your form and create a data schema.
2. Add a new component class file to your project by selecting the **Project** menu and clicking **Add New Item**, **Component Class**.



3. Enter a name for the class in the **Name** text box and click **Add**.

4. Double-click the C1SchemaRef component in the Toolbox to add it to the design surface of the component class file.
5. In the Visual Studio Properties window, set the SchemaDef property of the **C1SchemaRef** to the C1SchemaDef you created in step 1.

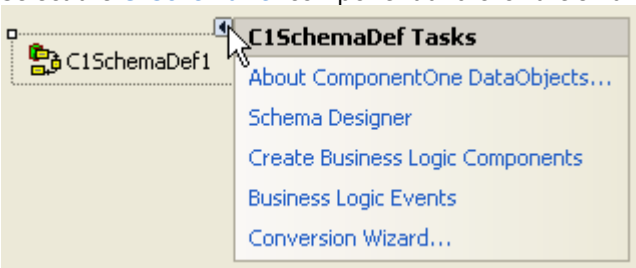
   > **Note:** When working with multiple files using C1SchemaRef, the file containing the C1SchemaDef component must be open.

6. Add any desired business logic components such as C1TableLogic and C1DataSetLogic to the component class file.
7. For each business logic component, set its SchemaComponent property to the C1SchemaRef component added in step 4.
8. If a C1TableLogic component was added, choose a table from the Table property drop-down box in the Visual Studio properties window. If a C1DataSetLogic component was added, select a data set from the DataSetDef property drop-down box.
9. In the initialization section of your code, call Add to add a C1SchemaRef component to the list of C1SchemaRef components associated with the C1SchemaDef component.

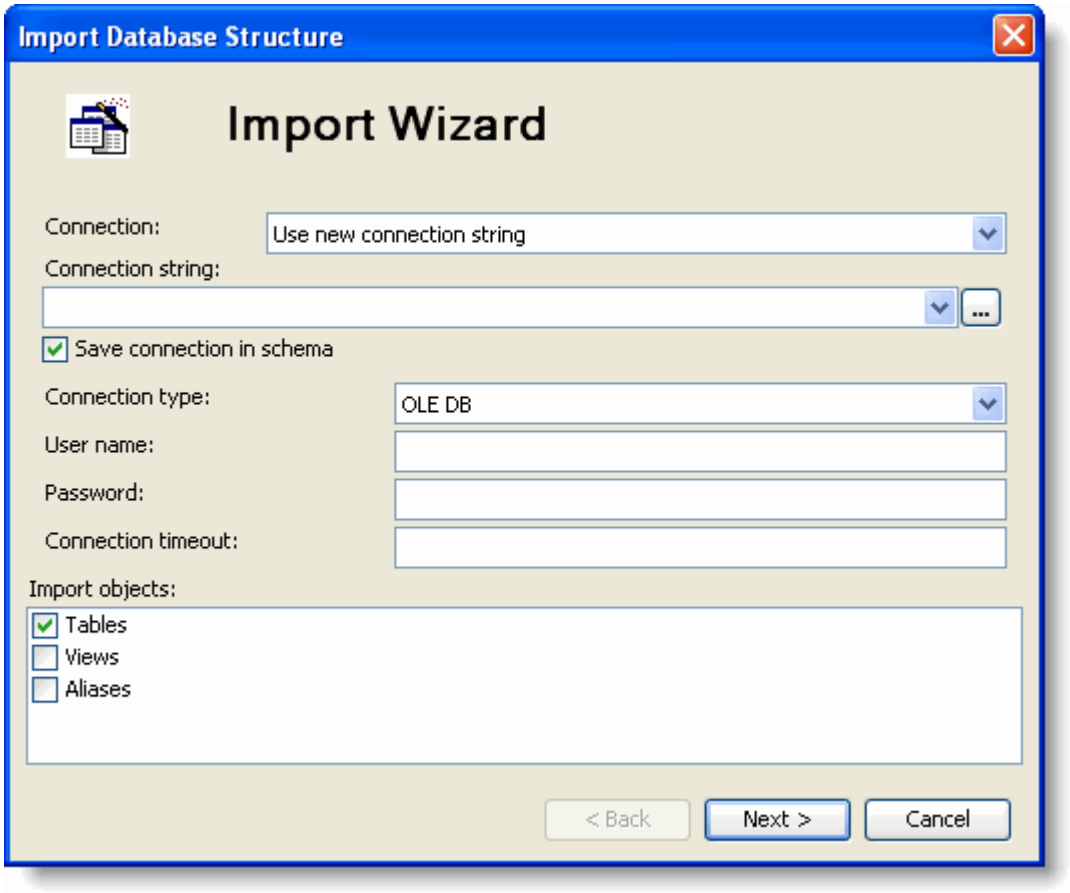
Creating a Schema with the Import Wizard

To use the **Import Wizard** to create a schema based on the structured information stored in a specified database, complete the following steps:

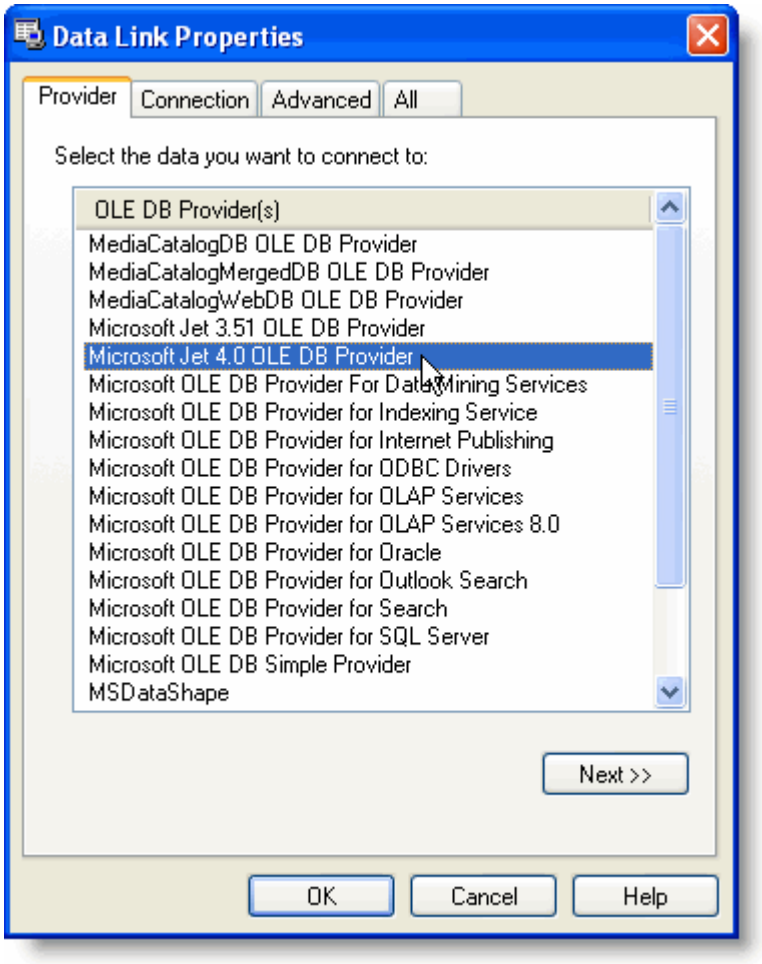
1. Add a C1SchemaDef control to your form.
2. Select the C1SchemaDef component and click the smart tag to open the **C1SchemaDef Tasks** menu.



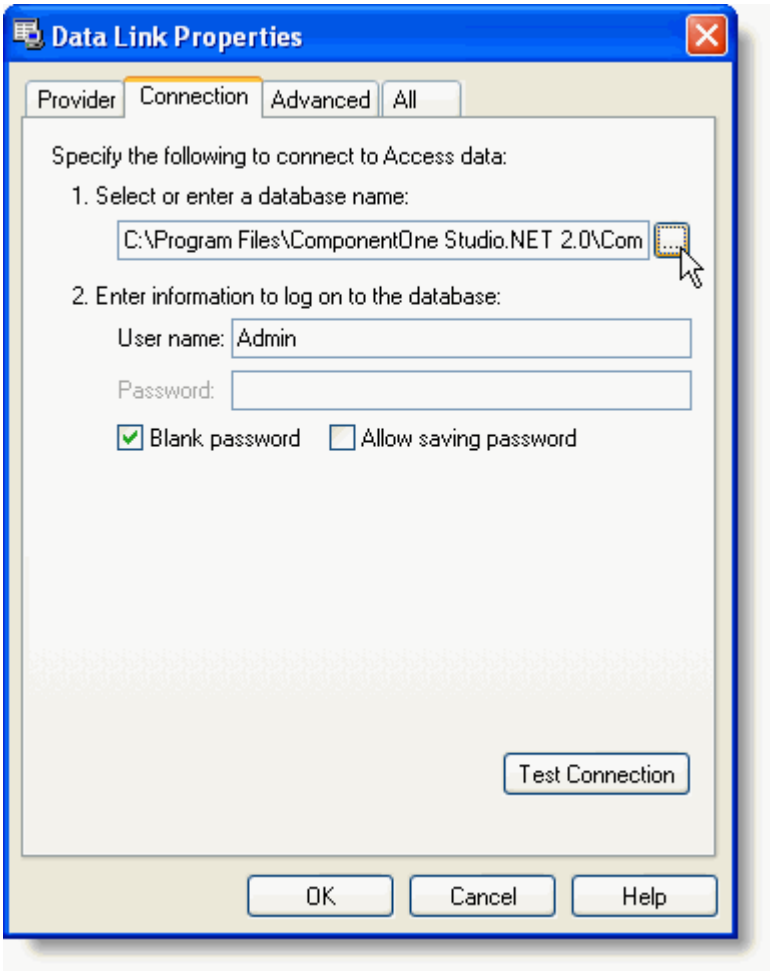
3. Select **Schema Designer**. The **Import Wizard** appears.



4. The**Import Wizard** guidesyou through the steps of connecting to a database. You can either use an existingconnection or create a new one.
    - To use an existing database connection, select it from the **Connection string** drop-down list.
    - To create a new connection:
    1. Select "Use new connection string" from the **Connection** drop-down list and check the "Save connection in schema" check box.
    2. Click the **ellipsis** button under**Connection string**. The standard OLE DB **Data Link Properties** window appears.
    3. Click the **Provider** tab and select the data to connect to.



4. Click the **Connection** tab and select a database. Enter a user name and password, if necessary.



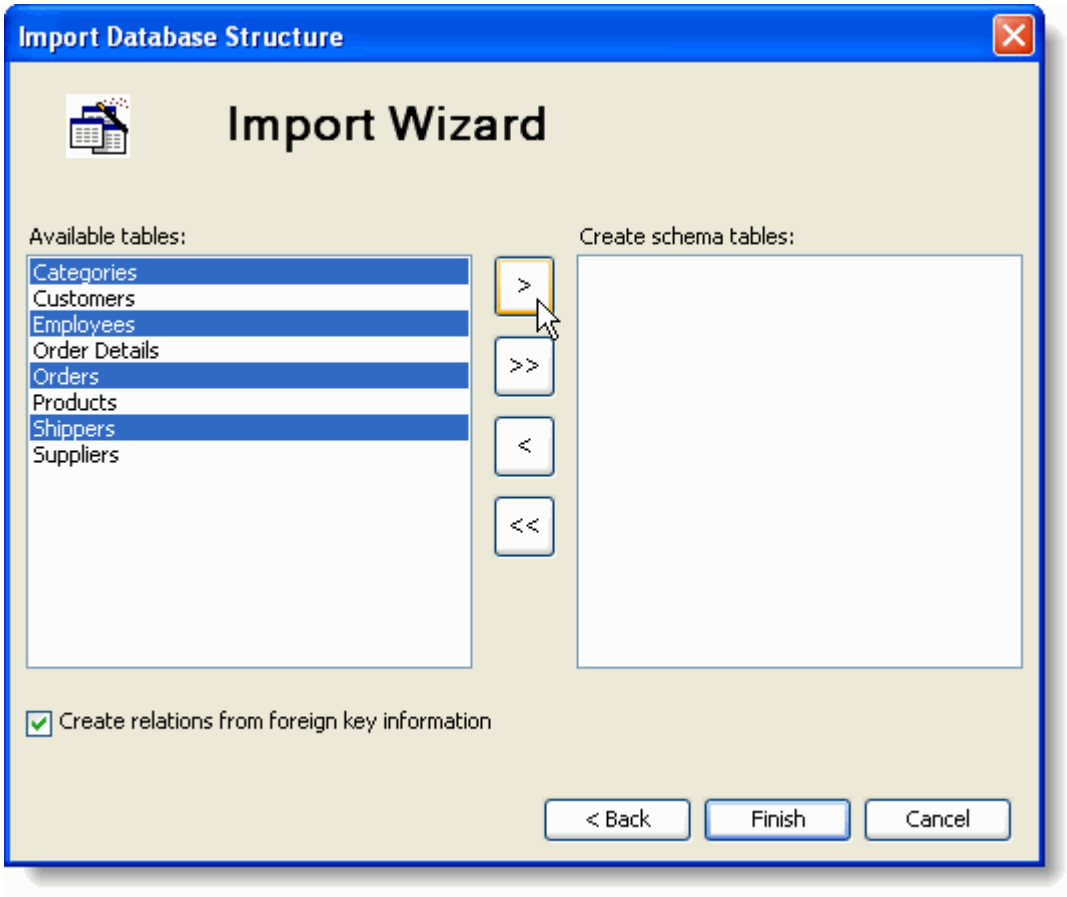
5. Click the **Test Connection** button to make sure that you have successfully connected to the database or server and click **OK** to return to the **Import Wizard**. The new string appears in the **ConnectionString** property box.

> **Note:** Using OLE DB is only one option. **DataObjects for .NET** supports other .NET data providers as well, see Database Connections. The information you must specify depends on the Provider you choose.

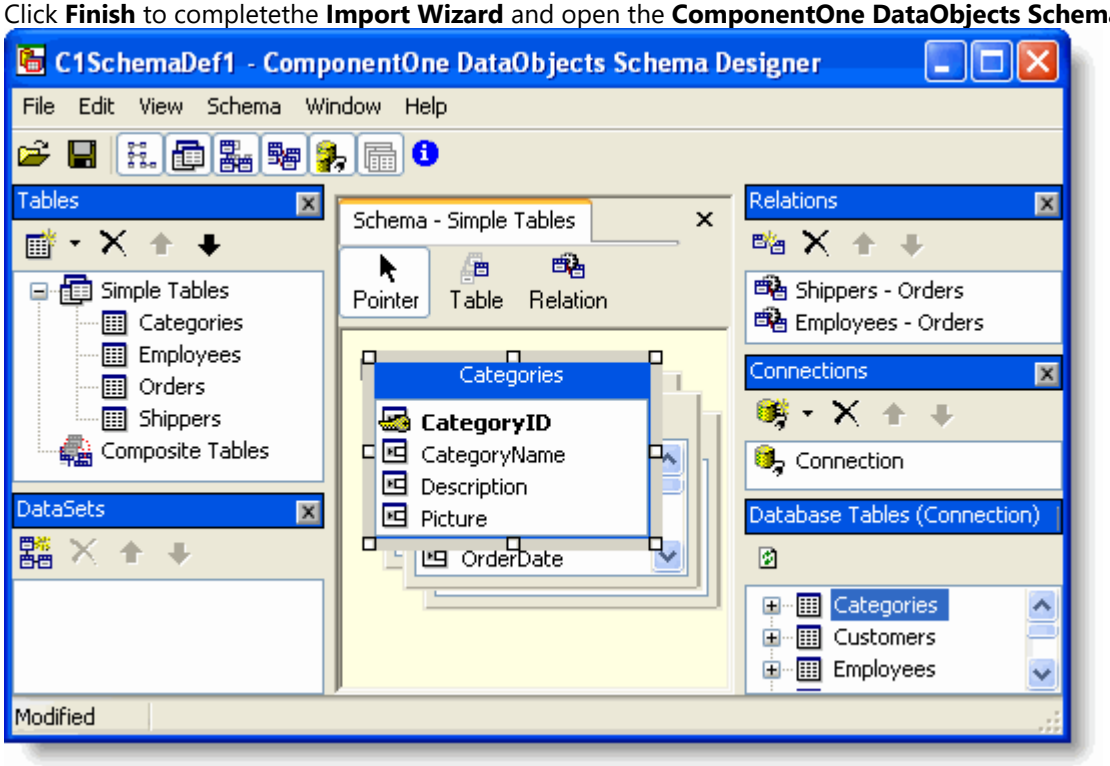
5. Enter any information necessary to connect to the specified database in the remaining fields and click **Next**:

Field	Description
Connection type	Choose from Custom, MSOracle, OLE DB, Oracle, SQL Server
User name	Username used to connect to the database
Password	Password to connect to the database
Connection timeout	Timeout, in seconds
Import objects	Types of objects to import from the database: tables, views, aliases, or any combination of the these

6. Select objects for import from the list of **Available tables** (also, views and aliases, if included) in the database structure. Click the arrows to move the tables to the **Create schema tables** box. There is also a check box that determines whether available foreign key information is used to create relations. If it is unchecked, no relations will be created.



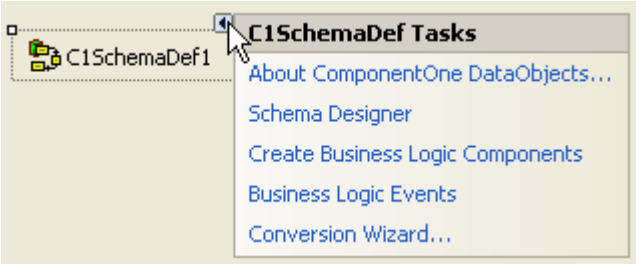
7. Click **Finish** to completethe **Import Wizard** and open the **ComponentOne DataObjects Schema Designer**.



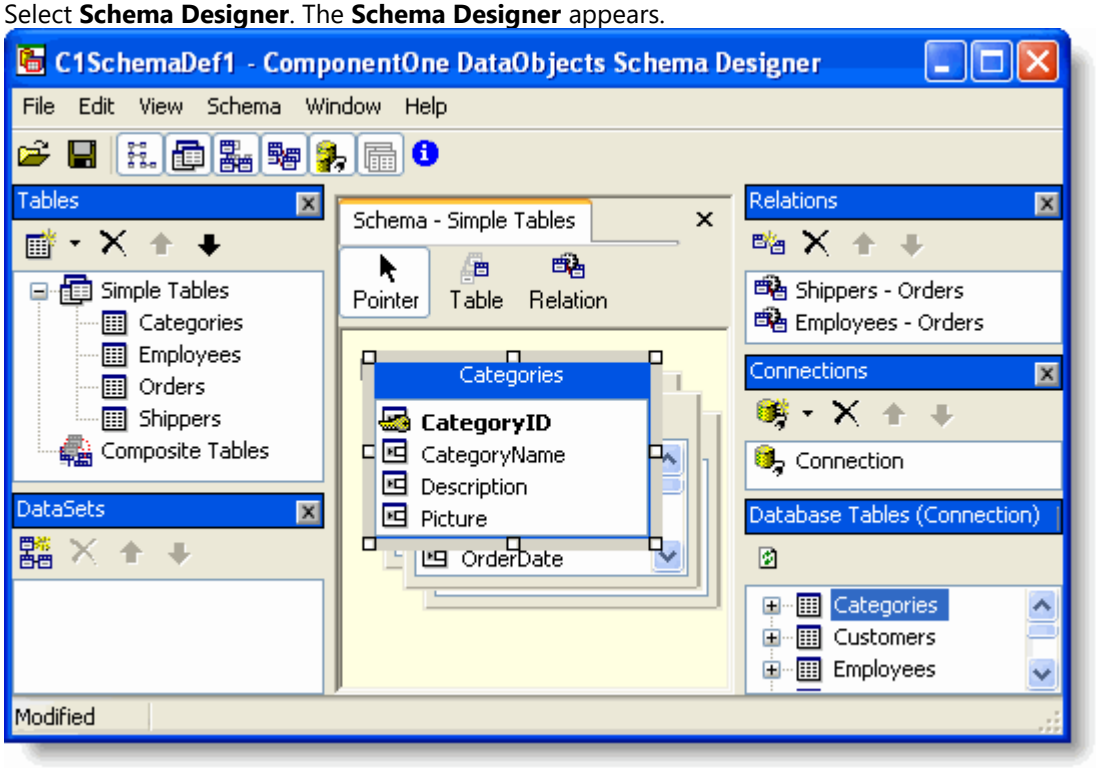
Creating and Customizing the Schema using the Schema Designer

The following topic explains how the **Schema Designer** can be used to create your schema.

1. To open the **Schema Designer**, select the C1SchemaDef component and use the smart tag to open the **C1SchemaDef Tasks** menu.



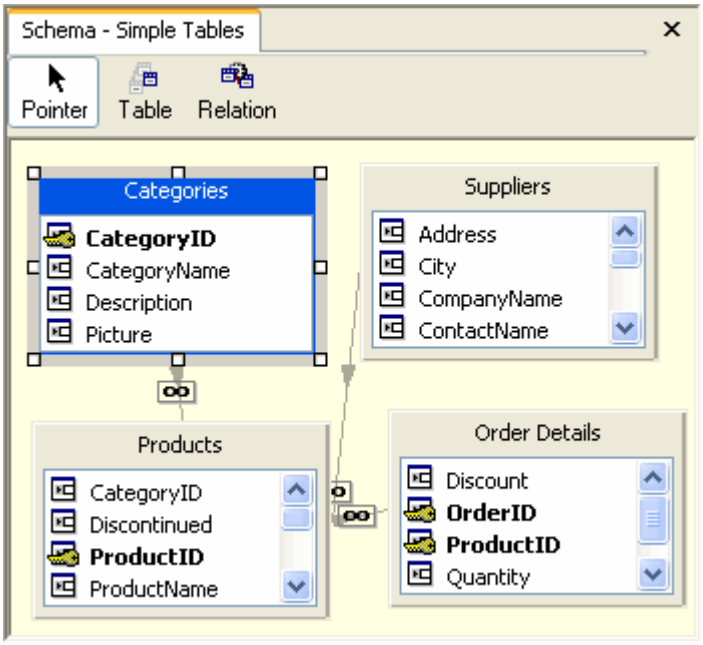
2. Select **Schema Designer**. The **Schema Designer** appears.



> **Note:** If you have not imported a database structure, you must use the **Import Wizard** to do so before using the **Schema Designer**.

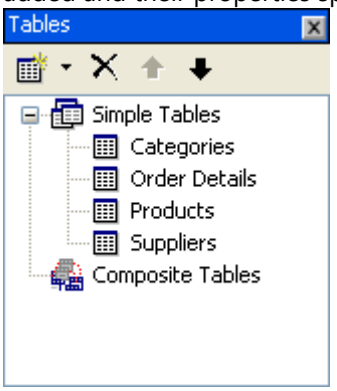
3. Use the **View** menu to open each window and begin customizing the schema. Select any of the following windows:

Window	Description
Schema graph	Tables and relations are represented in a graphical form in the center **Schema – Simple Tables** window. You can rearrange tables through a drag-and-drop operation.



Tables

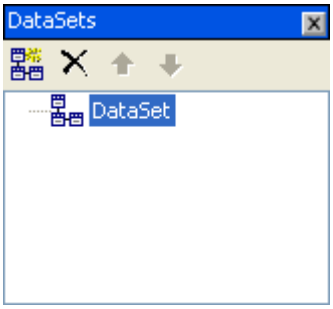
The **Tables** window contains the list of all simple and composite table objects created by the **Import Wizard** based on the database tables. Double-clicking a table opens the **Table Editor**, where tables can be added and their properties specified.



For more information on creating and customizing tables, see Simple Tables or Composite Tables.

DataSets

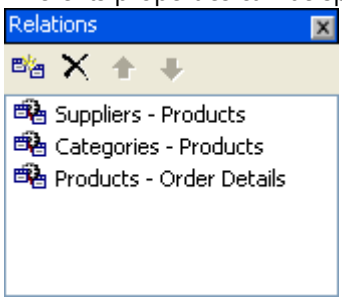
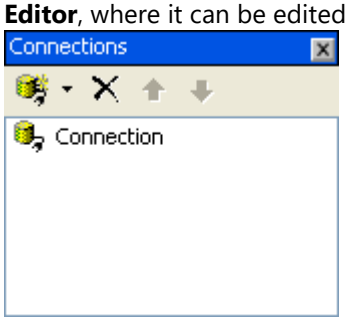
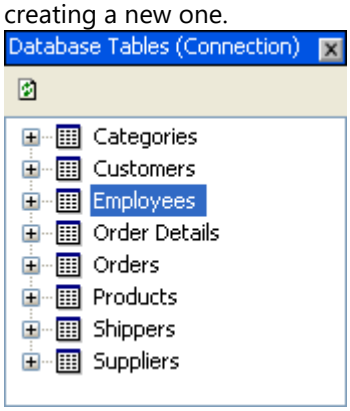
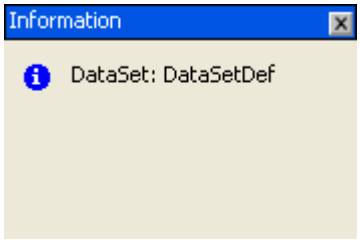
The **DataSets** window allows you to create and customize data set definitions. Double-clicking a data set opens the **DataSet Editor**, where table views can be added and their properties specified.



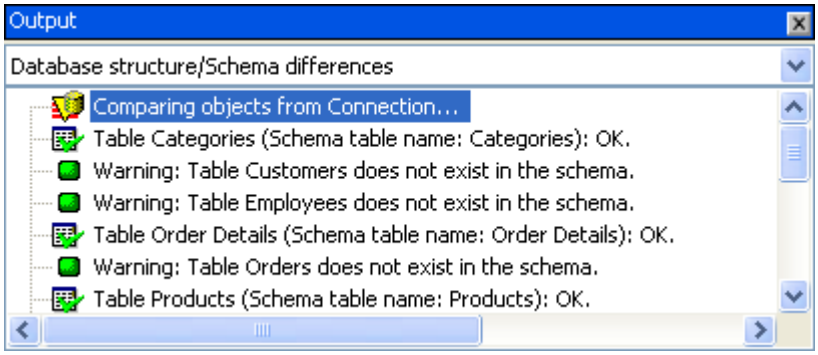
For more information, see Data Sets.

Relations

The **Relations** window contains inter-table relations created by the **Import Wizard** based on the relationships existing in the database. Double-clicking one of the relations opens the **Relations Editor**,

	where its properties can be specified.  For more information, see Simple Relations.
Connections	The **Connections** window allows you to define a connection(s) to a database(s). Double-clicking a connection opens the **Connection Editor**, where it can be edited and its properties can be specified.  For more information, see Database Connections.
Database tables	The **Database Tables** window contains a list of all the tables within the connected database. Change the database connection by clicking the **Connect to database** button and selecting a connection or creating a new one. 
Information	The **Information** window provides information on the item currently selected within the **Schema Designer**. For example, if the *Products – Order Details* relation is selected, the Information window shows the relation and states the relation type, SimpleRelation. 
Output	Select **Verify schema** from the **Schema** menu, and any errors in the schema are displayed in the **Output** window. Select **Compare Schema**

with database structure from the Schema menu to scan the schema and database structure. Any differences between them appear in the **Output** window. Use the drop-down arrow to switch between the **Verification** and **Database structure/Schema differences** views.



For more information, see Database Structure Evolution and the Schema.

Viewing the SQL Statement Generated when a TableView is Sorted

When a **TableView** is sorted, the generated SQL statement is available in the AfterGenerateSql event of the C1DataSetLogic component. To configure this component, follow these steps:

1. Add a C1DataSetLogic component to your project in the same location as the C1SchemaDef component (usually within a data library project).
2. Select C1DataSetLogic1 and set its SchemaComponent property to the C1SchemaDef component.
3. Set C1DataSetLogic1's DataSetDef property to the desired data set.
4. Create an event handler for the AfterGenerateSql event.
5. Set a new breakpoint there:
    1. In the **Debug** menu, select **New Breakpoint | Break at Function**.
    2. Enter **C1DataSetLogic1_AfterGenerateSql** in the Function text box and click **OK**.
6. Run the project and expand **e{C1.Data.GenerateSqlEventArgs}**.

The **e.Sql** argument contains the generated SQL statement.