Chapter 15 – Working with Databases

## Objectives

* Understand Microsoft’s database models
* Working with databases through the Wizards
* Working with databases through VB Code
* LINQ To SQL
* Review of ADO.NET Related Classes
* Using SQL

It's taken us a while to get here, but we are finally ready to look at working with databases from inside of VB.NET. We will begin with a brief introduction to the various database models that Microsoft has created over time.

We will then learn how to display the contents of our database through various controls that we can place on our application's forms. We will also learn how to directly drive everything through VB code: from the creation of the database through SQL (Structured Query Language) queries.

The chapter will end with a concise discussion and description of the SQL language. Including how to return values from the database, how to insert values into the database, how to change values in the database and how to remove values from the database.

# Understanding Microsoft's Database Models

One of the earliest strengths that VB provided to developers was an integrated approach to working with database management systems (DBMS). Microsoft Access, of course, was typically the database of choice since it was Microsoft's database solution at the time. Since at least VB 3, VB programmers have had the ability to read, write and use Access files natively from within VB. Microsoft calls this their Jet technology and the Jet Engine is built into Visual Basic. In fact many programmers think of Access as being little more than a pretty wrapper around the Jet Engine.

Microsoft came out with a slew of different database technologies over the years. The primary reason why there are so many different options is that each option was an evolutionary step forward, bringing more power and productivity to developers. Let's pause for a second and review the major database processing pieces that Microsoft has introduced.

* ***ODBC*** (Open DataBase Connectivity) – ODBC is essentially a piece of middleware that allows a Microsoft Windows user to set up a connection to a 3rd party database such as Oracle, Informix and so forth. The idea is that as new database systems come to the market, only a new ODBC driver would need to be supplied to allow existing applications to work with that new database. This technology has been quite successful and emulated in the Java world, where it is known as JDBC (Java DataBase Connectivity).
* ***DAO*** (Data Access Objects) – This was Microsoft’s first high-level programmatic database accessing system. DAO could talk natively to Jet (Access) databases or utilize ODBC to talk to third party databases. DAO would allow programmers to use standard SQL commands to manipulate the data stored in these databases.
* ***RDO*** (Remote Data Objects) – RDO was a step forward from DAO. Essentially RDO allowed VB programmers to work with databases that might be residing on remote (networked) systems. RDO was only used for a brief amount of time do to ADO quickly following on RDO's heels.
* ***OLE DB*** (Object Linking and Emedding for Databases) – OLE DB was another database attempt by Microsoft that was originally intended as a higher-level replacement for [ODBC](http://en.wikipedia.org/wiki/ODBC), extending its feature set to support a wider variety of non-[relational databases](http://en.wikipedia.org/wiki/Database#Relational), such as [object databases](http://en.wikipedia.org/wiki/Object_database) and [spreadsheets](http://en.wikipedia.org/wiki/Spreadsheet) that do not necessarily implement [SQL](http://en.wikipedia.org/wiki/SQL).
* ***ADO*** (ActiveX Data Objects) – This was Microsoft’s last pre-.NET technology that created lightweight ActiveX (COM) controls that permit web pages to utilize database technologies. Since Microsoft wanted ADO to be able to work with web technology, the controls tend to be fairly small, fast and efficient in their database processing. With the appearance of Visual Studio.NET, Microsoft modified ADO and released ADO.NET. This is *the* way to do databases in VB.NET.

No matter which middleware technology you chose to employ, the lingua franca here is SQL (structured query language). As long as you know it and have a connection to a database, you can pretty much do whatever you want with the data that's contained inside of a database. The last section in this chapter provides a quick summary of using SQL – if you're not familiar with using SQL, you might want to skip ahead and look at it before continuing on with the examples in this chapter.

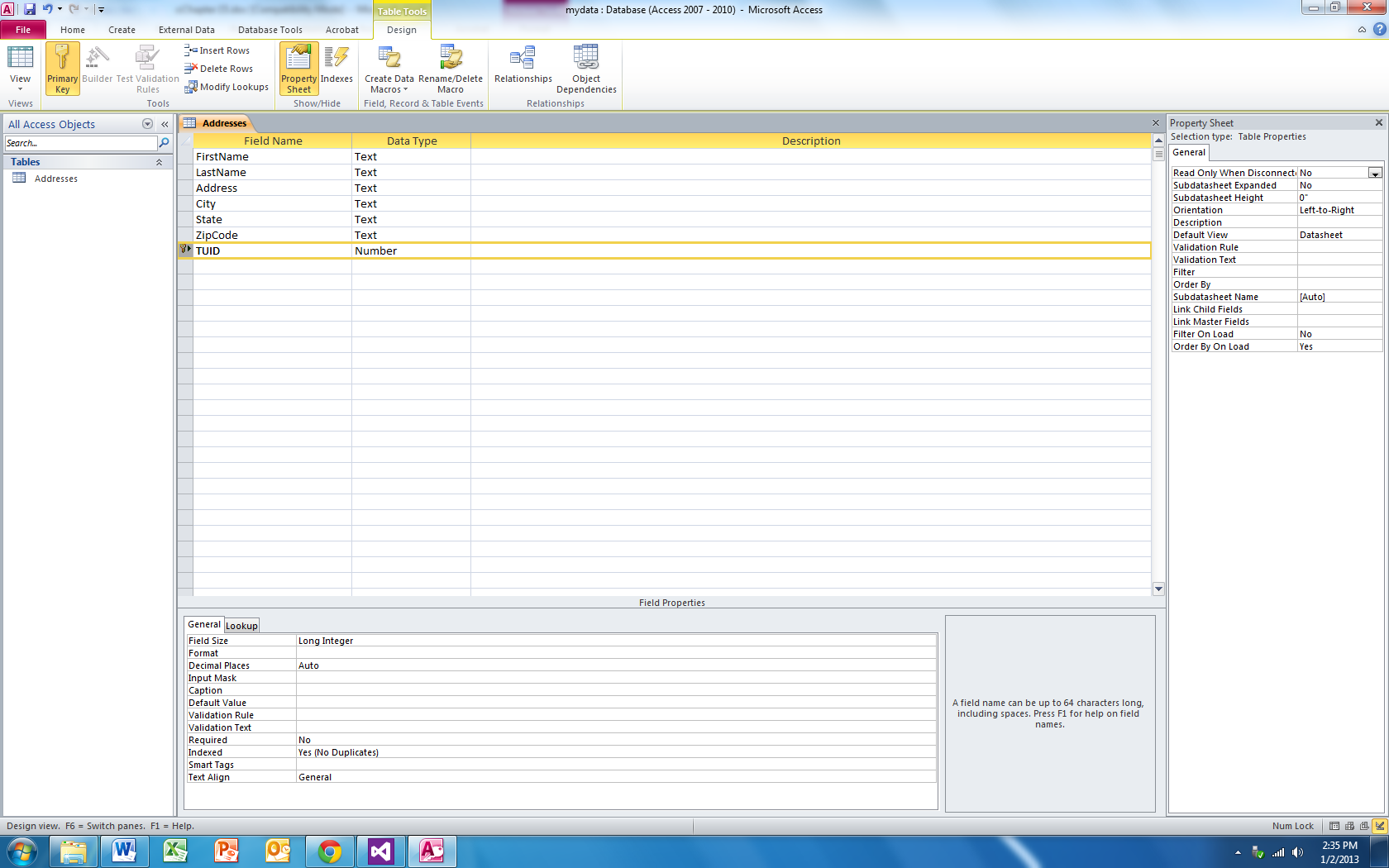
# Working it with the Wizards

As you might expect VB.NET makes it almost ridiculously easy to work with databases. If you have an Access database lying around somewhere, it’s almost a moot point to use Sequential or Direct files anymore. After all, why bother since databases were designed to quickly allow access to as little (or as much) data as a user needs.

Let's begin our example by using Microsoft Access to create a new Access database called C:\DB\mydata.accdb. I am not going to describe how to use Access to you – there are plenty of books available that cover that application. You have to create a folder off the root since Windows won’t let you write directly there.

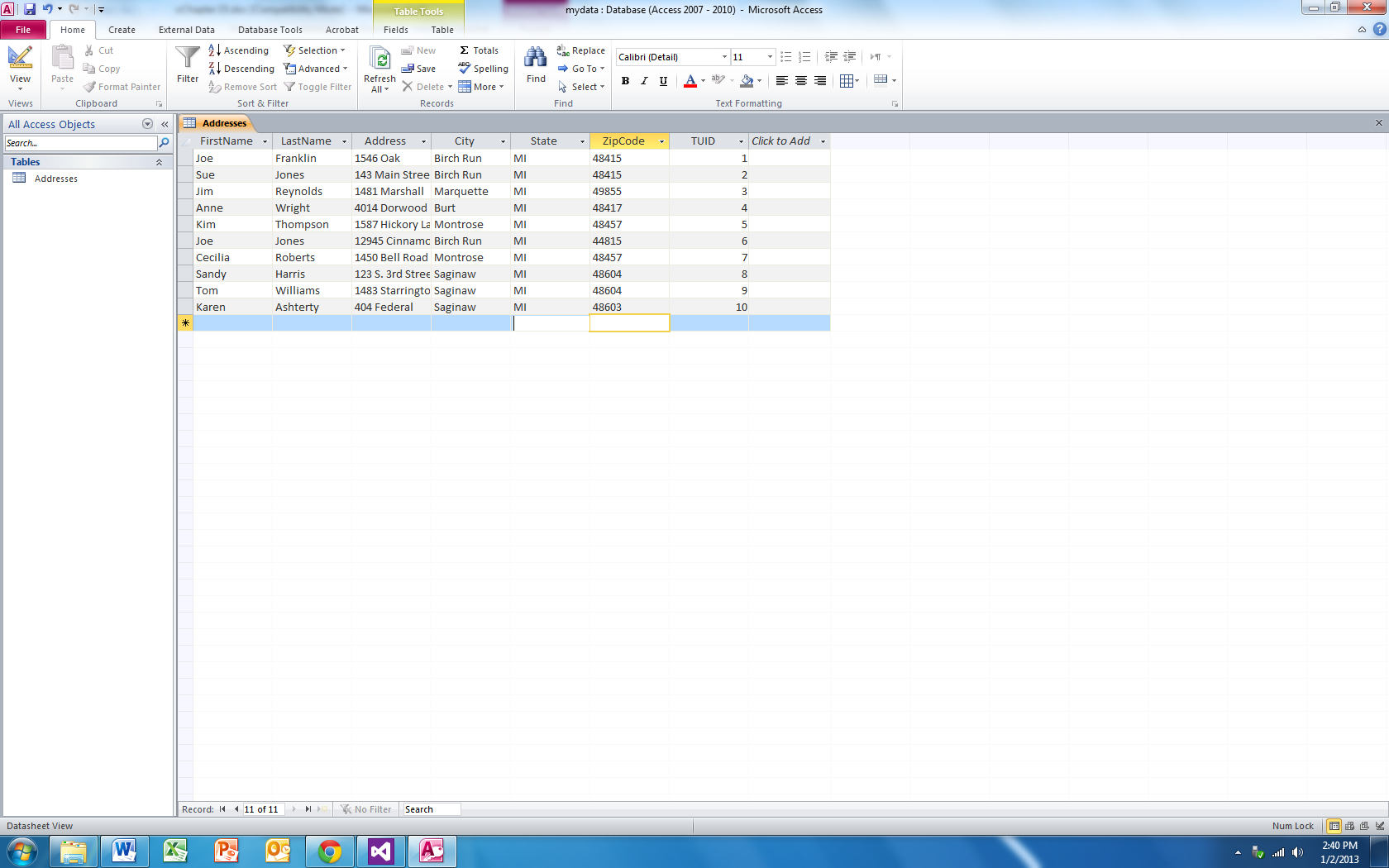
Our database will have a single database table called Addresses placed in it. The table will be comprised of the following fields: FirstName, LastName, Address, City, State, ZipCode and TUID (TUID stands for Table Unique Identifier – it's just a uniquely numbered field so that each row has a unique primary key. Each field will be of type text, except the TUID field which will be Numeric and a Primary Key.

Here is a screenshot of creating the Addresses table inside of Microsoft Access:



Once the database has been created, I will add a couple of records to it from inside of Access. Access also has a SQL statement builder that comes in very handy when you are trying to diagnose why one of your SQL queries doesn’t work right. You may want to invest some time in learning how to utilize Access or its big brother, Microsoft SQL Server – in fact Visual Basic comes with a subset of SQL Server built right in!

Here is another screenshot of Microsoft Access where I am entering in some default records to the Addresses table:



I have placed a copy of the database in the Projects folder inside of Chapter 13 – Program 1. You should copy this database to C:\DB\ folder before continuing on since that is the location where the sample applications expect the database to be located.

Here's a little "behind the scenes" on how ADO.NET works. There are two namespaces that we need to use to work with databases. You will import one or the other depending on which type of database you are working with. The first, System.Data.OleDb, is used for working with Microsoft Access databases and any other non-SQL Server databases. The second namespace is System.Data.SqlClient, which is used exclusively for Microsoft SQL Server databases. Both of these namespaces contain classes to connect to and modify datasources; all of the examples that we will look at in this chapter will use the System.Data.OleDb. Why two different namespaces? The SqlClient is optimized to write high performance SQL Server applications; everything through OleDb works, but isn’t going to be as high performance. Also, it is completely possible to drive SQL Server through the System.Data.OleDb namespace too!

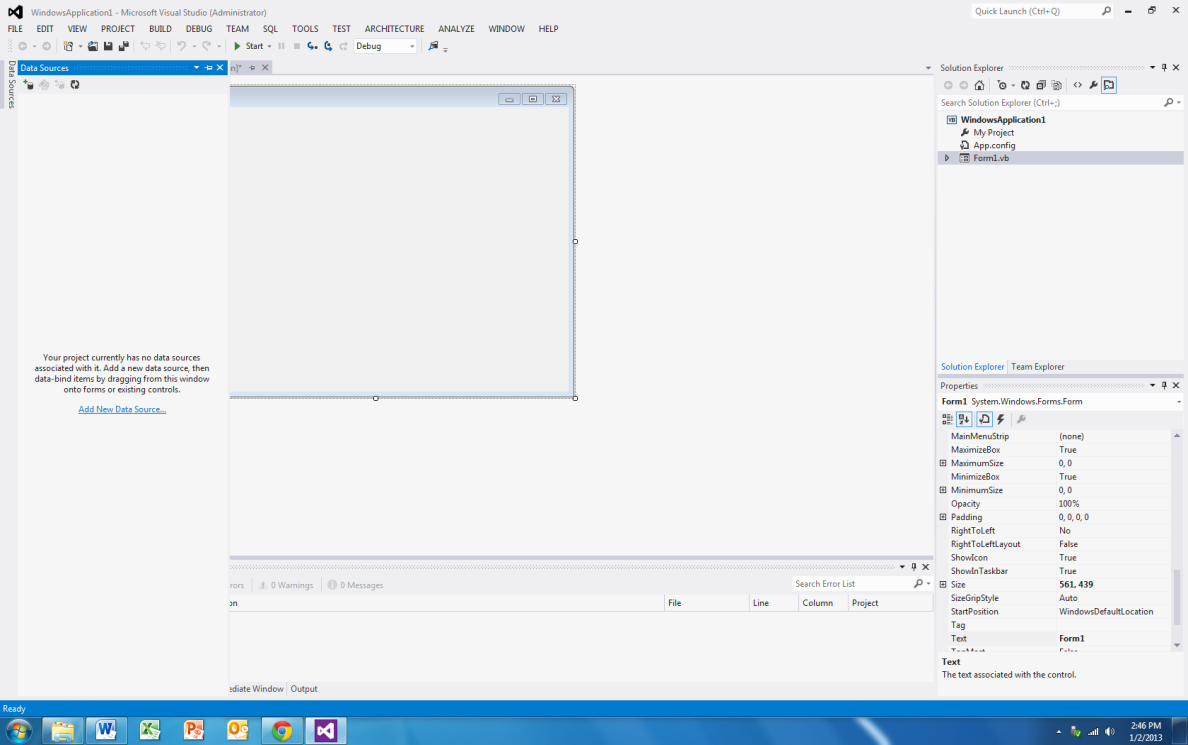
Some of the other classes that you will need to work with when using databases include System.Data.Dataset. This class consists of a set of data returned from a datasource which are a database's tables and its relationships. The class OleDbConnection represents a communication connection to a datasource. The third class, OleDbCommand, is used to issue a SQL command that is to be executed on a datasource.

Here's an overview of how the database connectivity works. You will populate a dataset through an instance of OleDbConnection. Once the dataset is populated, VB.NET will automatically disconnect from the datasource. Only when changes are to be written out to the datasource will the connection be re-established. Again, this will be automatically handled by VB for you. You might wonder why we simply wouldn't keep the initial connection open to the database. By closing the connection and only reopening it when needed, we are able to increase the throughput capabilities of our system and reduce wasted resources.

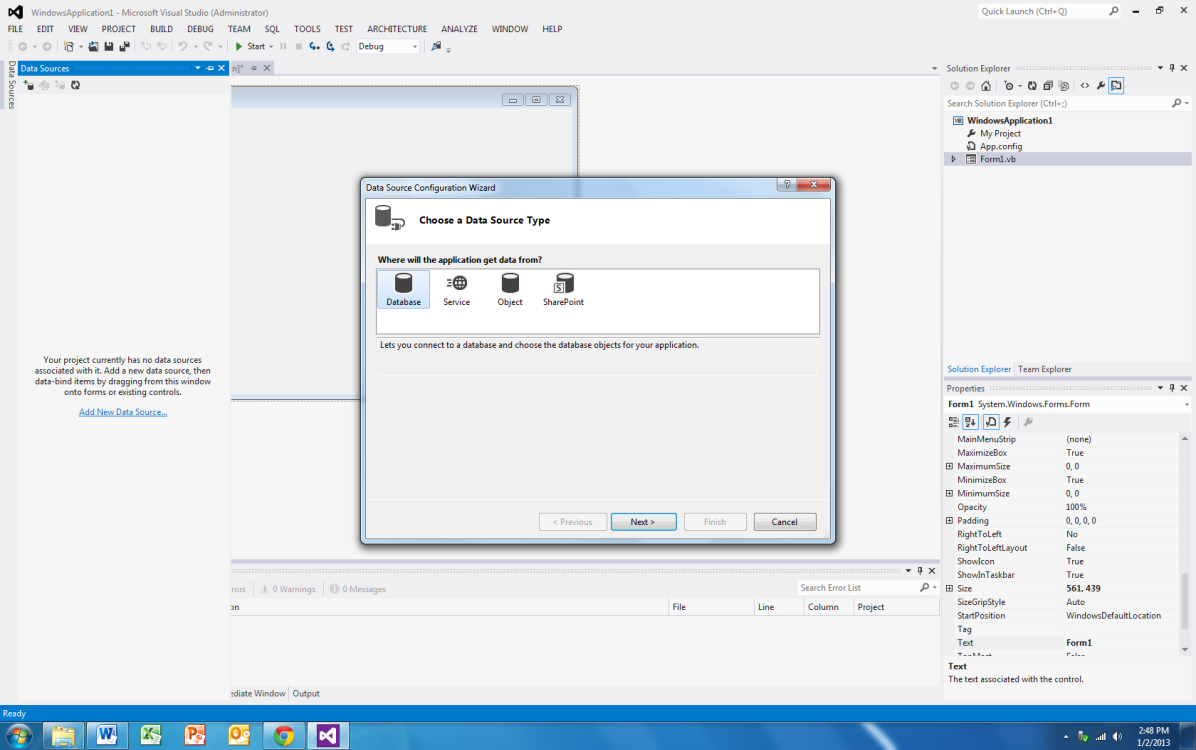
The default connection in ADO.NET is disconnected as we mentioned above. It is also possible to create a connection that stays open from the time we create it until our application closes it. These issues are beyond the time we have to discuss – so we will use the default connection-as-needed ADO.NET mode.

Now that we have the database built, it’s time to access it from Visual Basic. Go ahead by starting up a new VB Windows Application project type. You need to carefully walk through this example to try to understand everything that is taking place. I will provide screenshots detailing each step that I ask you to perform.

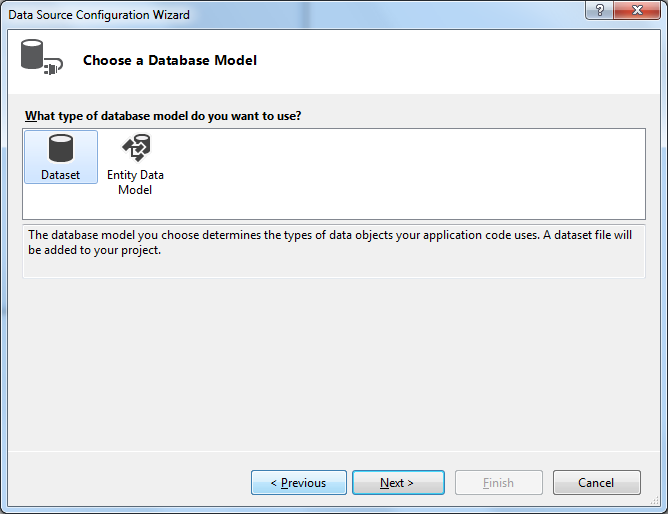
The first thing that you need to do is to create a connection to the database. The following screenshot shows the Data Sources pane activated (View🡪Other Windows🡪Data Sources). The Data Sources is located on a tab right next to the Toolbox tab in the left windowpane of the Visual Studio IDE.



To form a data connection from within .NET to your database, you will left click on the little plus sign icon in the Data Sources. At this point, a choose Data Source Type screen appears:

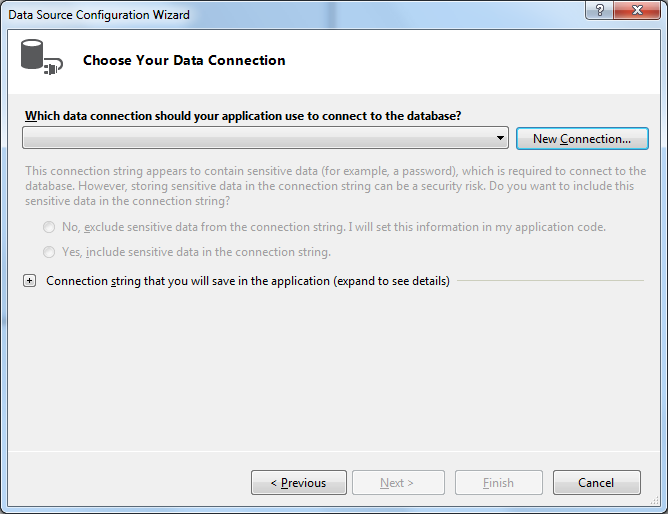


Since we want to work with an Access database, click on Database and press Next. We’re now asked about the database model we want to use.

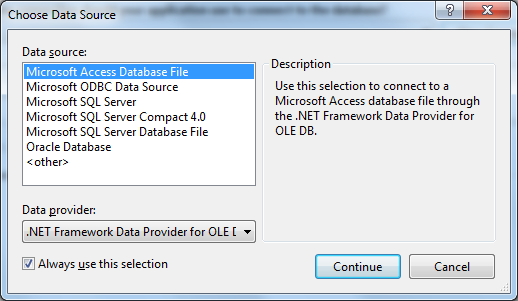


Select Dataset and click Next.

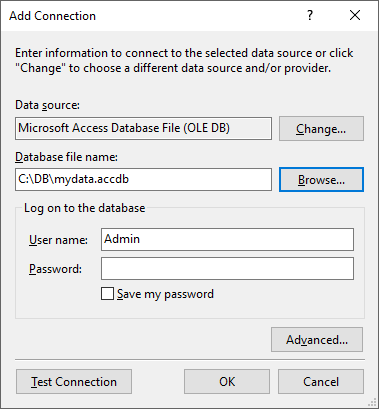
Now we’re asked about our Data Connection (the glue between the application and the database that we identified above:



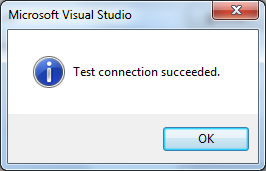
We need to click the New Connection button, since we don’t have a connection to our database set up yet. We get a new screen that pops up asking us about what the data source is. Again we know that it’s Microsoft Access, so select that choice and then click Continue.



Finally we get to the screen where we can browse and point and click on our database. Here’s what the screen looks like after I make my selection:

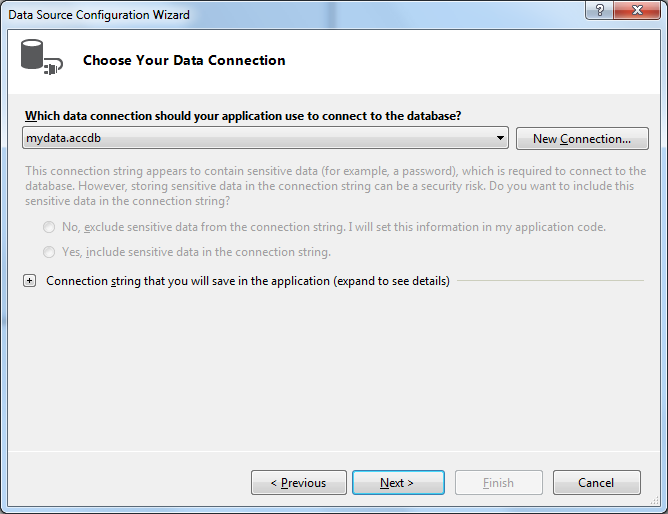


I’m not going to mess with anything other than setting the name. As good practice, though, I am going to make sure that I can talk to the database successfully, so I will press the “Test Connection” button. If everything goes well, I should see the following screen:

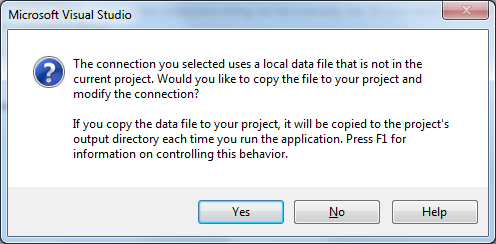


Click OK to close the test screen. That will put you back on the Add Connection screen. Click OK there as well. If you received an error about the “microsoft.ace.oledb.12.0 provider not registered” you will need the Microsoft Access 2010 Redistributable package (which I have). I have found that the 32 bit version is what you need regardless of your actual Windows or Office installation version…it used to be a standard on Windows, but no more…

We should now see a filled out data connection screen:

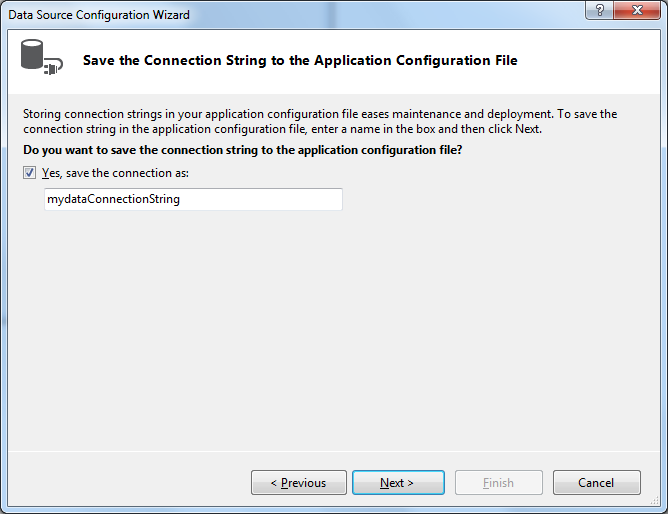


Click Next. Depending on the version of .NET you are using, you may or may not see the next screen which warns about the database not being in the project directory:

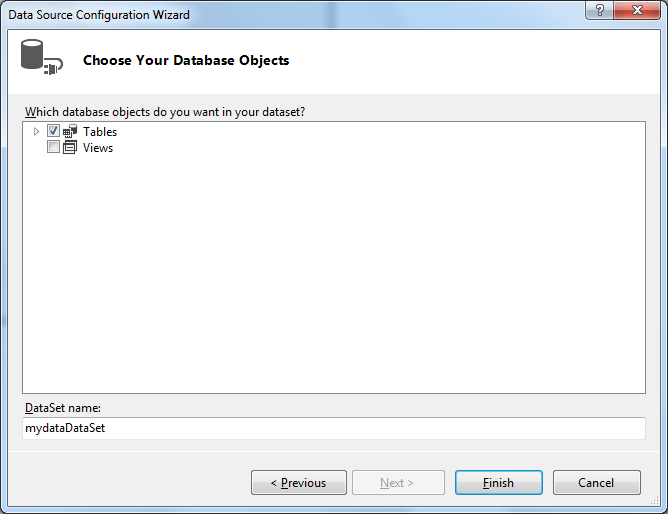


I am going to answer No this time, but in general, we would like our database to be tagged right along with our project files.

I am now asked about saving the connection string as an application configuration file setting.

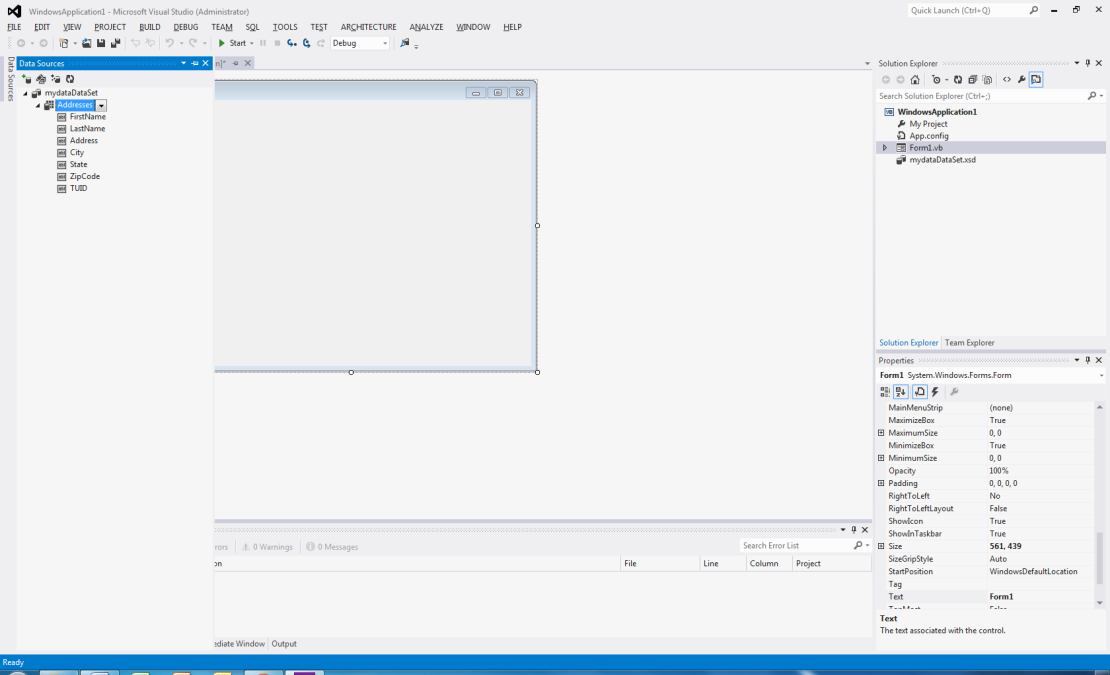


Leaving everything alone, I click Next. Now, I am asked about what database objects I want to use from the database. I will check the Tables checkbox, since that’s where my data is sitting:

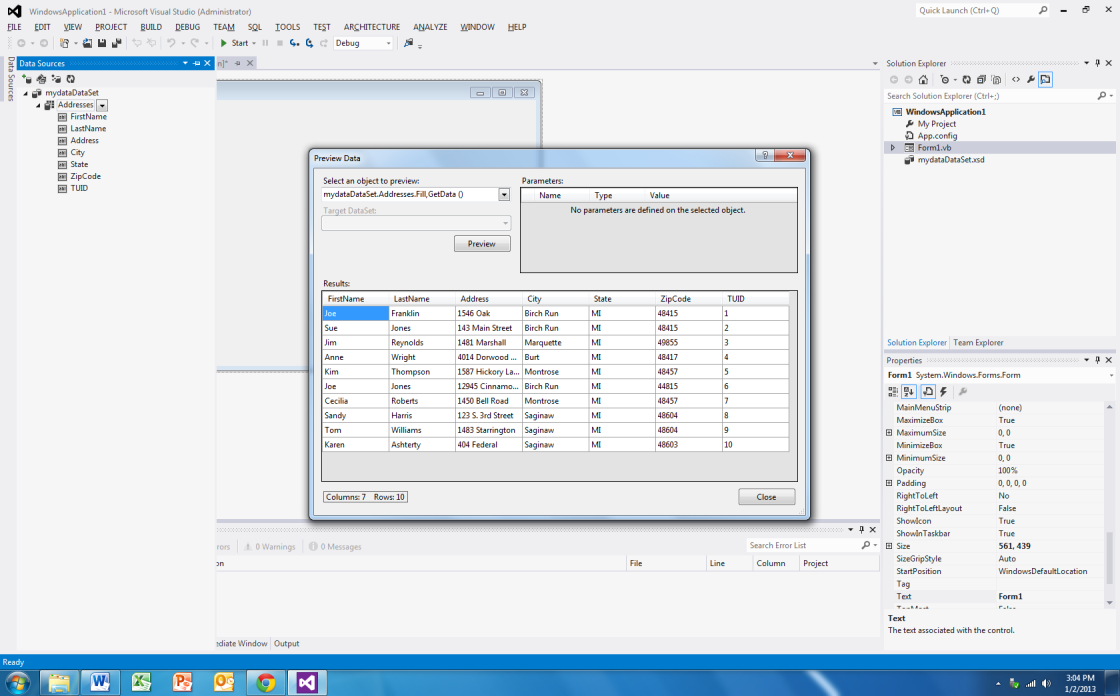


Click Finish! Yes, finally, we have reached the end of the Connection wizard!

All of that trouble, to get VB to “see” and “talk” to the database. I will eventually show you how to wire this up yourself without going through the Connection wizard, but as a database programming beginner, the wizard is a good idea. You can see the connection shows up in the Data Sources window. I can even see all of the fields:

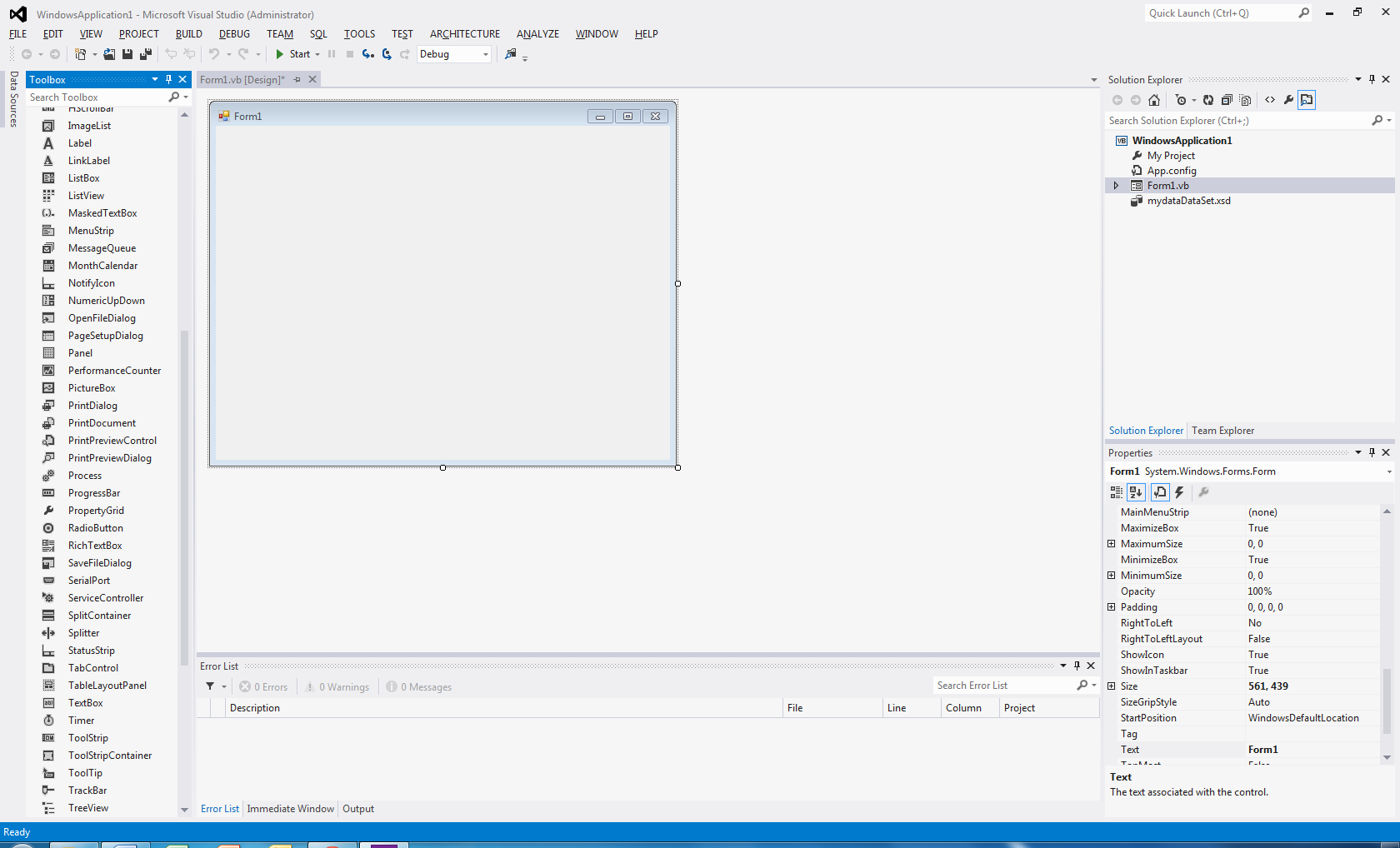


If I right click where it shows the Addresses table name, I am given the option of Preview Data, which when clicked upon returns the following screen:

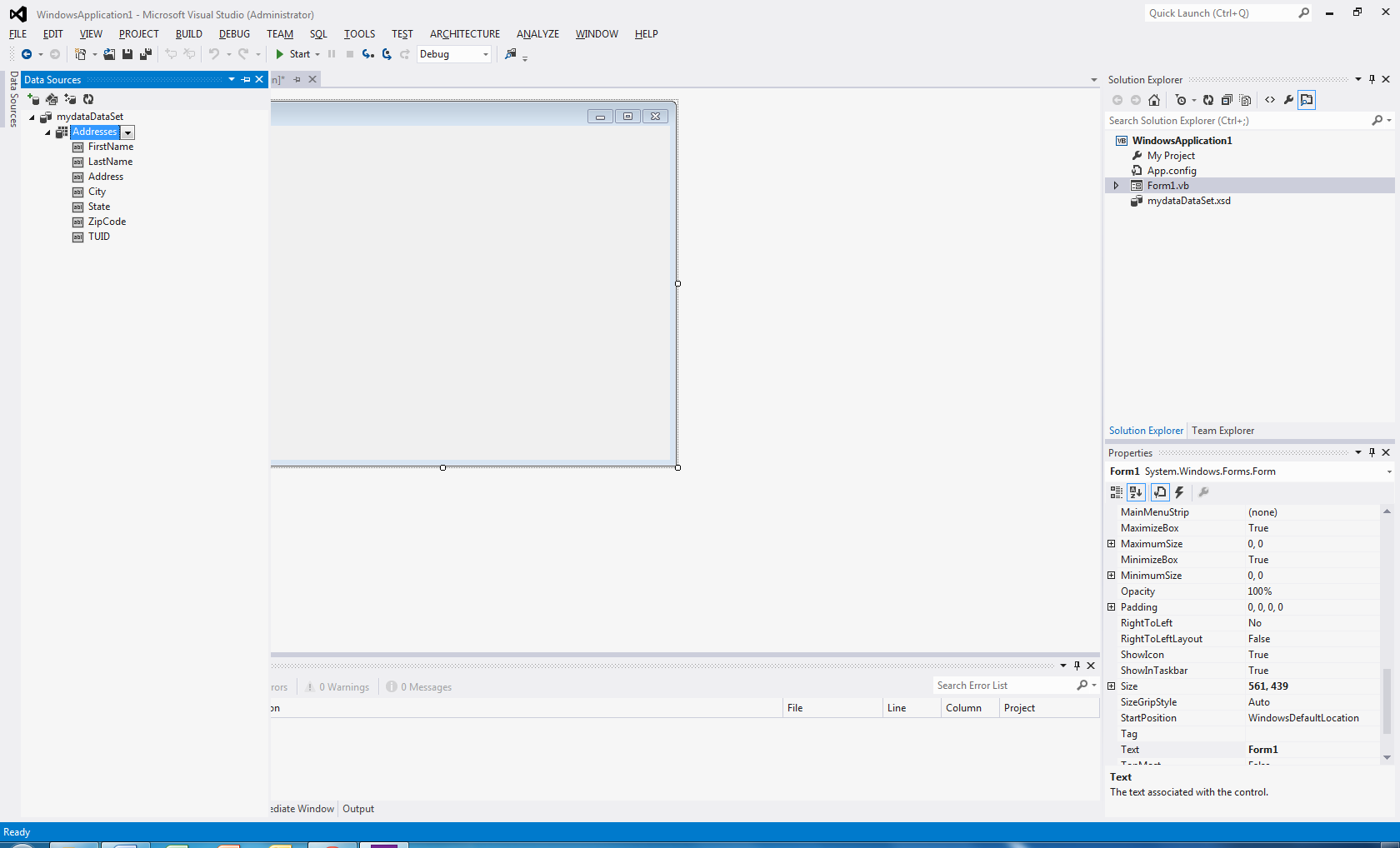


So there’s my data – at least I can view it and make sure that the data connection is allowing me access to the right stuff!

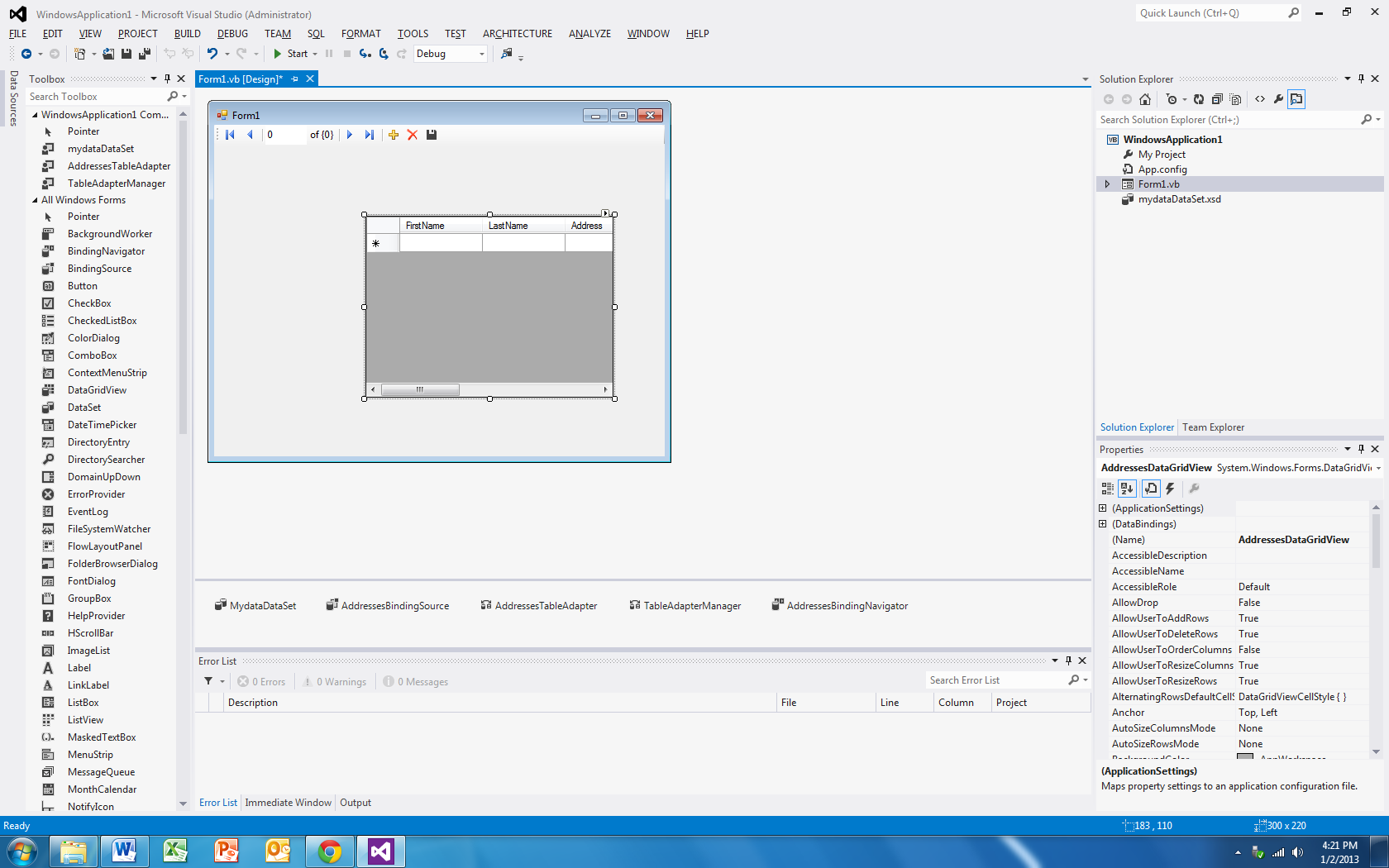
Once we close the preview window, we can shrink the data sources window down by clicking on its name. (If you accidentally close the window and it’s name no longer appears over the Toolbox, you can get it back by View🡪Other Windows🡪Data Sources). Here’s the screen we see:



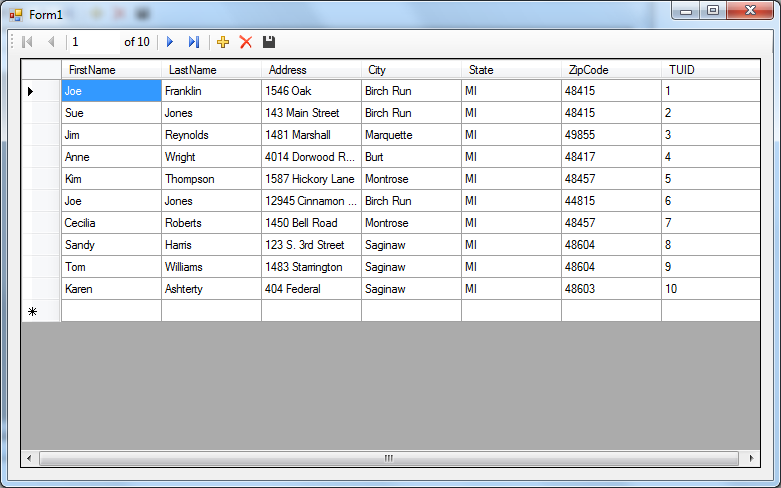
So the next thing we need to do is simply click on either the Addresses table or the mydataDataSet data set and drag and drop a copy from the Data Sources window to the form:



When you let go you’ll see something that looks like this:



Once we do a little bit of resizing and then run the application we get this:



Not too shabby! We can page through our records (we only have 10, but if there were more, we’d be able to move up and down a page at a time). We’ve got the ability to move to the first, last, next and previous records. We can add, delete and save all via a nice menu bar environment. Didn’t take hardly any work on our part and we got a real database application.

Now obviously a bunch of magic went on here. First of all, VB added a prebuilt menu bar and a datagridview control. The datagridview control really is awesome. Essentially you point it at a chunk of data in a database (or in memory) and it’ll display the results in an editable grid form.

If you look along the bottom of the form, a whole bunch of non-graphical controls were added:



Let’s start out with these items in levels of importance:

* AddressesTableAdapter is what allows communication between the data connection (remember all that wizard stuff we had to do?) and the data set.
* MydataDataSet is a container where the contents of the database are placed for our program to work with them. Remember in our case, it is a disconnected dataset that is only updated when necessary.
* TableAdapterManager is another container object that ensures that table behavior is preserved. This means that we have something watching our inserts, updates and deletes to make sure they don’t violate anything and in the case of tables that are joined, the TableAdapterManager will make sure that referential integrity is preserved.
* AddressesBindingSource is what ties the data set’s data into the datagridview on the form.
* AddressesBindingNavigator is what updates the view whenever we move through the bindingsource. If you click on the next, previous, first or last buttons, notice the highlighted row in the datagridview changes. This is due to the binding navigator.

So what code was written for us? Here’s the code behind the form:

'Chapter 15 - Program 1

Public Class Form1

Private Sub AddressesBindingNavigatorSaveItem\_Click(sender As Object, e As EventArgs) Handles AddressesBindingNavigatorSaveItem.Click

Me.Validate()

Me.AddressesBindingSource.EndEdit()

Me.TableAdapterManager.UpdateAll(Me.MydataDataSet)

End Sub

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

'TODO: This line of code loads data into the 'MydataDataSet.Addresses' table. You can move, or remove it, as needed.

Me.AddressesTableAdapter.Fill(Me.MydataDataSet.Addresses)

End Sub

End Class

Wow! There’s not much too that – in fact it looks like we fill things up in Load and then everything else just deals with changes in the bindingnavigator. How hard can it be to learn to do things like this? Well, let’s look at the form’s Designer code (remember to see this, we have to show all files):

<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> \_

Partial Class Form1

Inherits System.Windows.Forms.Form

'Form overrides dispose to clean up the component list.

<System.Diagnostics.DebuggerNonUserCode()> \_

Protected Overrides Sub Dispose(ByVal disposing As Boolean)

Try

If disposing AndAlso components IsNot Nothing Then

components.Dispose()

End If

Finally

MyBase.Dispose(disposing)

End Try

End Sub

'Required by the Windows Form Designer

Private components As System.ComponentModel.IContainer

'NOTE: The following procedure is required by the Windows Form Designer

'It can be modified using the Windows Form Designer.

'Do not modify it using the code editor.

<System.Diagnostics.DebuggerStepThrough()> \_

Private Sub InitializeComponent()

Me.components = New System.ComponentModel.Container()

Dim resources As System.ComponentModel.ComponentResourceManager = New System.ComponentModel.ComponentResourceManager(GetType(Form1))

Me.MydataDataSet = New Program1.mydataDataSet()

Me.AddressesBindingSource = New System.Windows.Forms.BindingSource(Me.components)

Me.AddressesTableAdapter = New Program1.mydataDataSetTableAdapters.AddressesTableAdapter()

Me.TableAdapterManager = New Program1.mydataDataSetTableAdapters.TableAdapterManager()

Me.AddressesBindingNavigator = New System.Windows.Forms.BindingNavigator(Me.components)

Me.BindingNavigatorMoveFirstItem = New System.Windows.Forms.ToolStripButton()

Me.BindingNavigatorMovePreviousItem = New System.Windows.Forms.ToolStripButton()

Me.BindingNavigatorSeparator = New System.Windows.Forms.ToolStripSeparator()

Me.BindingNavigatorPositionItem = New System.Windows.Forms.ToolStripTextBox()

Me.BindingNavigatorCountItem = New System.Windows.Forms.ToolStripLabel()

Me.BindingNavigatorSeparator1 = New System.Windows.Forms.ToolStripSeparator()

Me.BindingNavigatorMoveNextItem = New System.Windows.Forms.ToolStripButton()

Me.BindingNavigatorMoveLastItem = New System.Windows.Forms.ToolStripButton()

Me.BindingNavigatorSeparator2 = New System.Windows.Forms.ToolStripSeparator()

Me.BindingNavigatorAddNewItem = New System.Windows.Forms.ToolStripButton()

Me.BindingNavigatorDeleteItem = New System.Windows.Forms.ToolStripButton()

Me.AddressesBindingNavigatorSaveItem = New System.Windows.Forms.ToolStripButton()

Me.AddressesDataGridView = New System.Windows.Forms.DataGridView()

Me.DataGridViewTextBoxColumn1 = New System.Windows.Forms.DataGridViewTextBoxColumn()

Me.DataGridViewTextBoxColumn2 = New System.Windows.Forms.DataGridViewTextBoxColumn()

Me.DataGridViewTextBoxColumn3 = New System.Windows.Forms.DataGridViewTextBoxColumn()

Me.DataGridViewTextBoxColumn4 = New System.Windows.Forms.DataGridViewTextBoxColumn()

Me.DataGridViewTextBoxColumn5 = New System.Windows.Forms.DataGridViewTextBoxColumn()

Me.DataGridViewTextBoxColumn6 = New System.Windows.Forms.DataGridViewTextBoxColumn()

Me.DataGridViewTextBoxColumn7 = New System.Windows.Forms.DataGridViewTextBoxColumn()

CType(Me.MydataDataSet, System.ComponentModel.ISupportInitialize).BeginInit()

CType(Me.AddressesBindingSource, System.ComponentModel.ISupportInitialize).BeginInit()

CType(Me.AddressesBindingNavigator, System.ComponentModel.ISupportInitialize).BeginInit()

Me.AddressesBindingNavigator.SuspendLayout()

CType(Me.AddressesDataGridView, System.ComponentModel.ISupportInitialize).BeginInit()

Me.SuspendLayout()

'

'MydataDataSet

'

Me.MydataDataSet.DataSetName = "mydataDataSet"

Me.MydataDataSet.SchemaSerializationMode = System.Data.SchemaSerializationMode.IncludeSchema

'

'AddressesBindingSource

'

Me.AddressesBindingSource.DataMember = "Addresses"

Me.AddressesBindingSource.DataSource = Me.MydataDataSet

'

'AddressesTableAdapter

'

Me.AddressesTableAdapter.ClearBeforeFill = True

'

'TableAdapterManager

'

Me.TableAdapterManager.AddressesTableAdapter = Me.AddressesTableAdapter

Me.TableAdapterManager.BackupDataSetBeforeUpdate = False

Me.TableAdapterManager.UpdateOrder = Program1.mydataDataSetTableAdapters.TableAdapterManager.UpdateOrderOption.InsertUpdateDelete

'

'AddressesBindingNavigator

'

Me.AddressesBindingNavigator.AddNewItem = Me.BindingNavigatorAddNewItem

Me.AddressesBindingNavigator.BindingSource = Me.AddressesBindingSource

Me.AddressesBindingNavigator.CountItem = Me.BindingNavigatorCountItem

Me.AddressesBindingNavigator.DeleteItem = Me.BindingNavigatorDeleteItem

Me.AddressesBindingNavigator.Items.AddRange(New System.Windows.Forms.ToolStripItem() {Me.BindingNavigatorMoveFirstItem, Me.BindingNavigatorMovePreviousItem, Me.BindingNavigatorSeparator, Me.BindingNavigatorPositionItem, Me.BindingNavigatorCountItem, Me.BindingNavigatorSeparator1, Me.BindingNavigatorMoveNextItem, Me.BindingNavigatorMoveLastItem, Me.BindingNavigatorSeparator2, Me.BindingNavigatorAddNewItem, Me.BindingNavigatorDeleteItem, Me.AddressesBindingNavigatorSaveItem})

Me.AddressesBindingNavigator.Location = New System.Drawing.Point(0, 0)

Me.AddressesBindingNavigator.MoveFirstItem = Me.BindingNavigatorMoveFirstItem

Me.AddressesBindingNavigator.MoveLastItem = Me.BindingNavigatorMoveLastItem

Me.AddressesBindingNavigator.MoveNextItem = Me.BindingNavigatorMoveNextItem

Me.AddressesBindingNavigator.MovePreviousItem = Me.BindingNavigatorMovePreviousItem

Me.AddressesBindingNavigator.Name = "AddressesBindingNavigator"

Me.AddressesBindingNavigator.PositionItem = Me.BindingNavigatorPositionItem

Me.AddressesBindingNavigator.Size = New System.Drawing.Size(578, 25)

Me.AddressesBindingNavigator.TabIndex = 0

Me.AddressesBindingNavigator.Text = "BindingNavigator1"

'

'BindingNavigatorMoveFirstItem

'

Me.BindingNavigatorMoveFirstItem.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image

Me.BindingNavigatorMoveFirstItem.Image = CType(resources.GetObject("BindingNavigatorMoveFirstItem.Image"), System.Drawing.Image)

Me.BindingNavigatorMoveFirstItem.Name = "BindingNavigatorMoveFirstItem"

Me.BindingNavigatorMoveFirstItem.RightToLeftAutoMirrorImage = True

Me.BindingNavigatorMoveFirstItem.Size = New System.Drawing.Size(23, 22)

Me.BindingNavigatorMoveFirstItem.Text = "Move first"

'

'BindingNavigatorMovePreviousItem

'

Me.BindingNavigatorMovePreviousItem.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image

Me.BindingNavigatorMovePreviousItem.Image = CType(resources.GetObject("BindingNavigatorMovePreviousItem.Image"), System.Drawing.Image)

Me.BindingNavigatorMovePreviousItem.Name = "BindingNavigatorMovePreviousItem"

Me.BindingNavigatorMovePreviousItem.RightToLeftAutoMirrorImage = True

Me.BindingNavigatorMovePreviousItem.Size = New System.Drawing.Size(23, 22)

Me.BindingNavigatorMovePreviousItem.Text = "Move previous"

'

'BindingNavigatorSeparator

'

Me.BindingNavigatorSeparator.Name = "BindingNavigatorSeparator"

Me.BindingNavigatorSeparator.Size = New System.Drawing.Size(6, 25)

'

'BindingNavigatorPositionItem

'

Me.BindingNavigatorPositionItem.AccessibleName = "Position"

Me.BindingNavigatorPositionItem.AutoSize = False

Me.BindingNavigatorPositionItem.Name = "BindingNavigatorPositionItem"

Me.BindingNavigatorPositionItem.Size = New System.Drawing.Size(50, 23)

Me.BindingNavigatorPositionItem.Text = "0"

Me.BindingNavigatorPositionItem.ToolTipText = "Current position"

'

'BindingNavigatorCountItem

'

Me.BindingNavigatorCountItem.Name = "BindingNavigatorCountItem"

Me.BindingNavigatorCountItem.Size = New System.Drawing.Size(35, 15)

Me.BindingNavigatorCountItem.Text = "of {0}"

Me.BindingNavigatorCountItem.ToolTipText = "Total number of items"

'

'BindingNavigatorSeparator1

'

Me.BindingNavigatorSeparator1.Name = "BindingNavigatorSeparator"

Me.BindingNavigatorSeparator1.Size = New System.Drawing.Size(6, 6)

'

'BindingNavigatorMoveNextItem

'

Me.BindingNavigatorMoveNextItem.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image

Me.BindingNavigatorMoveNextItem.Image = CType(resources.GetObject("BindingNavigatorMoveNextItem.Image"), System.Drawing.Image)

Me.BindingNavigatorMoveNextItem.Name = "BindingNavigatorMoveNextItem"

Me.BindingNavigatorMoveNextItem.RightToLeftAutoMirrorImage = True

Me.BindingNavigatorMoveNextItem.Size = New System.Drawing.Size(23, 20)

Me.BindingNavigatorMoveNextItem.Text = "Move next"

'

'BindingNavigatorMoveLastItem

'

Me.BindingNavigatorMoveLastItem.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image

Me.BindingNavigatorMoveLastItem.Image = CType(resources.GetObject("BindingNavigatorMoveLastItem.Image"), System.Drawing.Image)

Me.BindingNavigatorMoveLastItem.Name = "BindingNavigatorMoveLastItem"

Me.BindingNavigatorMoveLastItem.RightToLeftAutoMirrorImage = True

Me.BindingNavigatorMoveLastItem.Size = New System.Drawing.Size(23, 20)

Me.BindingNavigatorMoveLastItem.Text = "Move last"

'

'BindingNavigatorSeparator2

'

Me.BindingNavigatorSeparator2.Name = "BindingNavigatorSeparator"

Me.BindingNavigatorSeparator2.Size = New System.Drawing.Size(6, 6)

'

'BindingNavigatorAddNewItem

'

Me.BindingNavigatorAddNewItem.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image

Me.BindingNavigatorAddNewItem.Image = CType(resources.GetObject("BindingNavigatorAddNewItem.Image"), System.Drawing.Image)

Me.BindingNavigatorAddNewItem.Name = "BindingNavigatorAddNewItem"

Me.BindingNavigatorAddNewItem.RightToLeftAutoMirrorImage = True

Me.BindingNavigatorAddNewItem.Size = New System.Drawing.Size(23, 22)

Me.BindingNavigatorAddNewItem.Text = "Add new"

'

'BindingNavigatorDeleteItem

'

Me.BindingNavigatorDeleteItem.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image

Me.BindingNavigatorDeleteItem.Image = CType(resources.GetObject("BindingNavigatorDeleteItem.Image"), System.Drawing.Image)

Me.BindingNavigatorDeleteItem.Name = "BindingNavigatorDeleteItem"

Me.BindingNavigatorDeleteItem.RightToLeftAutoMirrorImage = True

Me.BindingNavigatorDeleteItem.Size = New System.Drawing.Size(23, 20)

Me.BindingNavigatorDeleteItem.Text = "Delete"

'

'AddressesBindingNavigatorSaveItem

'

Me.AddressesBindingNavigatorSaveItem.DisplayStyle = System.Windows.Forms.ToolStripItemDisplayStyle.Image

Me.AddressesBindingNavigatorSaveItem.Image = CType(resources.GetObject("AddressesBindingNavigatorSaveItem.Image"), System.Drawing.Image)

Me.AddressesBindingNavigatorSaveItem.Name = "AddressesBindingNavigatorSaveItem"

Me.AddressesBindingNavigatorSaveItem.Size = New System.Drawing.Size(23, 23)

Me.AddressesBindingNavigatorSaveItem.Text = "Save Data"

'

'AddressesDataGridView

'

Me.AddressesDataGridView.AutoGenerateColumns = False

Me.AddressesDataGridView.ColumnHeadersHeightSizeMode = System.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize

Me.AddressesDataGridView.Columns.AddRange(New System.Windows.Forms.DataGridViewColumn() {Me.DataGridViewTextBoxColumn1, Me.DataGridViewTextBoxColumn2, Me.DataGridViewTextBoxColumn3, Me.DataGridViewTextBoxColumn4, Me.DataGridViewTextBoxColumn5, Me.DataGridViewTextBoxColumn6, Me.DataGridViewTextBoxColumn7})

Me.AddressesDataGridView.DataSource = Me.AddressesBindingSource

Me.AddressesDataGridView.Dock = System.Windows.Forms.DockStyle.Fill

Me.AddressesDataGridView.Location = New System.Drawing.Point(0, 25)

Me.AddressesDataGridView.Name = "AddressesDataGridView"

Me.AddressesDataGridView.Size = New System.Drawing.Size(578, 274)

Me.AddressesDataGridView.TabIndex = 1

'

'DataGridViewTextBoxColumn1

'

Me.DataGridViewTextBoxColumn1.DataPropertyName = "FirstName"

Me.DataGridViewTextBoxColumn1.HeaderText = "FirstName"

Me.DataGridViewTextBoxColumn1.Name = "DataGridViewTextBoxColumn1"

'

'DataGridViewTextBoxColumn2

'

Me.DataGridViewTextBoxColumn2.DataPropertyName = "LastName"

Me.DataGridViewTextBoxColumn2.HeaderText = "LastName"

Me.DataGridViewTextBoxColumn2.Name = "DataGridViewTextBoxColumn2"

'

'DataGridViewTextBoxColumn3

'

Me.DataGridViewTextBoxColumn3.DataPropertyName = "Address"

Me.DataGridViewTextBoxColumn3.HeaderText = "Address"

Me.DataGridViewTextBoxColumn3.Name = "DataGridViewTextBoxColumn3"

'

'DataGridViewTextBoxColumn4

'

Me.DataGridViewTextBoxColumn4.DataPropertyName = "City"

Me.DataGridViewTextBoxColumn4.HeaderText = "City"

Me.DataGridViewTextBoxColumn4.Name = "DataGridViewTextBoxColumn4"

'

'DataGridViewTextBoxColumn5

'

Me.DataGridViewTextBoxColumn5.DataPropertyName = "State"

Me.DataGridViewTextBoxColumn5.HeaderText = "State"

Me.DataGridViewTextBoxColumn5.Name = "DataGridViewTextBoxColumn5"

'

'DataGridViewTextBoxColumn6

'

Me.DataGridViewTextBoxColumn6.DataPropertyName = "ZipCode"

Me.DataGridViewTextBoxColumn6.HeaderText = "ZipCode"

Me.DataGridViewTextBoxColumn6.Name = "DataGridViewTextBoxColumn6"

'

'DataGridViewTextBoxColumn7

'

Me.DataGridViewTextBoxColumn7.DataPropertyName = "TUID"

Me.DataGridViewTextBoxColumn7.HeaderText = "TUID"

Me.DataGridViewTextBoxColumn7.Name = "DataGridViewTextBoxColumn7"

'

'Form1

'

Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!, 13.0!)

Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font

Me.ClientSize = New System.Drawing.Size(578, 299)

Me.Controls.Add(Me.AddressesDataGridView)

Me.Controls.Add(Me.AddressesBindingNavigator)

Me.Name = "Form1"

Me.Text = "Form1"

CType(Me.MydataDataSet, System.ComponentModel.ISupportInitialize).EndInit()

CType(Me.AddressesBindingSource, System.ComponentModel.ISupportInitialize).EndInit()

CType(Me.AddressesBindingNavigator, System.ComponentModel.ISupportInitialize).EndInit()

Me.AddressesBindingNavigator.ResumeLayout(False)

Me.AddressesBindingNavigator.PerformLayout()

CType(Me.AddressesDataGridView, System.ComponentModel.ISupportInitialize).EndInit()

Me.ResumeLayout(False)

Me.PerformLayout()

End Sub

Friend WithEvents MydataDataSet As mydataDataSet

Friend WithEvents AddressesBindingSource As BindingSource

Friend WithEvents AddressesTableAdapter As mydataDataSetTableAdapters.AddressesTableAdapter

Friend WithEvents TableAdapterManager As mydataDataSetTableAdapters.TableAdapterManager

Friend WithEvents AddressesBindingNavigator As BindingNavigator

Friend WithEvents BindingNavigatorAddNewItem As ToolStripButton

Friend WithEvents BindingNavigatorCountItem As ToolStripLabel

Friend WithEvents BindingNavigatorDeleteItem As ToolStripButton

Friend WithEvents BindingNavigatorMoveFirstItem As ToolStripButton

Friend WithEvents BindingNavigatorMovePreviousItem As ToolStripButton

Friend WithEvents BindingNavigatorSeparator As ToolStripSeparator

Friend WithEvents BindingNavigatorPositionItem As ToolStripTextBox

Friend WithEvents BindingNavigatorSeparator1 As ToolStripSeparator

Friend WithEvents BindingNavigatorMoveNextItem As ToolStripButton

Friend WithEvents BindingNavigatorMoveLastItem As ToolStripButton

Friend WithEvents BindingNavigatorSeparator2 As ToolStripSeparator

Friend WithEvents AddressesBindingNavigatorSaveItem As ToolStripButton

Friend WithEvents AddressesDataGridView As DataGridView

Friend WithEvents DataGridViewTextBoxColumn1 As DataGridViewTextBoxColumn

Friend WithEvents DataGridViewTextBoxColumn2 As DataGridViewTextBoxColumn

Friend WithEvents DataGridViewTextBoxColumn3 As DataGridViewTextBoxColumn

Friend WithEvents DataGridViewTextBoxColumn4 As DataGridViewTextBoxColumn

Friend WithEvents DataGridViewTextBoxColumn5 As DataGridViewTextBoxColumn

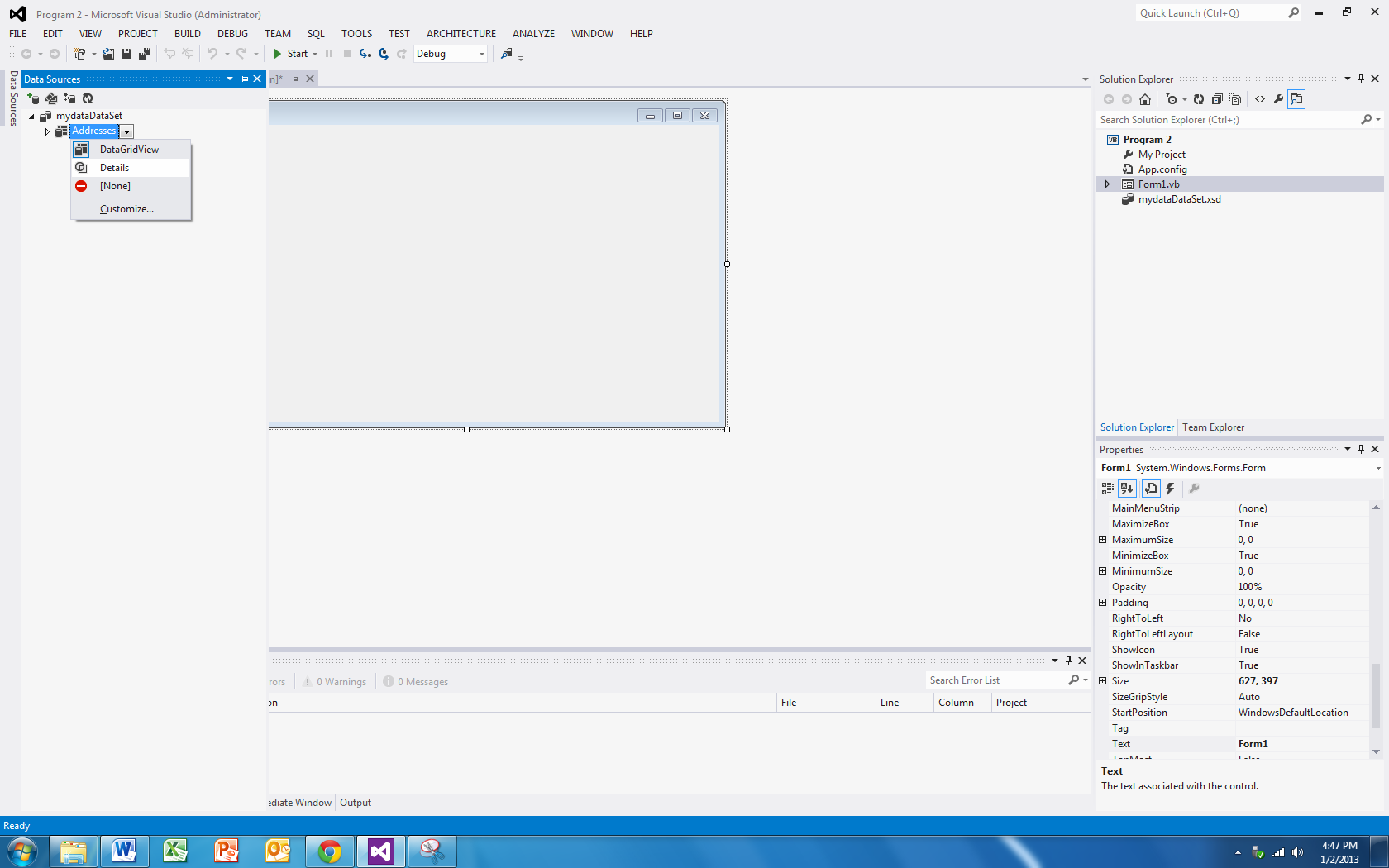
Friend WithEvents DataGridViewTextBoxColumn6 As DataGridViewTextBoxColumn

Friend WithEvents DataGridViewTextBoxColumn7 As DataGridViewTextBoxColumn

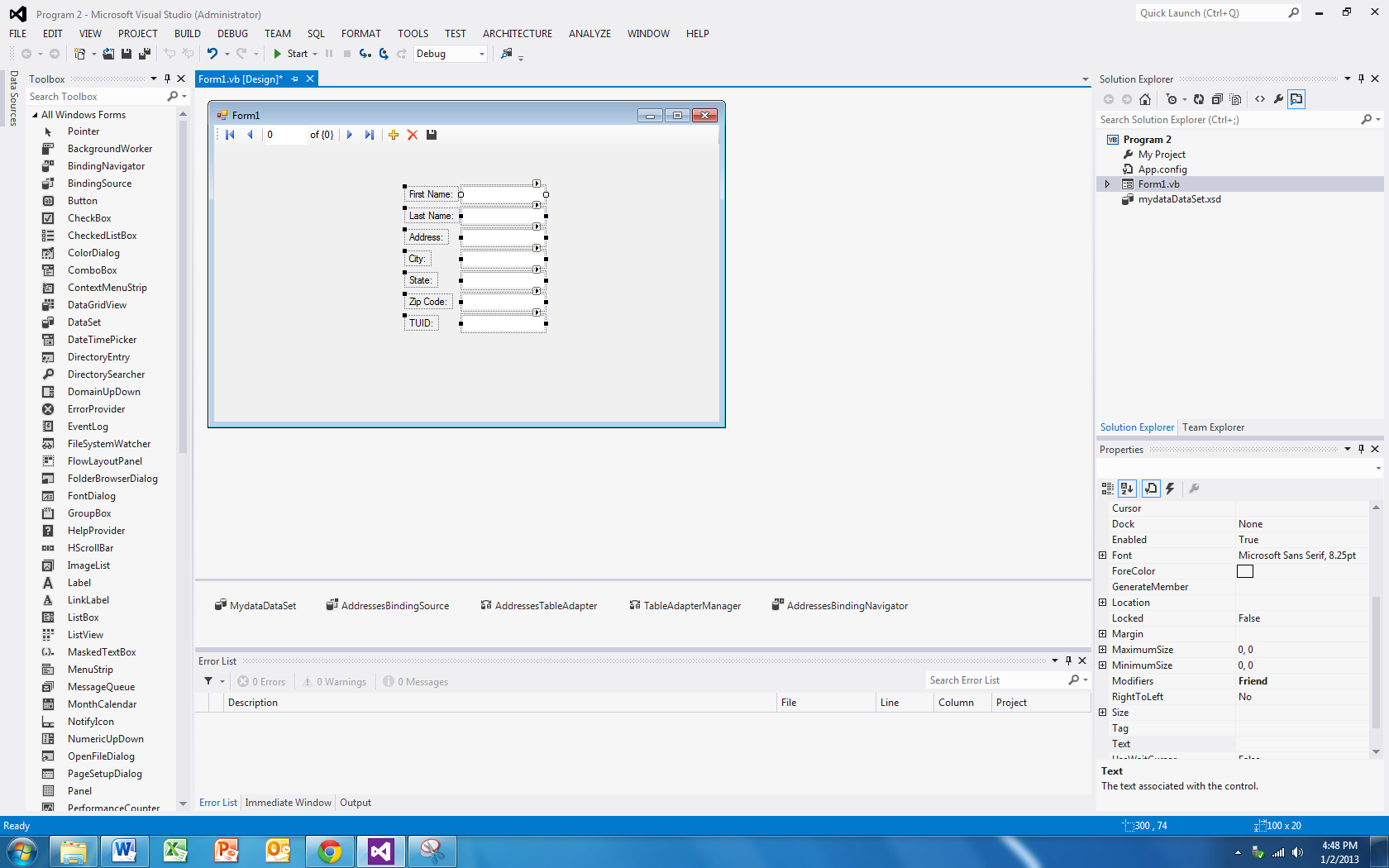
End Class

Yikes! There’s where a whole bunch of nastiness was hidden. But, if we wade through it, we still see that it’s not horribly bad to duplicate. Let me show you something else that’s pretty cool before continuing on.

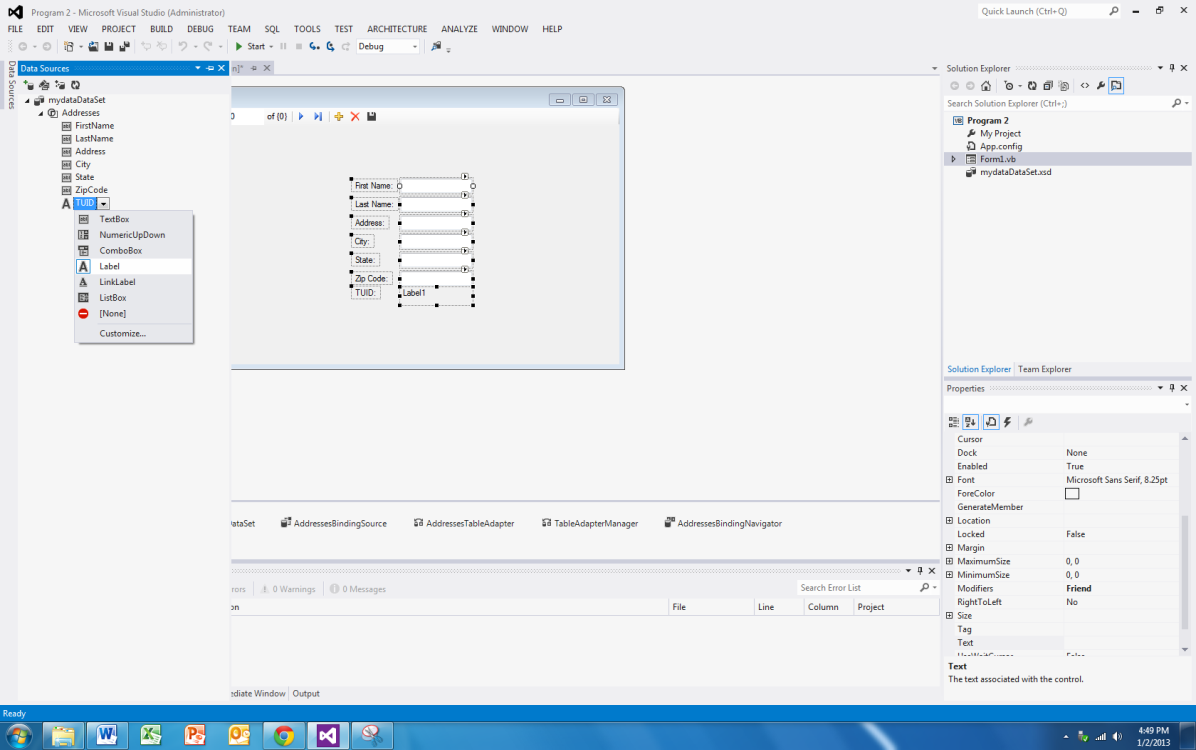
So I started a new project and went through the process of the data connection again (eventually it gets to a point where I can see the last one that I created). When everything finished up, I left clicked on Addresses and from the context menu, I chose Details:



Check out what happens when I drag that over to my new empty form:



That’s right – instead of a datagridview, I get each field in its own container (and back over on the Data Source, I can pick the container – here I changed the TUID to a label…)



If we fire the application up, we get this:



How slick is that? The code, well, it’s pretty much the same:

'Chapter 15 - Program 2

Public Class Form1

Private Sub AddressesBindingNavigatorSaveItem\_Click(sender As Object, e As EventArgs) Handles AddressesBindingNavigatorSaveItem.Click

Me.Validate()

Me.AddressesBindingSource.EndEdit()

Me.TableAdapterManager.UpdateAll(Me.MydataDataSet)

End Sub

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

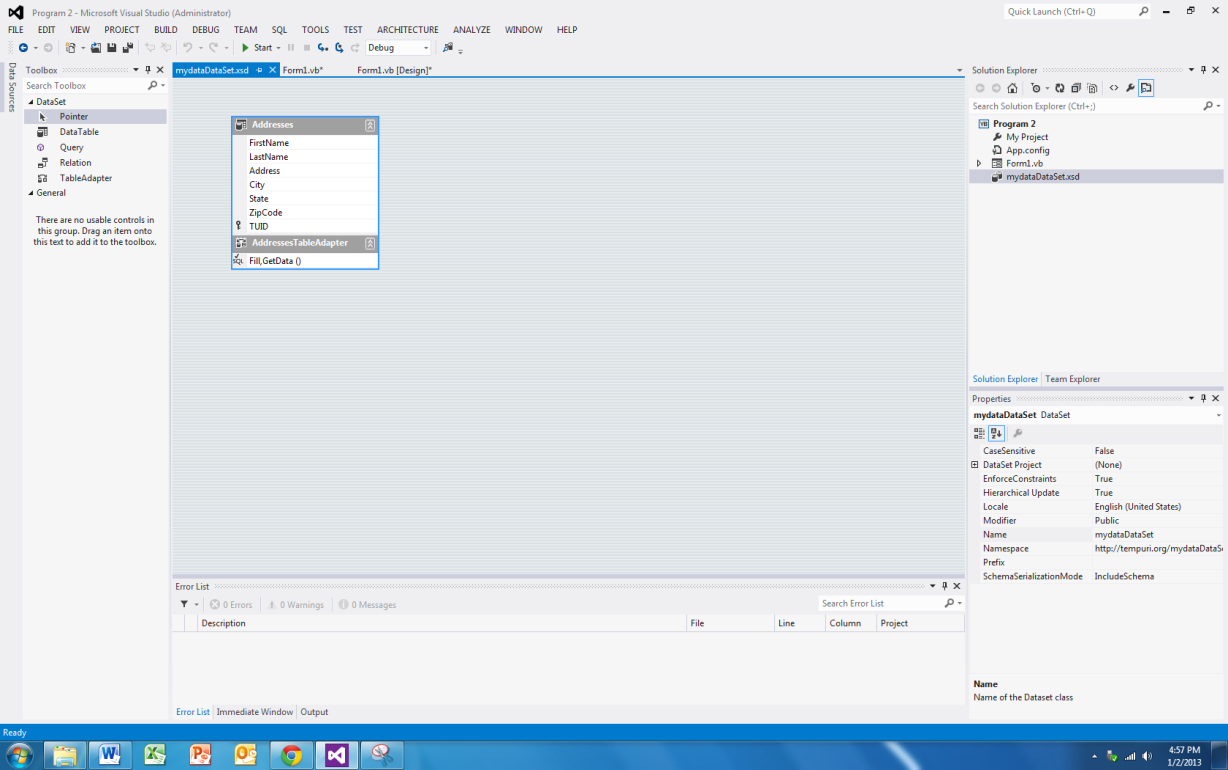
'TODO: This line of code loads data into the 'MydataDataSet.Addresses' table. You can move, or remove it, as needed.

Me.AddressesTableAdapter.Fill(Me.MydataDataSet.Addresses)

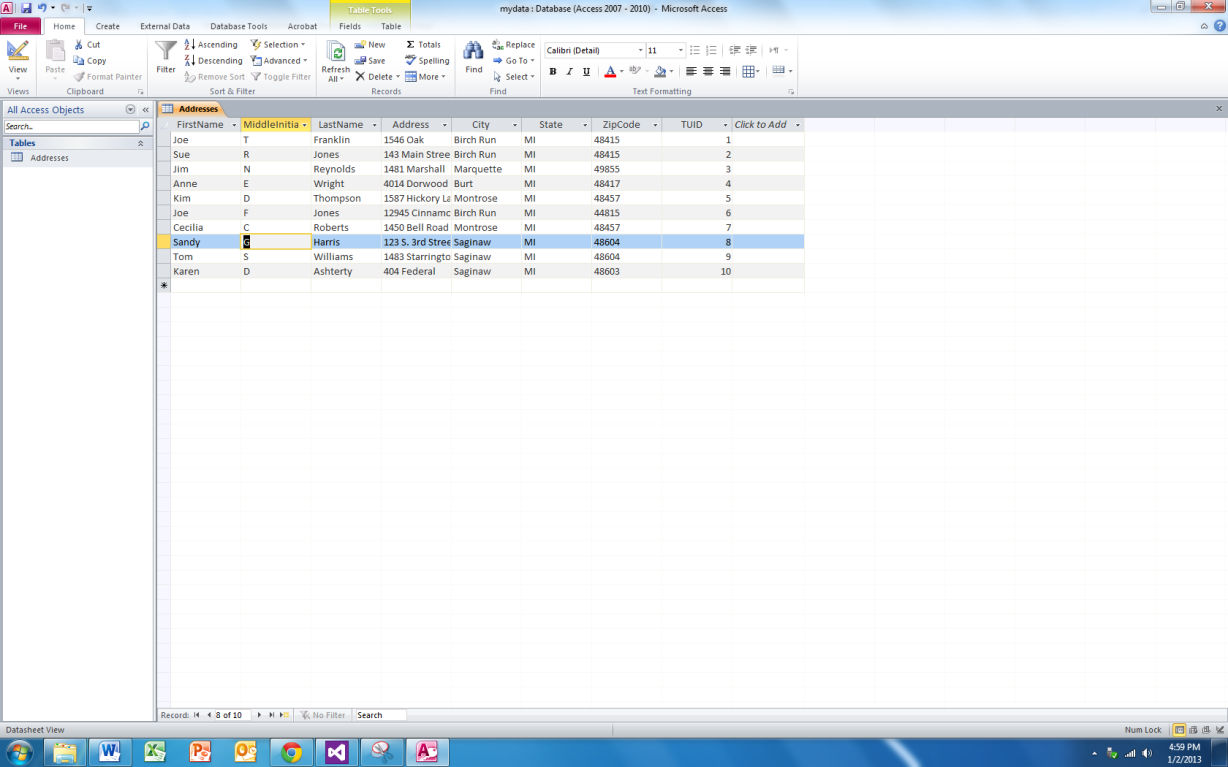
End Sub

End Class

One more piece to the mystery? Did you notice that a new file called mydataDataSet.xsd shows up in the project? Click on it to open it, and voila, Visual Basic built an XML Schema Definition showing the makeup of the data source and how the adapter fills the dataset from it:

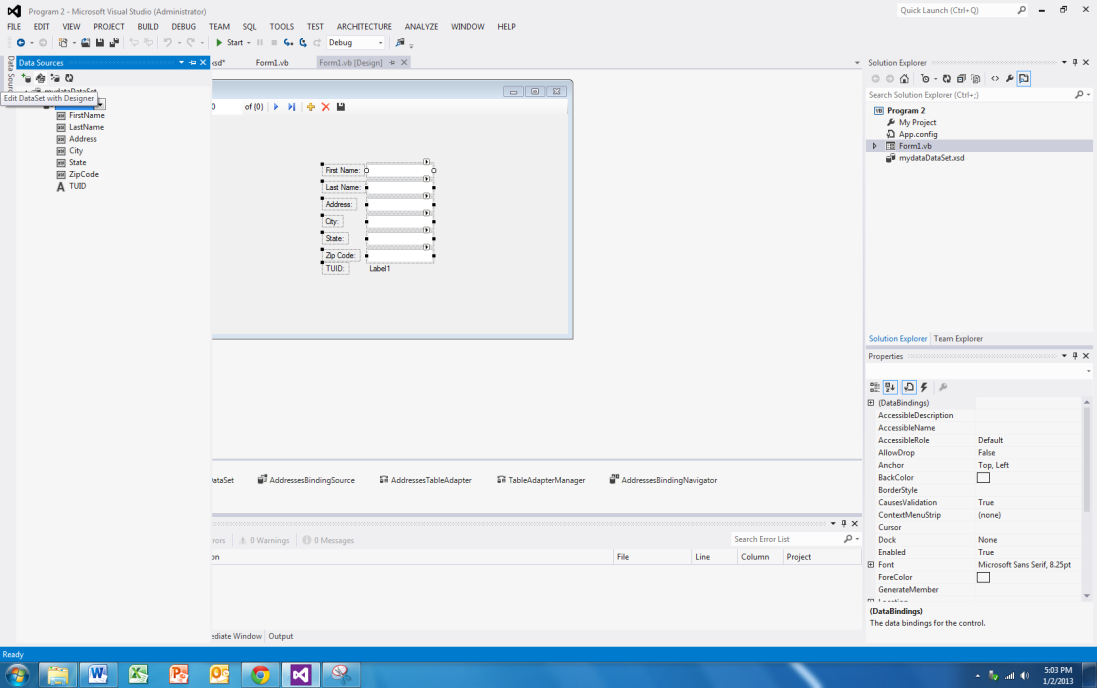


So, now the question you should ask, well all of this stuff is fine and good, but what happens if I change the underlying database. Let’s temporarily add a Middle Initial column to our database:

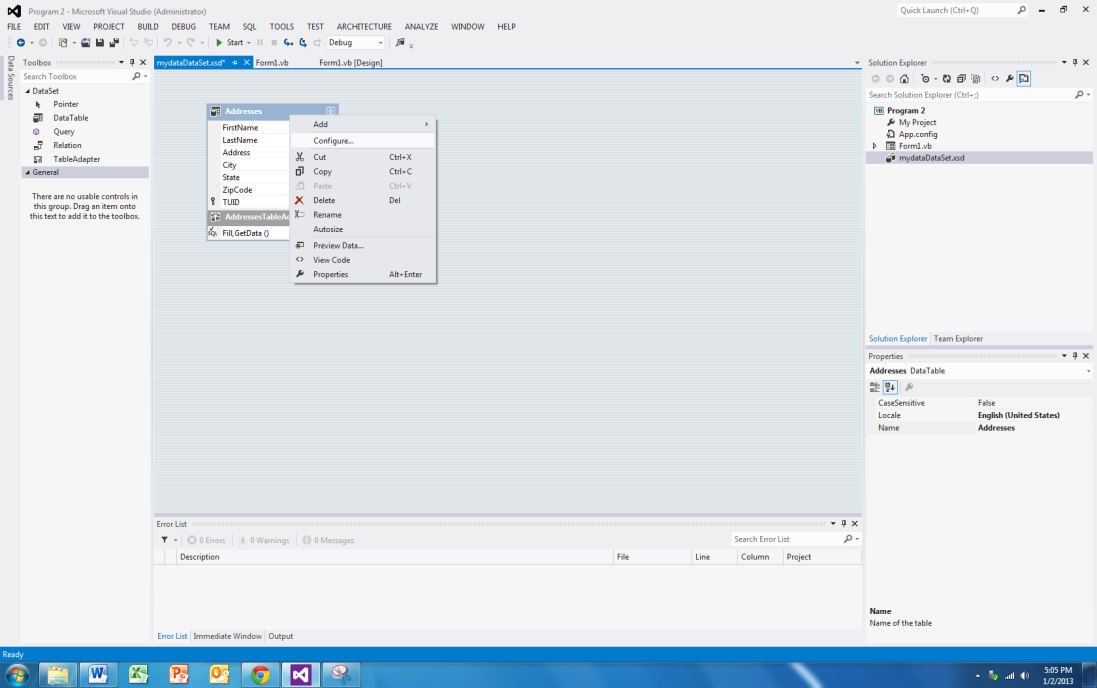


If I run my application, I don’t see anything new and that’s because all of the backing information is still looking at the “old” version. Basically, we’re hosed…we will need to do whatever manipulations to bring the new field in – that’s problem number one with the magic: it works great on new stuff initially; make a change and you get heartache.

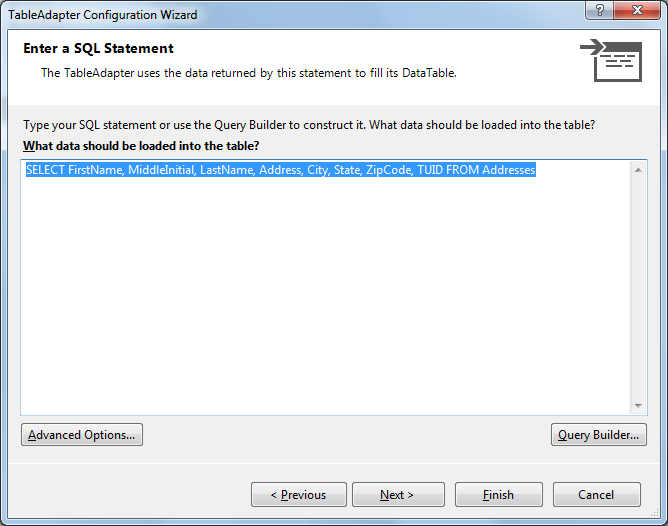
So, if I do want to continue down this path, I would go back to my Data Sources window and click on the Edit Dataset with Designer:



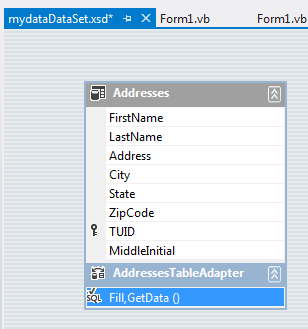
This brings up the xsd page that we previously saw:



If I right-click and select Configure, I can change the SQL to bring back the new field. I can also Preview my Data.

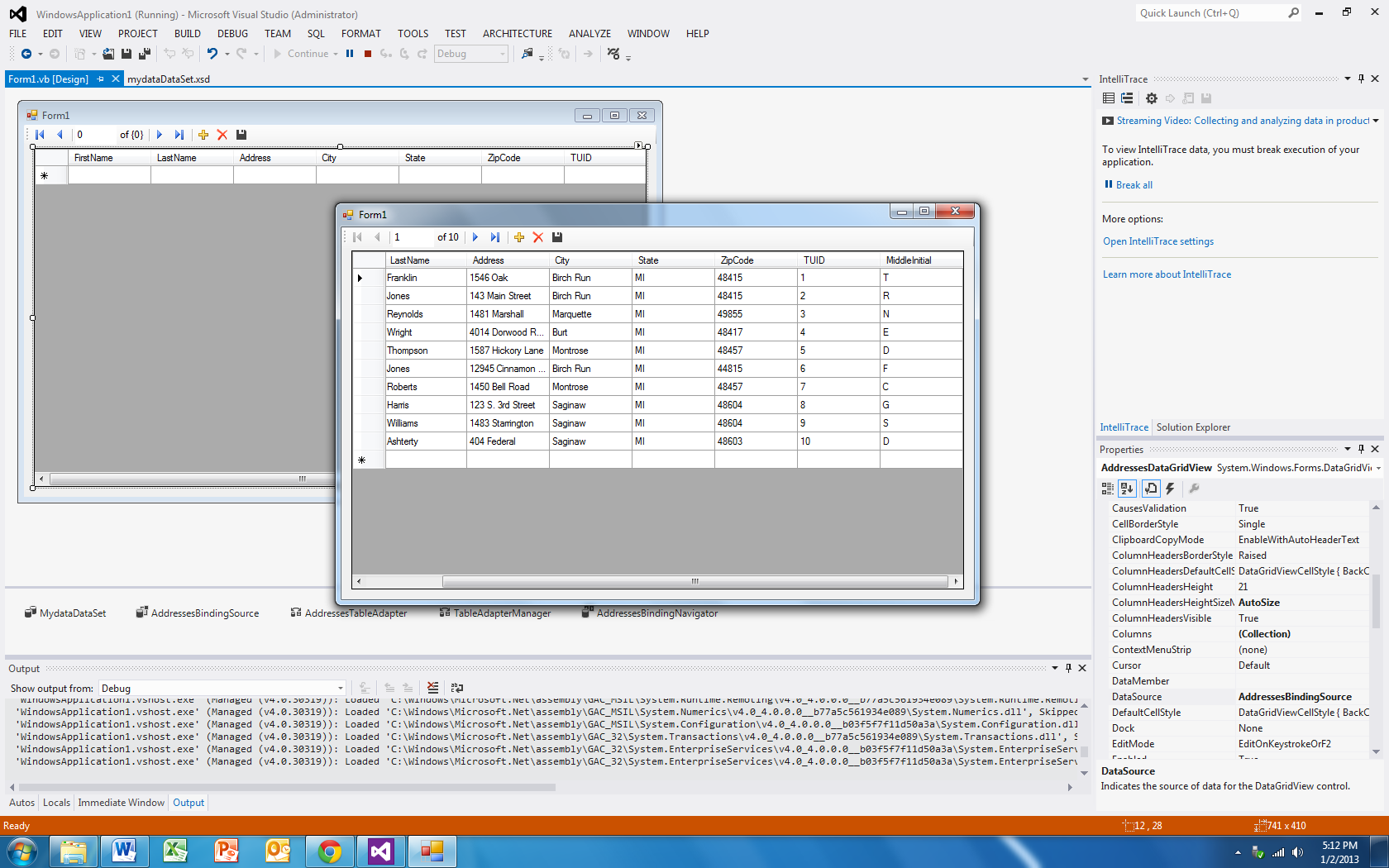


Clicking Next a few times (or just Finish from here), will add the new column to my dataset:



Now, I would have to go in and add the new fields to the form and wire them up to the bindingnavigator – Yuck!

At least if I would have stuck with the datagridview, the column would have shown up for me if I refreshed the datagridview’s Data Source to the updated BindingSource.



Therein lies problem two with this method: you’ve got no real understanding of where and what VB has done for you/to you. While it wasn’t too bad to mess with the datagridview to get it to show the new database, the individual field version isn’t that nice!

The only other issue that you need to be aware of is with regards to making changes to the data displayed in the DataGridView. You need to be careful in thinking the data that you are seeing is a "live" copy as it is not. You have the ability to make changes to the data that is in the DataGridView, but these changes are not automatically written back out to the database. In other words, if you make some changes, exit the application and then start the application back up, you will see that all of your changes disappeared and the database looks exactly like it did before you made them. If you want to save the changes out to the database there is additional work that you have to do.

# Working with Databases Through VB Code

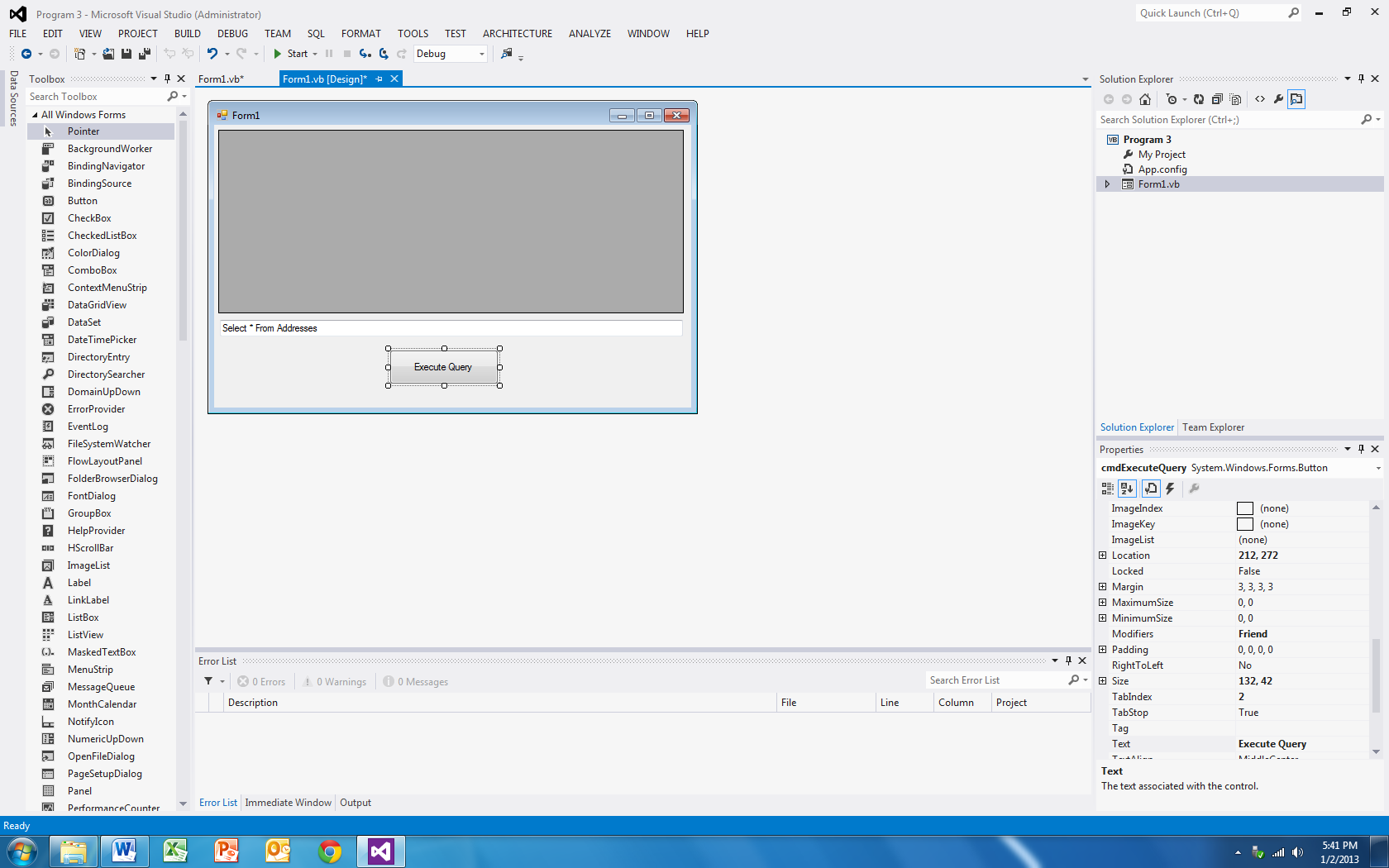
So we’ve covered a lot of territory, although in the long run, not all of it will get us to where we want to be. Hopefully, with the negative points I pointed out above, I’ve now convinced you to not do databases through draggy-droppy since you really can’t easily see what VB’s done under the hood. Everytime we make a change to our underlying data source, we’ve got rework ranging from easy (datagridview refresh of the data source) to hard (controls that are bound) that we have to add ourselves. While it might be more work, it’s better in my opinion to understand how things operate so that we can be in control of the changes that are there.

Let’s take a look at the important ADO objects we need to be able to work with:

* Connection objects – again, this directs the communication between your program and the data source.
* DataAdapter objects – this is what makes the connection between the data source and your program’s data sets. Think of this as the wire between the database and your program.
* DataSet objects – like mini databases, made up of DataTables and DataRelations – we can fill them from a database or completely build them up in memory.
* Command objects – this is how we execute our various SQL statements over the connection object.
* CommandBuilder objects – if our database tables are set up correctly, e.g. unique primary keys and what not, the CommandBuilder can actually write the SELECT, UPDATE, INSERT and DELETE commands for us.
* DataReader objects – sometimes we just care about retrieving results quickly and efficiently – no changing, no backward reading; just pass through and print – that’s the time when you want to use a DataReader.

So now that we know who the team players are, let’s build up an application from scratch that connects to our database, fills the data set, hooks the data set to the datagridview and displays the results. Oh yeah, let’s also throw in a textbox that allows us to selectively filter records. The datagridview is called dgvResults, the textbox is called txtQuery and the command button is called cmdExecuteQuery.

Here’s the form:



Here’s the code:

'Chapter 15 - Program 3

'Make sure that we bring in the OleDb namespace to access the database pieces

Imports System.Data.OleDb

Public Class Form1

'Here's where we will define the database; where it lives and how to access it:

Public Const gstrDBName As String = "C:\DB\mydata.accdb" 'Office 2010 and newer format

Public Const gstrConnString As String =

"Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" & gstrDBName &

";Persist Security Info=False"

Private Sub cmdExecuteQuery\_Click(sender As Object, e As EventArgs) Handles cmdExecuteQuery.Click

'Here's the connection to the database object

Dim DBConn As OleDbConnection

'We'll build up our query in SQL

Dim strSQLCmd As String

'We place the SQL query in a command object

Dim DBCommand As New OleDbCommand

'This is where our results will land

Dim myDataset As New DataSet

'We will use this to fill our data set with the DB

Dim DBAdapter As New OleDbDataAdapter

'Create a new connection based on the connection string and open it

DBConn = New OleDbConnection(gstrConnString)

DBConn.Open()

'Get the current text from the textbox and that'll be our SQL query

strSQLCmd = txtQuery.Text

'Load the SQL query into the command object

DBCommand.CommandText = strSQLCmd

'Hook the data adapter up to run the SQL command on our connection

DBAdapter = New OleDbDataAdapter(strSQLCmd, DBConn)

'Fill the dataset

DBAdapter.Fill(myDataset, "Addresses")

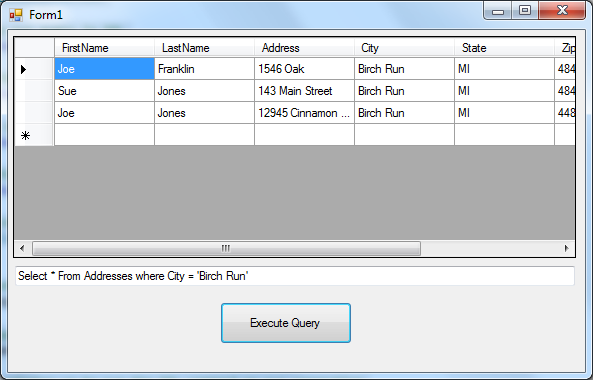
'Now hook the datagridview up to the Addresses table in the dataset

dgvResults.DataSource = myDataset.Tables("Addresses")

End Sub

End Class

Let’s take a look at our application in action:



## Using the DataReader

The next example is going to show you how to work with the DataReader. Now even though we didn’t do anything other than display the contents in the database, because we used DataSets and what not, we had the ability to change things if we chose. That’s costly – what if all we really wanted to do was to iterate over each record for printing? Here’s where the data reader comes in.

'Chapter 15 - Program 4

Imports System.Data.OleDb

'Databases don't have to be Forms based!

Module Module1

Const gstrDBName As String = "C:\mydata.accdb" 'Office 2010

Const gstrConnString As String =

"Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" &

gstrDBName & ";Persist Security Info=False" 'Office 2010

Sub Main()

'These variable serve the same purpose as in the preceding example

Dim DBConn As OleDbConnection

Dim strSQLCmd As String

Dim DBCommand As New OleDbCommand

'Create a data reader object

Dim myDataReader As OleDbDataReader

'Create a new connection based on the connection string and open it

DBConn = New OleDbConnection(gstrConnString)

DBConn.Open()

'Set the command up with its command text and connection

strSQLCmd = "Select \* From Addresses"

DBCommand.CommandText = strSQLCmd

DBCommand.Connection = DBConn

'Execute the reader

myDataReader = DBCommand.ExecuteReader

'While there's data, read it

While myDataReader.Read()

'and print it -- notice the use of numeric indexes or field names –

'either is okay

Debug.WriteLine("TUID: {0} Name: {1} {2} Address: {3} {4}," &

" {5} {6}", myDataReader(6), myDataReader("FirstName"),

myDataReader("LastName"), myDataReader(2),

myDataReader("City"), myDataReader("State"),

myDataReader("ZipCode"))

End While

'Close the reader up

myDataReader.Close()

End Sub

End Module

Here’s the execution:

TUID: 1 Name: Joe Franklin Address: 1546 Oak Birch Run, MI 48415

TUID: 2 Name: Sue Jones Address: 143 Main Street Birch Run, MI 48415

TUID: 3 Name: Jim Reynolds Address: 1481 Marshall Marquette, MI 49855

TUID: 4 Name: Anne Wright Address: 4014 Dorwood Road Burt, MI 48417

TUID: 5 Name: Kim Thompson Address: 1587 Hickory Lane Montrose, MI 48457

TUID: 6 Name: Joe Jones Address: 12945 Cinnamon Steet Birch Run, MI 44815

TUID: 7 Name: Cecilia Roberts Address: 1450 Bell Road Montrose, MI 48457

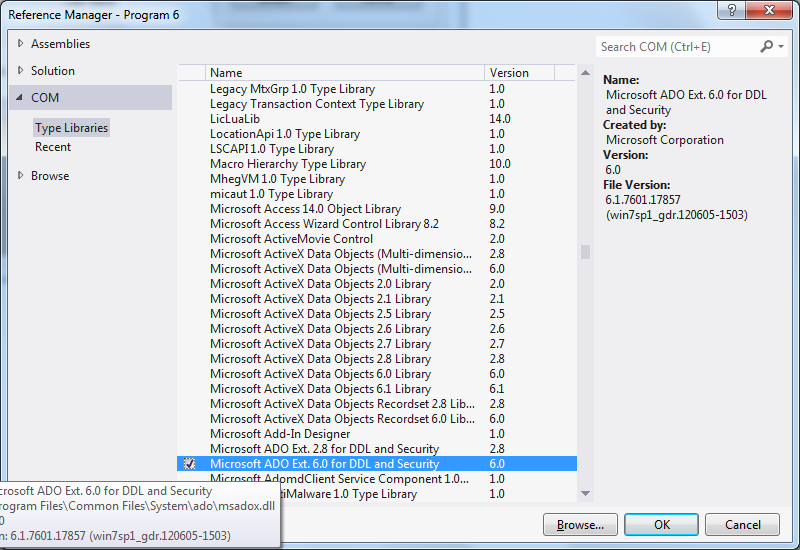
TUID: 8 Name: Sandy Harris Address: 123 S. 3rd Street Saginaw, MI 48604

TUID: 9 Name: Tom Williams Address: 1483 Starrington Saginaw, MI 48604

TUID: 10 Name: Karen Ashterty Address: 404 Federal Saginaw, MI 48603

## Driving Databases Purely Through VB Code

I just want you to realize that you can handle everything through code in .NET. There is one caveat to this example and that is that you must include a COM reference to Microsoft ADO Ext 6.0 (or whatever the latest version is that you have on your system) for DDL and Security in order to make this project work



The reason that we have to add that particular reference is due to the changes that Microsoft made in ADO.NET. In previous versions of VB, ADO directly had the ability to create new Jet Engine databases on the fly. Unfortunately that capability has been removed from ADO.NET. This really isn't a worry for us though, since we can utilize the ADOX Catalog function to perform the same work. Also, it’s worthwhile to point out that the ADOX really is 32 bit only, so on a 64 bit system make sure that you set the Project Properties🡪Compile🡪Target CPU from Any CPU to x86.

This example is the last one that we will look at in the database chapter. It rolls everything that we've talked about up into one application. The purpose of this application is to simulate a college registration system. Three separate tables will be used in this application: Students, Courses and Registration. All three tables reside in the same database.

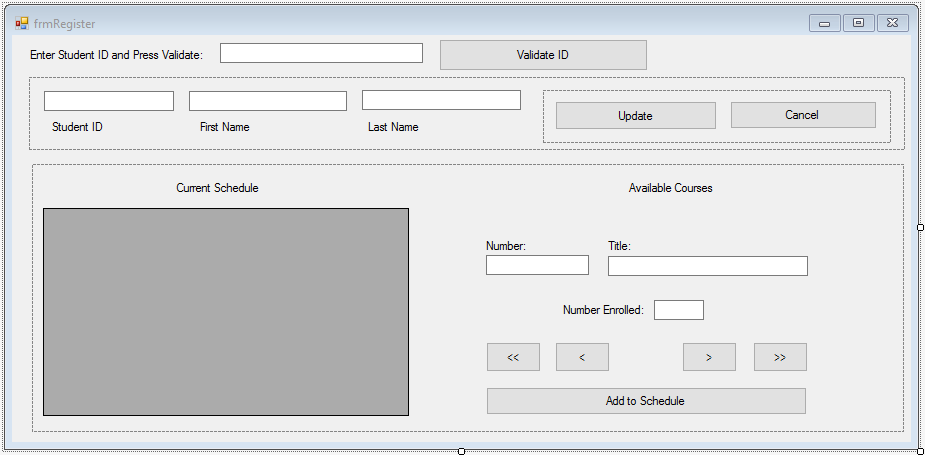
The first table, Students, is responsible for holding student information, which in this application is the Student's ID Number (in the table this field is known as SID), First Name (FirstName) and Last Name (LastName). The second table, Courses is a listing of all courses that the college offers. There is a Course ID Number (CID), the Course Title (CourseTitle) and the number of students enrolled in the course (NumberEnrolled). The last table, Registration, simply contains the SID of students that are signed up for CID courses.

Let's spend a moment and discuss how the registration should work. The first thing that should happen is a student should type in his or her Student ID number. If the ID number is located in the Students table, the application can proceed. If the number is not located, the student should be provided an opportunity to enter his or her personal information into the Students database. Assuming that the student types in his or her data, the application will proceed exactly in the same manner is if the student's ID number had immediately been located.

Once the Student ID number has been verified, the application system should then provide the student with ability to add courses to his or her schedule. In addition the application needs to display the student's current course schedule if the student has previously registered for courses. The application doesn't perform any error trapping such as if a student attempts to register for a course that he or she has already registered for. It would not be difficult to add this component, but the example is already complex enough.

If you want to have a better understanding of how the application works before you start examining its code, you should either load and run the application (Chapter 15 – Program 5). You could also take a look at the walkthrough of the application which is located just after the source code.

The next screenshot shows the main application form. Notice that there are three panels in place. There is a panel in the middle that contains the student's personal information and a panel nested inside of that panel which contains Update and Cancel buttons. The third panel is located on the lower portion of the application and holds the student's current schedule in a DataGridView control as well as the course listing information.



The following table lists the names of the various controls on the form from top to bottom and left to right.

|  |  |
| --- | --- |
| Control Name | Description |
| txtStudentID | Textbox where the student enters his or her ID number |
| cmdValidateID | Command button used to indicate that the student has entered his or her ID and wants the application to attempt to validate it |
| pnlStudentInfo | Panel containing the student information as well as the second panel containing the Update and Cancel command buttons |
| txtID | Textbox containing the student's identification number |
| txtFirstName | Textbox containing the student's first name |
| txtLastName | Textbox containing the student's last name |
| pnlStudentUpdate | Panel containing the Update and Cancel command buttons |
| cmdUpdateStudentInfo | Command button used to add a new student's information to the Students table |
| cmdStudentUpdateCancel | Command button used to cancel adding a new student's information to the Students table |
| pnlRegistration | Panel containing the current student schedule as well as all of the course related controls |
| dgvSchedule | DataGridView containing the student's current schedule |
| txtCourseNumber | Textbox containing the current Courses record's course number |
| txtCourseTitle | Textbox containing the current Courses record's course title |
| txtNumberEnrolled | Textbox containing the current number of students enrolled in the current Courses record's course |
| cmdFirst | The "<|" command button used to move to the first record in the Courses table |
| cmdPrevious | The "<" command button used to move to the previous record in the Courses table |
| cmdNext | The ">" command button used to move to the next record in the Courses table |
| cmdLast | The "|>" command button used to move to the last record in the Courses table |
| cmdAddToSchedule | The command button used to add the current course to the student's schedule |

The real work is all in the application's code, which consists of a code module and the code behind the application's form. If you spend time combing over this code example you will get an idea of what is involved in building a robust database application. Here's the code from the Module1.vb module:

'Chapter 15 - Program 5

'Database Example

'Remember to add a reference to Microsoft ADOX Ext For DDL and Security

'Watch the path for the database -- Windows doesn't like creating things

'directly on the root

'Finally, make sure that you set the Build type from Any CPU to x86 underneath the

'Project Properties-->Compile-->Target CPU since the ADOX doesn't work correctly

'under 64 bit

'We will be working with items from both of the following namespaces

Imports System.Data.OleDb

Imports System.IO

Module Module1

Sub Main()

Dim strConn As String 'used to tell type of database & path

Dim strDBName As String = "c:\DB\new.mdb" 'path of database

Dim myfrmRegister As New frmRegister() 'the main form

'Pointer to the database we will use

strConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & strDBName

'If the database doesn't exist, create it

If Not (File.Exists(strDBName)) Then

CreateDatabase(strConn)

End If

'Make sure all tables are clean each time we run this

CleanOutStudentsTable(strConn)

CleanOutCoursesTable(strConn)

CleanOutRegistrationTable(strConn)

'Put some data into the tables

PopulateStudentsTable(strConn) '1 student record

PopulateCoursesTable(strConn) '6 courses

PopulateRegistrationTable(strConn) 'the 1 student is signed up for 2

'courses

myfrmRegister.ShowDialog()

End Sub

Sub CreateDatabase(ByVal strConn As String)

'Let's build an Access database

Dim DBCat As New ADOX.Catalog()

Dim DBConn As OleDbConnection

Dim DBCmd As OleDbCommand = New OleDbCommand()

Try

DBCat.Create(strConn)

MessageBox.Show("Created database")

Catch Ex As Exception

MessageBox.Show("Database already exists")

End Try

'Build the Tables

DBConn = New OleDbConnection(strConn)

DBConn.Open()

'Build the Student Table

DBCmd.CommandText = "CREATE TABLE Students (" &

"SID varchar(6), " &

"FirstName varchar(50), " &

"LastName varchar(50))"

DBCmd.Connection = DBConn

Try

DBCmd.ExecuteNonQuery()

MessageBox.Show("Created Students Table")

Catch Ex As Exception

MessageBox.Show("Clients Table Already Exists")

End Try

'Build the Courses Table

DBCmd.CommandText = "CREATE TABLE Courses (" &

"CID varchar(6), " &

"CourseTitle varchar(50), " &

"NumberEnrolled varchar(50))"

DBCmd.Connection = DBConn

Try

DBCmd.ExecuteNonQuery()

MessageBox.Show("Created Courses Table")

Catch Ex As Exception

MessageBox.Show("Courses Table Already Exists")

End Try

'Build the Registration Table

DBCmd.CommandText = "CREATE TABLE Registration (" &

"SID varchar(6), " &

"CID varchar(6))"

DBCmd.Connection = DBConn

Try

DBCmd.ExecuteNonQuery()

MessageBox.Show("Created Registration Table")

Catch Ex As Exception

MessageBox.Show("Registration Table Already Exists")

End Try

DBConn.Close()

End Sub

Sub CleanOutStudentsTable(ByVal strConn As String)

Dim DBConn As OleDbConnection

Dim DBCmd As OleDbCommand = New OleDbCommand()

'Now try to open up a connection to the database

DBConn = New OleDbConnection(strConn)

DBConn.Open()

'Use SQL to zap the contents of the table

DBCmd.CommandText = "DELETE \* FROM Students"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

MessageBox.Show("Deleted Everything In Students")

DBConn.Close()

End Sub

Sub CleanOutCoursesTable(ByVal strConn As String)

Dim DBConn As OleDbConnection

Dim DBCmd As OleDbCommand = New OleDbCommand()

'Now try to open up a connection to the database

DBConn = New OleDbConnection(strConn)

DBConn.Open()

'Use SQL to zap the contents of the table

DBCmd.CommandText = "DELETE \* FROM Courses"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

MessageBox.Show("Deleted Everything In Courses")

DBConn.Close()

End Sub

Sub CleanOutRegistrationTable(ByVal strConn As String)

Dim DBConn As OleDbConnection

Dim DBCmd As OleDbCommand = New OleDbCommand()

'Now try to open up a connection to the database

DBConn = New OleDbConnection(strConn)

DBConn.Open()

'Use SQL to zap the contents of the table

DBCmd.CommandText = "DELETE \* FROM Registration"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

MessageBox.Show("Deleted Everything In Registration")

DBConn.Close()

End Sub

Sub PopulateStudentsTable(ByVal strConn As String)

Dim DBConn As OleDbConnection

Dim DBCmd As OleDbCommand = New OleDbCommand()

'Now try to open up a connection to the database

DBConn = New OleDbConnection(strConn)

DBConn.Open()

'Add a student using SQL

DBCmd.CommandText = "INSERT INTO Students (SID, FirstName, LastName) " &

"VALUES ('123','Scott','James')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

DBConn.Close()

MessageBox.Show("Added a Student To Students Table")

End Sub

Sub PopulateCoursesTable(ByVal strConn As String)

Dim DBConn As OleDbConnection

Dim DBCmd As OleDbCommand = New OleDbCommand()

'Now try to open up a connection to the database

DBConn = New OleDbConnection(strConn)

DBConn.Open()

'Add courses using SQL

DBCmd.CommandText = "INSERT INTO Courses (CID, CourseTitle, NumberEnrolled) " &

"VALUES ('CS 116','Computer Programming I','0')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

DBCmd.CommandText = "INSERT INTO Courses (CID, CourseTitle, NumberEnrolled) " &

"VALUES ('CS 216','Computer Programming II','0')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

DBCmd.CommandText = "INSERT INTO Courses (CID, CourseTitle, NumberEnrolled) " &

"VALUES ('CIS311','Visual Basic Programming','0')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

DBCmd.CommandText = "INSERT INTO Courses (CID, CourseTitle, NumberEnrolled) " &

"VALUES ('CIS422','Systems Analysis & Design','1')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

DBCmd.CommandText = "INSERT INTO Courses (CID, CourseTitle, NumberEnrolled) " &

"VALUES ('CIS424','Systems Design & Implementation','0')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

DBCmd.CommandText = "INSERT INTO Courses (CID, CourseTitle, NumberEnrolled) " &

"VALUES ('CIS486','Enterprise Database Systems','1')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

DBConn.Close()

MessageBox.Show("Added Courses To Courses Table")

End Sub

Sub PopulateRegistrationTable(ByVal strConn As String)

Dim DBConn As OleDbConnection

Dim DBCmd As OleDbCommand = New OleDbCommand()

'Now try to open up a connection to the database

DBConn = New OleDbConnection(strConn)

DBConn.Open()

'Add student registration using SQL

DBCmd.CommandText = "INSERT INTO Registration (SID, CID) " &

"VALUES ('123','CIS422')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

DBCmd.CommandText = "INSERT INTO Registration (SID, CID) " &

"VALUES ('123','CIS486')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery()

DBConn.Close()

MessageBox.Show("Added Registrations to Registration Table")

End Sub

End Module

Now we need to take a look at the code behind the form:

'Chapter 15 - Program 5

Imports System.Data.OleDb

Public Class frmRegister

'Create a dataset to point to each table -- do it here so that we don't

'have to keep passing things around

Dim dsStudents As New DataSet

Dim dsCourses As New DataSet

Dim dsRegistration As New DataSet

'Here's the connection string

Dim strConn As String =

"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\DB\new.mdb"

'We'll also create an OleDBConnection object since we will execute some

'straight SQL rather than relying on the DBAdapters...

Dim myConn As New OleDbConnection(strConn)

'Likewise create three data adapters so we don't mess stuff up

Dim DBAdaptStudents As OleDbDataAdapter

Dim DBAdaptCourses As OleDbDataAdapter

Dim DBAdaptRegistration As OleDbDataAdapter

Private Sub frmRegister\_Load(sender As Object, e As EventArgs) Handles Me.Load

Dim strSQLCmd As String

'All panels should start out hidden

pnlRegistration.Visible = False

pnlStudentInfo.Visible = False

pnlStudentUpdate.Visible = False

'Load up all courses since they will never change while program runs

strSQLCmd = "Select \* From Courses"

DBAdaptCourses = New OleDbDataAdapter(strSQLCmd, strConn)

DBAdaptCourses.Fill(dsCourses, "Courses")

'Set bindings on the course related fields since we now have some data in

'context and can understand the schema of the table

txtCourseNumber.DataBindings.Add(New Binding("Text", dsCourses,

"Courses.CID"))

txtCourseTitle.DataBindings.Add(New Binding("Text", dsCourses,

"Courses.CourseTitle"))

txtNumberEnrolled.DataBindings.Add(New Binding("Text", dsCourses,

"Courses.NumberEnrolled"))

End Sub

Private Sub cmdValidateID\_Click(sender As Object, e As EventArgs) Handles

cmdValidateID.Click

'This is called when we put in a student number

Dim strSQLCmd As String

Dim cmdBuilder As OleDbCommandBuilder 'This is cool -- look below...

'Make sure other panels aren't showing yet

pnlRegistration.Visible = False

pnlStudentUpdate.Visible = False

pnlStudentInfo.Visible = False

'Set up the data adapter for Students...

strSQLCmd = "Select \* From Students Where SID = '" &

Trim(txtStudentID.Text) & "'"

DBAdaptStudents = New OleDbDataAdapter(strSQLCmd, strConn)

'We must include how an "AddNew" to the Bound Controls will handle an

'Insert. Forunately, VB.NET provides a cmdBuilder object that will create

'the right statements...you can also get the DeleteCommand and

'UpdateCommands...

cmdBuilder = New OleDbCommandBuilder(DBAdaptStudents)

DBAdaptStudents.InsertCommand = cmdBuilder.GetInsertCommand

'Fill the dataset

dsStudents.Clear()

DBAdaptStudents.Fill(dsStudents, "Students")

'See if we've already set bindings on the student related controls...

If txtID.DataBindings.Count = 0 Then

'We need to set the bindings for the student information

txtID.DataBindings.Add(New Binding("Text", dsStudents,

"Students.SID"))

txtFirstName.DataBindings.Add(New Binding("Text", dsStudents,

"Students.FirstName"))

txtLastName.DataBindings.Add(New Binding("Text", dsStudents,

"Students.LastName"))

End If

'Attempt to get this student's Schedule

strSQLCmd = "Select \* From Registration Where SID = '" &

Trim(txtStudentID.Text) & "'"

DBAdaptRegistration = New OleDbDataAdapter(strSQLCmd, strConn)

dsRegistration.Clear()

DBAdaptRegistration.Fill(dsRegistration, "Registration")

'We couldn't bind the datagridview until we had a dataset, so do it now

dgvSchedule.DataSource = dsRegistration.Tables("Registration")

'Show student information panel

pnlStudentInfo.Visible = True

'Check to see if student exists

If dsStudents.Tables("Students").Rows.Count <= 0 Then

'Student doesn't exist...

MessageBox.Show("Enter info into the boxes and press Update to " &

"add a new student or Cancel to go back to Validation")

'Clear out the current edits

BindingContext(dsStudents, "Students").EndCurrentEdit()

'Add a new record to the recordset

BindingContext(dsStudents, "Students").AddNew()

'Show panel with push buttons and enable textboxes that

'we want user to enter information into

pnlStudentUpdate.Visible = True

txtFirstName.Enabled = True

txtLastName.Enabled = True

txtID.Text = txtStudentID.Text

Else

'Student already exists so show registration panel

pnlRegistration.Visible = True

End If

End Sub

Private Sub cmdUpdateStudentInfo\_Click(sender As Object, e As EventArgs)

Handles cmdUpdateStudentInfo.Click

'If we got here, then a new student number was entered and the user

'clicked on the Update button to indicate that they wanted the

'student information they entered to be placed into the database.

'Notice that this routine is using the AddNew methods of the

'dataset and not any direct SQL manipulation...this is one way

'we can modify what's in the database

'Stop any current edits.

BindingContext(dsStudents, "Students").EndCurrentEdit()

'Update the database

myConn.Open()

DBAdaptStudents.Update(dsStudents, "Students")

myConn.Close()

'Update the dataset to correspond with database.

dsStudents.AcceptChanges()

'Hide the panel and reset the textboxes

txtFirstName.Enabled = False

txtLastName.Enabled = False

pnlStudentUpdate.Visible = False

'Show the registration panel since the student is now valid

pnlRegistration.Visible = True

End Sub

Private Sub cmdStudentUpdateCancel\_Click(sender As Object, e As EventArgs)

Handles cmdStudentUpdateCancel.Click

'If we got here, the user typed in a new student number and then

'decided that they did not want it entered and hit the Cancel button

'Throw away the current edit

BindingContext(dsStudents, "Students").CancelCurrentEdit()

'Hide the panel and reset the textboxes

txtFirstName.Enabled = False

txtLastName.Enabled = False

pnlStudentInfo.Visible = False

pnlStudentUpdate.Visible = False

End Sub

Private Sub cmdFirst\_Click(sender As Object, e As EventArgs) Handles

cmdFirst.Click

'Called when we click the move to first button << in Course listing

BindingContext(dsCourses, "Courses").Position = 0

End Sub

Private Sub cmdPrevious\_Click(sender As Object, e As EventArgs) Handles

cmdPrevious.Click

'Called when we click the move to previous button < in Course listing

BindingContext(dsCourses, "Courses").Position = (BindingContext(dsCourses,

"Courses").Position - 1)

End Sub

Private Sub cmdNext\_Click(sender As Object, e As EventArgs) Handles

cmdNext.Click

'Called when we click the move to next button > in Course listing

BindingContext(dsCourses, "Courses").Position = (BindingContext(dsCourses,

"Courses").Position + 1)

End Sub

Private Sub cmdLast\_Click(sender As Object, e As EventArgs) Handles

cmdLast.Click

'Called when we click the move to last button >> in Course listing

BindingContext(dsCourses, "Courses").Position =

(dsCourses.Tables("Courses").Rows.Count - 1)

End Sub

Private Sub cmdAddToSchedule\_Click(sender As Object, e As EventArgs) Handles

cmdAddToSchedule.Click

Dim DBConn As OleDbConnection

Dim DBCmd As OleDbCommand = New OleDbCommand()

Dim strSQLCmd As String

Dim DBAdapt As OleDbDataAdapter

Dim dsTemp As New DataSet()

Dim intCurrentRecord As Integer

'This technique is going to illustrate the second way that we can

'update a database...that is, through direct SQL manipulation.

'Now try to open up a connection to the database

DBConn = New OleDbConnection(strConn)

DBConn.Open()

'Use SQL to insert a new row into Registration

DBCmd.CommandText = "INSERT INTO Registration (SID, CID) VALUES ('" &

txtID.Text & "','" & txtCourseNumber.Text & "')"

DBCmd.Connection = DBConn

DBCmd.ExecuteNonQuery() 'Since it's a non-SELECT statement

'Now use another SQL statement to update the number of students that

'are enrolled in the class...

DBCmd.CommandText = "UPDATE Courses Set NumberEnrolled = '" &

Trim(CStr((CInt(txtNumberEnrolled.Text) + 1))) &

"' WHERE CID = '" & txtCourseNumber.Text & "'"

DBCmd.ExecuteNonQuery()

DBConn.Close()

'Refresh the Course database to reflect the new student counts

'and then move back to current record

'store current record number that we are on

intCurrentRecord = BindingContext(dsCourses, "Courses").Position

'refresh dataset

strSQLCmd = "Select \* From Courses"

DBAdaptCourses = New OleDbDataAdapter(strSQLCmd, strConn)

dsCourses.Clear()

DBAdaptCourses.Fill(dsCourses, "Courses")

'move back to same record we were on before refresh

BindingContext(dsCourses, "Courses").Position = intCurrentRecord

'Now we need to refresh this student's Schedule as well, so...

strSQLCmd = "Select \* From Registration Where SID = '" &

Trim(txtStudentID.Text) & "'"

DBAdaptRegistration = New OleDbDataAdapter(strSQLCmd, strConn)

dsRegistration.Clear()

DBAdaptRegistration.Fill(dsRegistration, "Registration")

'Refresh the data grid showing the schedule so that it's accurate

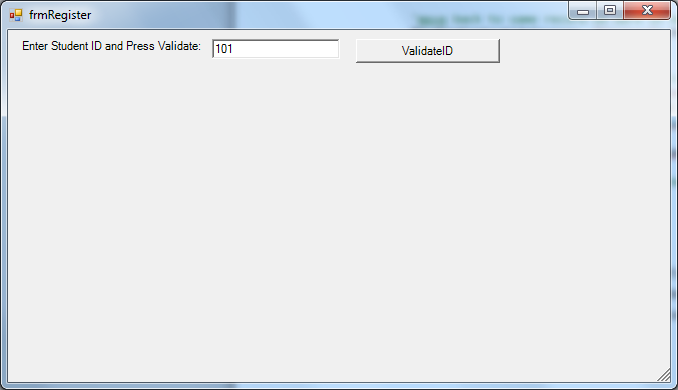
dgvSchedule.Refresh()

End Sub

End Class

Now that we've seen the source code, let's take a look at the application when it's running. I skipped over the first several message boxes that simply inform the user that the database was created, the tables were created and some data was placed in the tables.

The next screenshot shows the main application form as it first appears to the user. The only thing that a user can do at this point is enter his or her student ID number and press the Validate button to attempt to get access to the system.

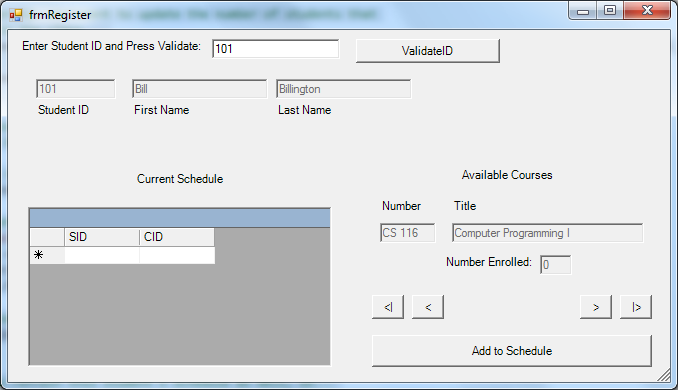


If the student was successfully validated by the system, i.e. the student's ID number was located in the Students table, the student will then be shown his or her current schedule along with the ability to move through and add new courses to the schedule. If the student couldn't be validated, then he or she is given the opportunity to add his or her personal information and then press the Update button which will cause the data to be written out to the Students table.

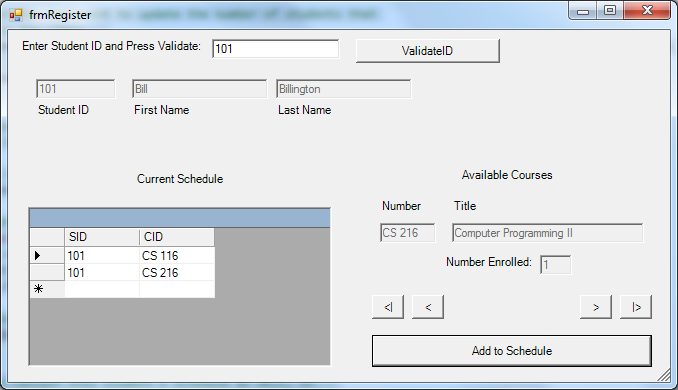
The following screenshot shows the student entering in personal information. The student can press the Cancel button at any time to abort the data entry and return back to the initial ID and Validate piece. Notice the appearance/disappearance of the Update/Cancel panel.



Once the student has been validated, the system will then show the student's current course schedule in the lower left hand DataGridView. The lower right hand group of controls will allow the student to move through and select courses that he or she wants to add to his or her schedule. The following screenshot shows an example of this.



The student can now add classes to his or her schedule. If you watch the Number Enrolled textbox, you will see that it automatically is incremented by 1 each time the student adds that particular course. The application as it currently exists does not provide any deletion options to remove courses nor does it contain any course conflict logic.



As you can see, working with databases and building up elaborate forms isn’t really all that tough – and the good news is that we are always in control! If the whole idea of having to set the CPU type to x86 (32 bit) doesn’t sit well with you using ADOX, don’t despair – we can also natively create SQL Server databases from Visual Basic too and those can be 32 or 64 bit!

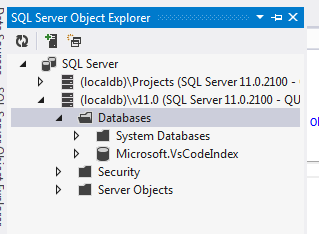
## LINQ to SQL

So, you already understand that SQL is the lingua franca of databases. Microsoft, in its wisdom has decided that if LINQ is good for querying in memory data sources, why not extend it to databases. Personally, I wish they would have done it the other way around. Anyway if you decide you really don’t want to learn SQL, you can use LINQ to do all of your database talking.

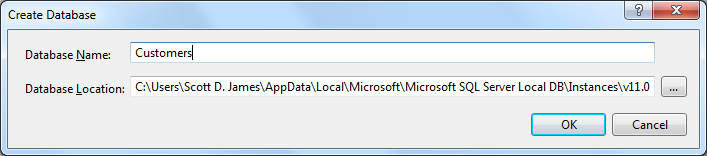
What LINQ will do is create two classes for each table in your database: one class that holds a representation of a row from the table and then a second class that represents the table as a collection of all the rows. Any cardinality information is also preserved in the classes that are created.

Now LINQ to SQL only works with SQL Server databases. To me, that’s a big limitation! Secondly you have to go through several gyrations of wizards and add data connections to get it to work. Since you’re not speaking real SQL anyway, I ask the question why bother? Anyway, if you have to know…

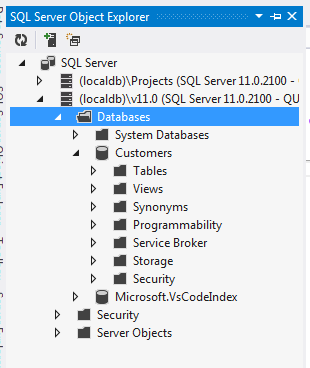
Since we can only use SQL Server tables, let’s begin by Adding one. You can view the SQL Server Explorer by selecting View🡪SQL Server Explorer:



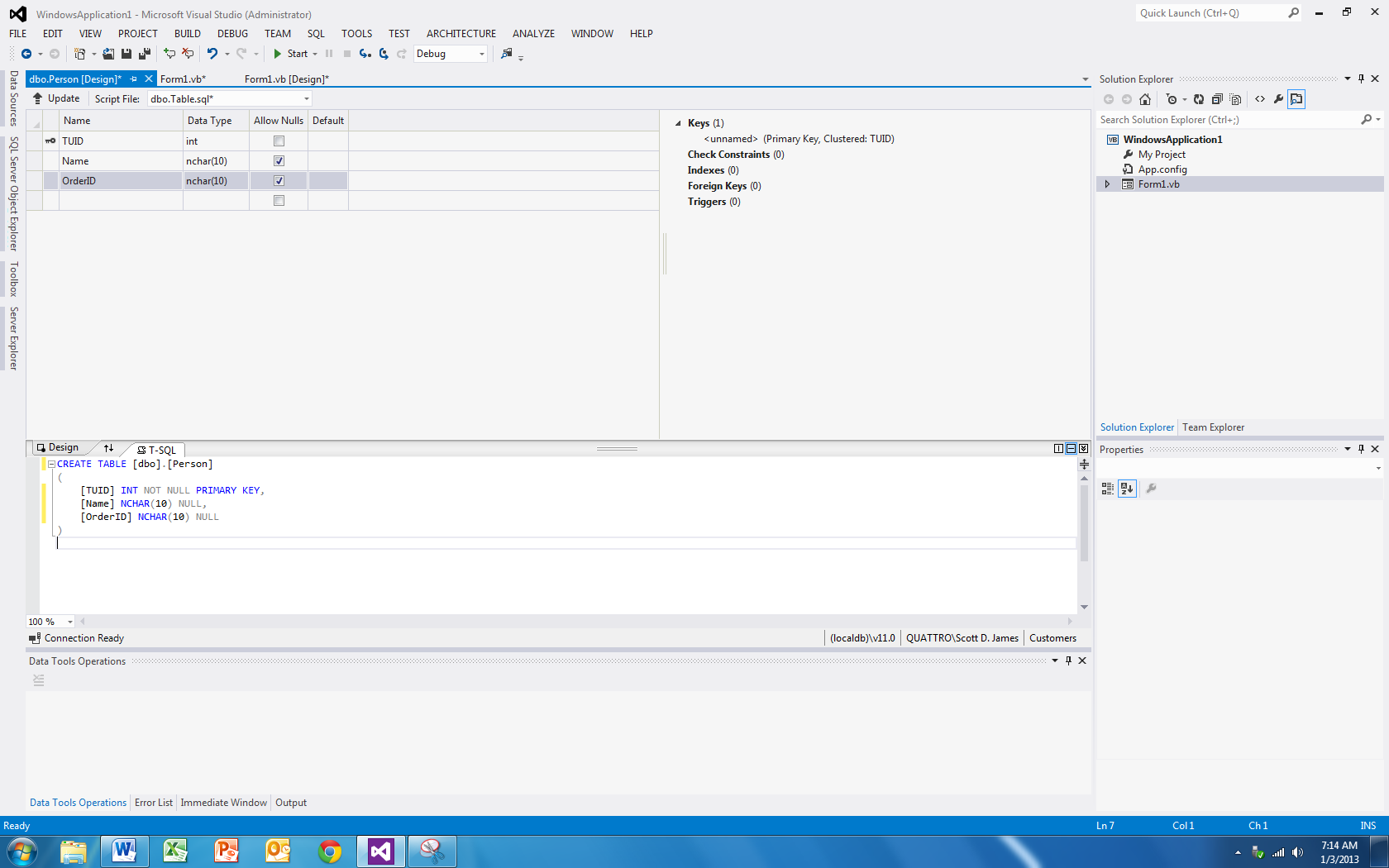
What’s really cool about this is that you can right click and add databases and tables (and eventually even the data) right from the Visual Studio interface. Now, things aren’t as powerful as the SQL Server Management Studio software itself, but they aren’t too bad either. I am going to right click and Add New Database (called Customers):



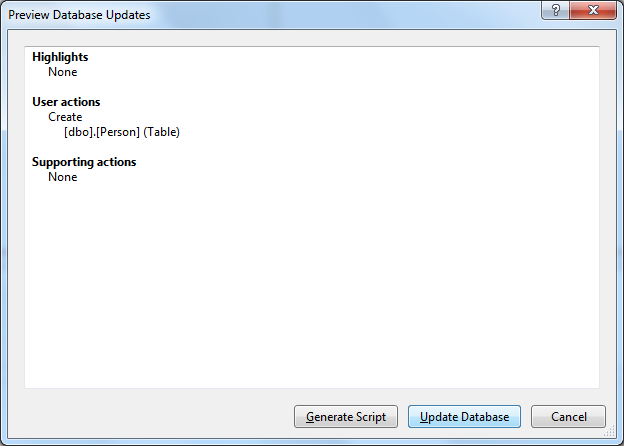
Here’s the updated Explorer window:



Now, I will right click on Tables and Add New Table. When I do this I get an editor area similar to Access:

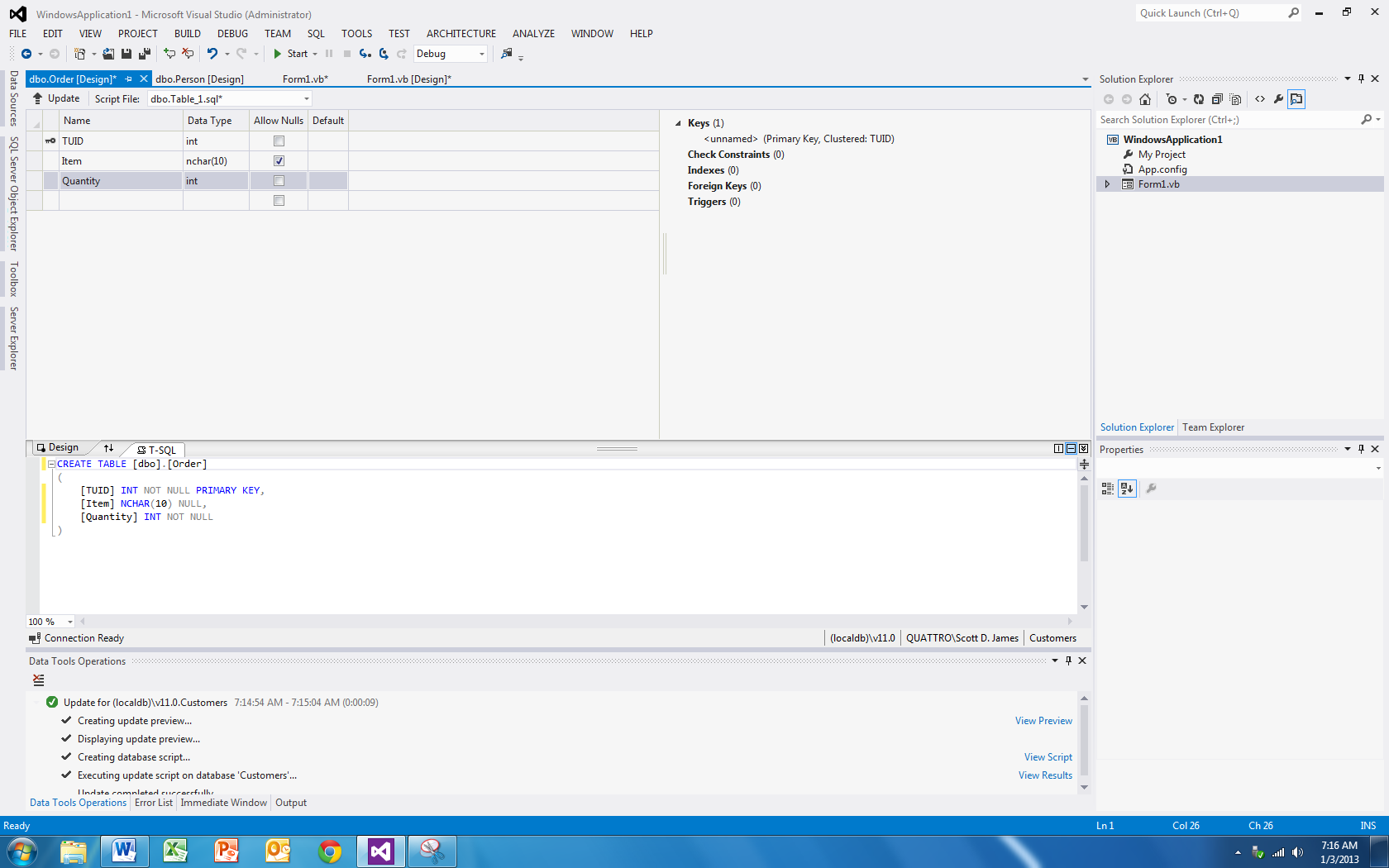


I added my columns and their types in the upper pane, changed the table name in the SQL text window and then pressed the Update button (just under the ribbon) to make the changes. A new screen pops up:

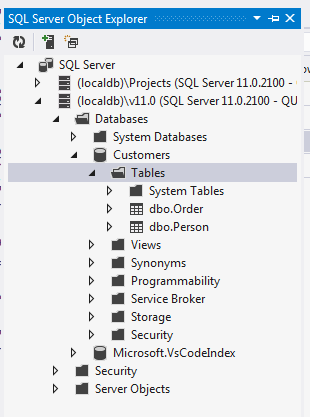


From here, I click Update Database and the table is actually created!

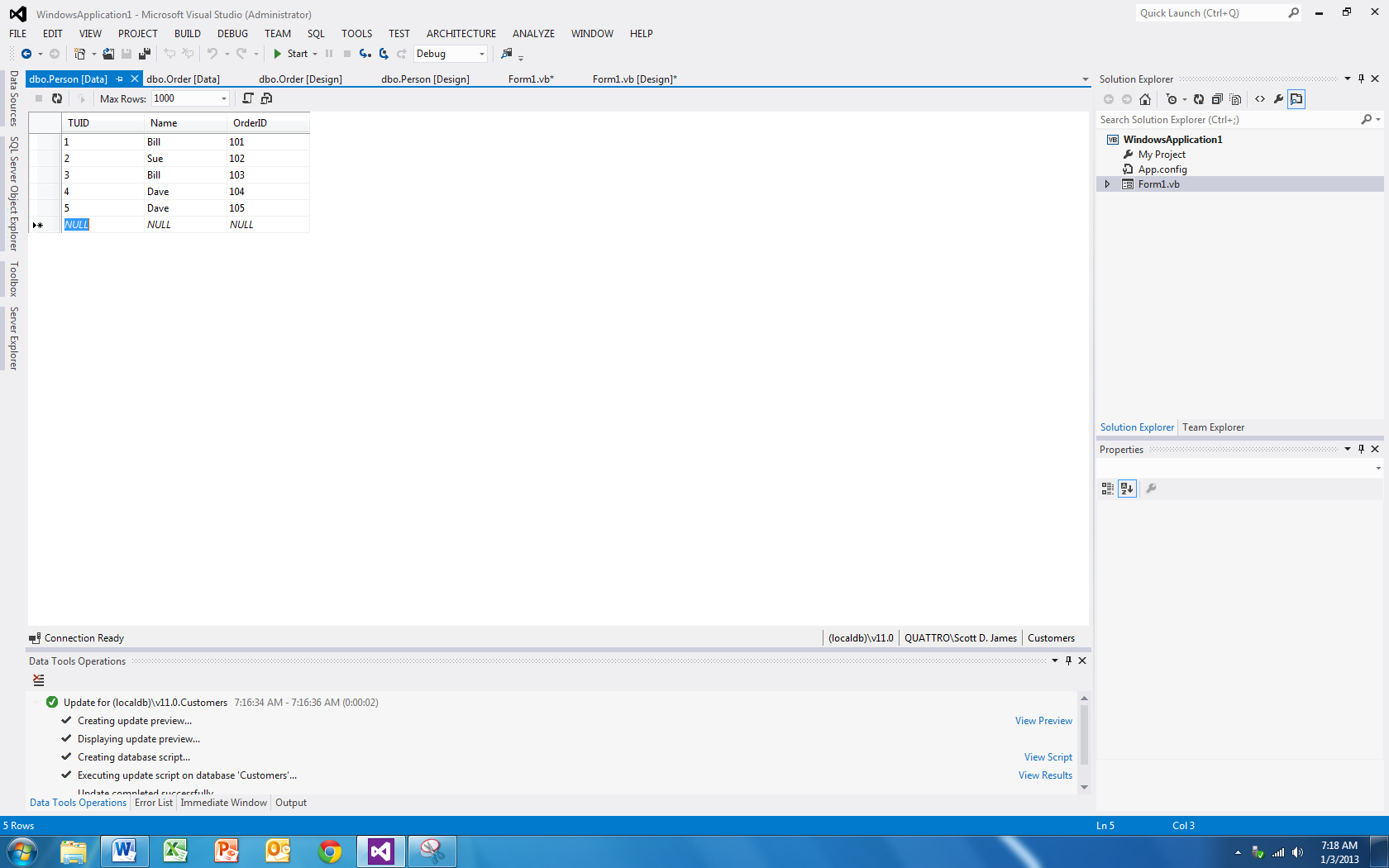
Now, I am going to do the same thing for an Order table:



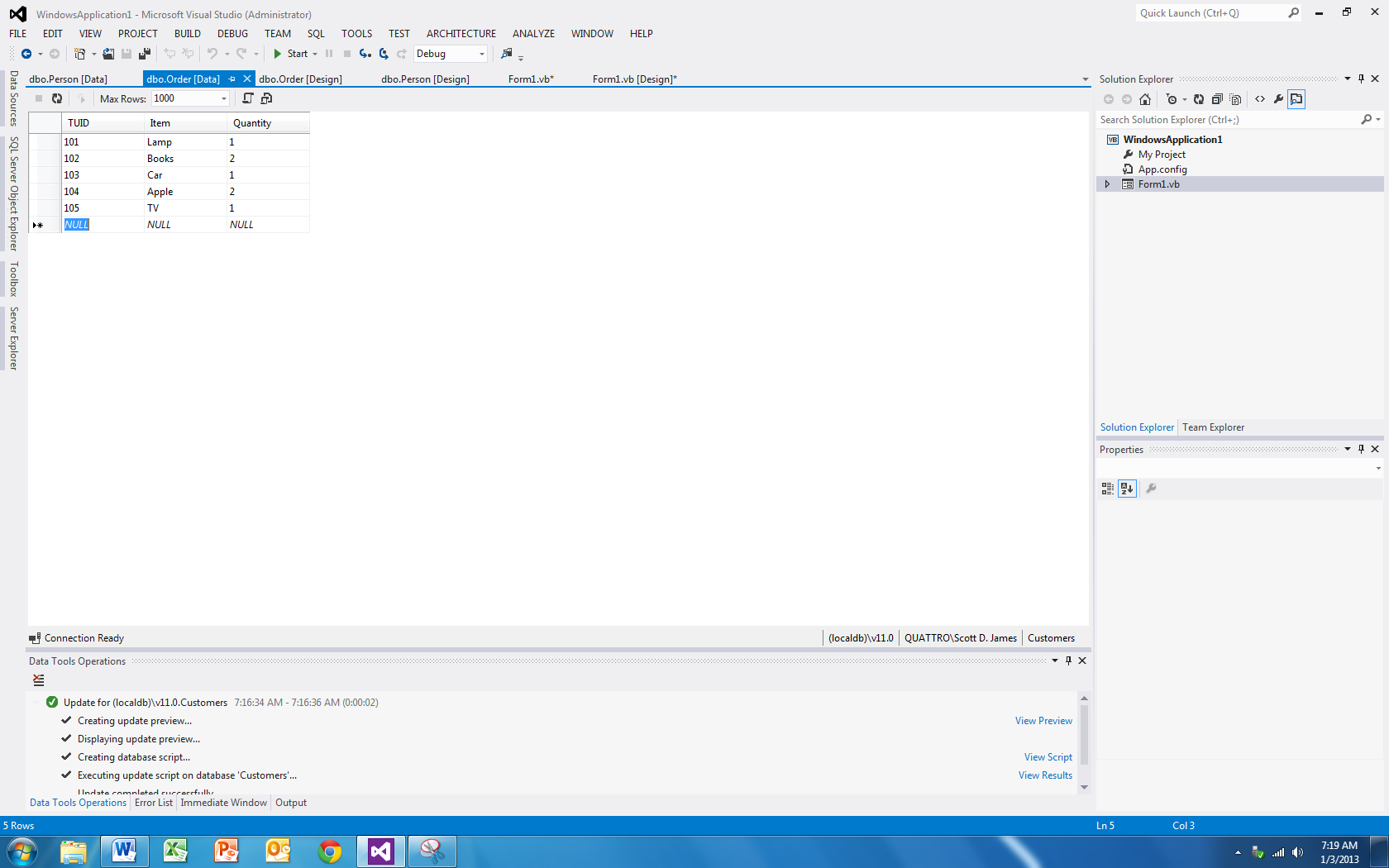
Here’s our updated explorer window:



Next, we need to add some data to each table. To do this, I simply right click on a table name in the Explorer window and select View Data. The following screen appears where I can enter my data:

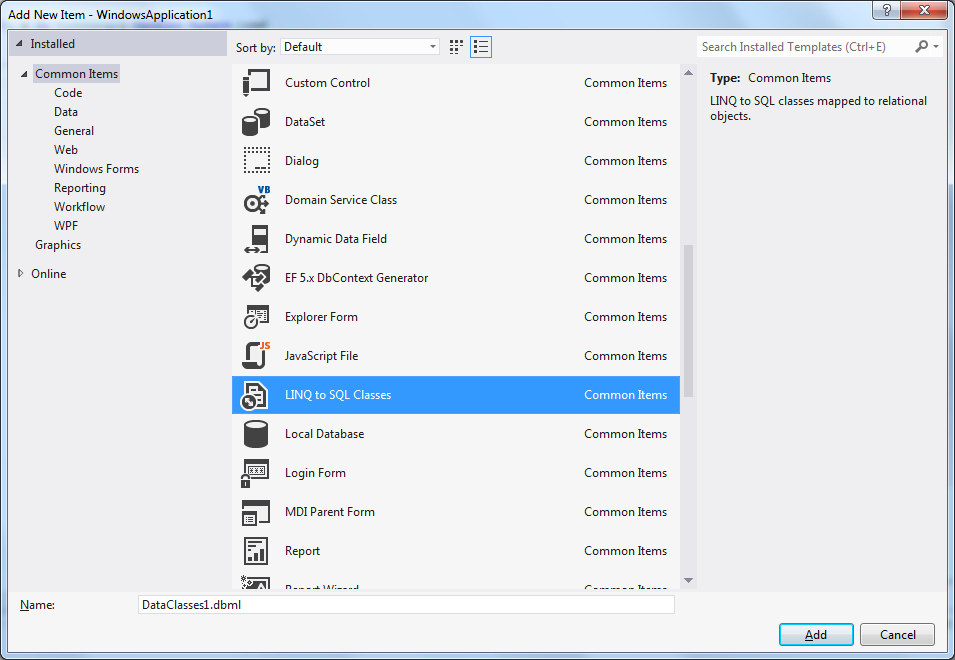


Here’s the Order table data:

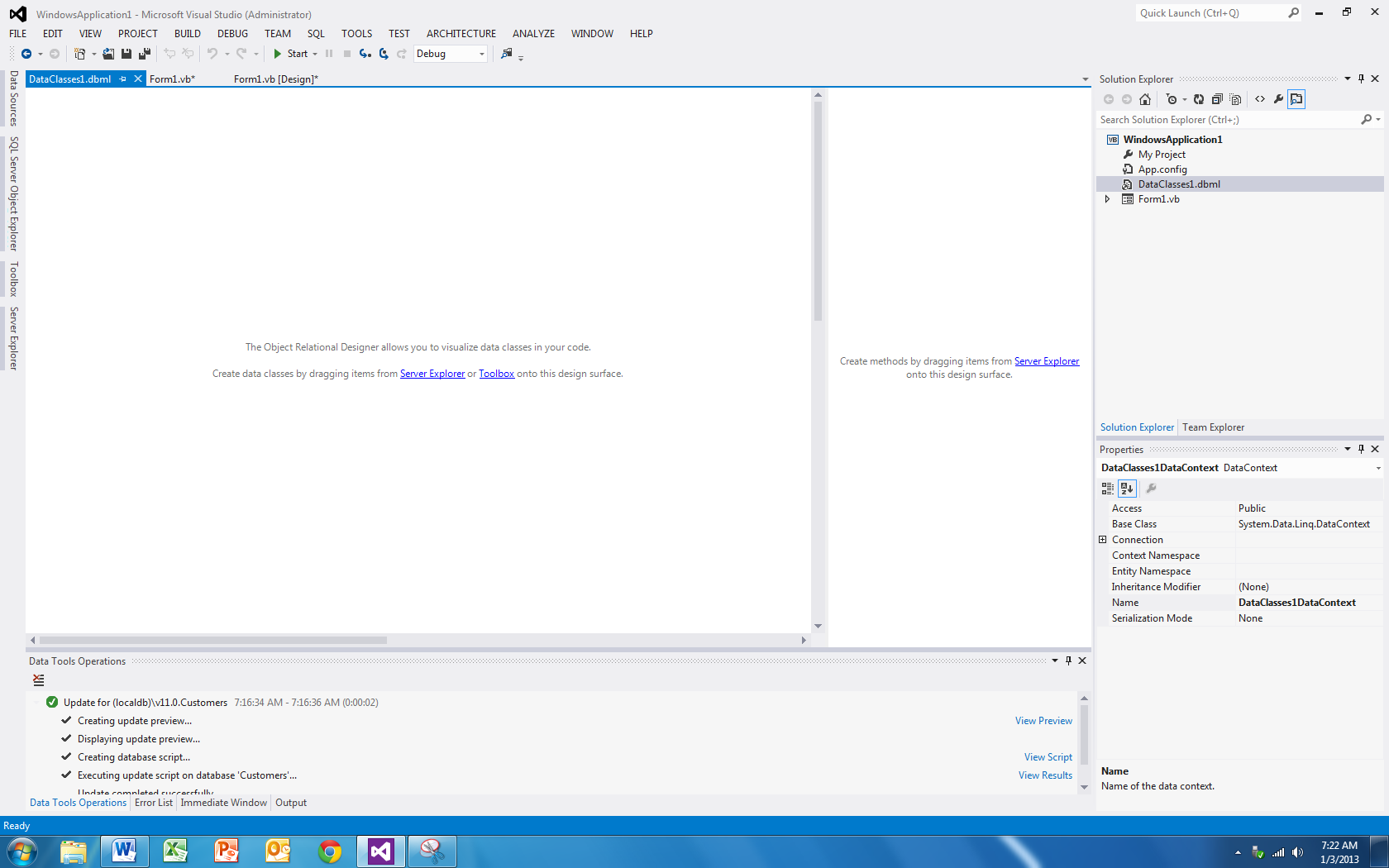


Congratulations, you just built a SQL Server database, added some tables and put some data in. That wasn’t so scary was it – in fact it’s probably preferable to messing with Access since everything is built in. We can now shut all of the windows down except for our form.

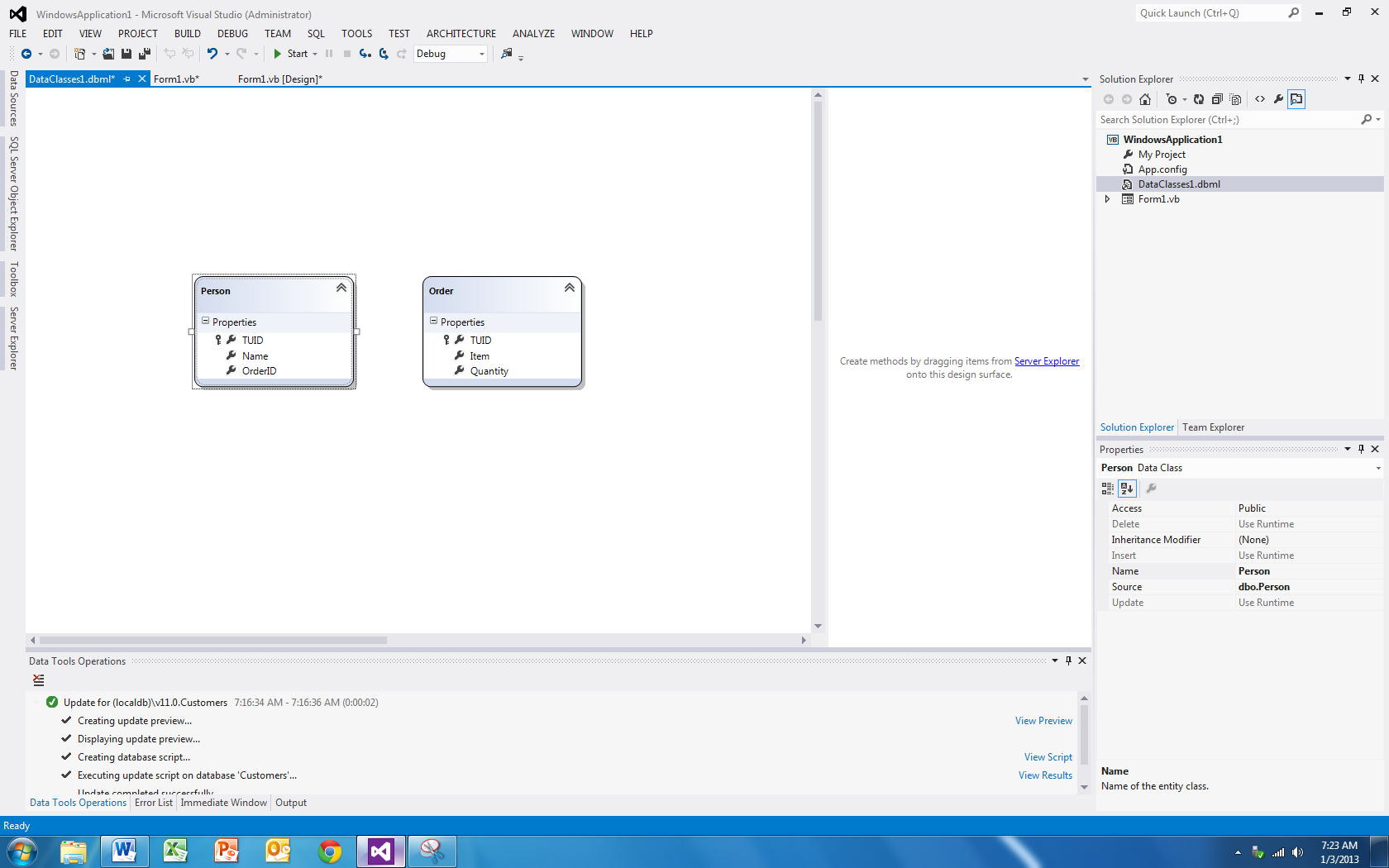
The next step is to get our Object Relation Designer canvas open. Here is where we will be able to add the tables we want LINQ to SQL classes generated for. To do this add a new item under project, and select LINQ to SQL as the type:



We will get a new screen that shows up (a database modeling language or dbml):



The next step simply involves dragging your tables from the explorer window onto the new canvas:



How’s that for slick? We have a graphical representation of our database tables. Now, the next thing we need to do is build the application. Sometimes Visual Studio isn’t too swift on updating things until after a build is complete. To do this simply select Build🡪Build Application from the menu.

Now add a command button to your form and enter the following code:

'Chapter 15 - Program 6

Public Class Form1

Private Sub Button1\_Click(sender As Object, e As EventArgs) Handles Button1.Click

'A connection string which I copied from walking through the Add New Data

'Connection after the Build command

Dim CustomerDB As New SqlClient.SqlConnection("Data Source=(localdb)\v11.0;Initial Catalog=Customers;Integrated Security=True")

'The link to the database through the classes that were generated

'-- the dbml file was called DataClasses1, so the generated data

'context will be called DataClasses1DataContext

Dim CustomerLink = New DataClasses1DataContext(CustomerDB)

Dim strOutput As String

'Here's good old LINQ...

Dim myPeeps = From aPerson In CustomerLink.Persons, anOrder In

CustomerLink.Orders

Where aPerson.OrderID = anOrder.TUID

Select aPerson.Name, anOrder.Item, anOrder.Quantity

For Each apeep In myPeeps

strOutput &= "Name: " & apeep.Name & " - Item: " & apeep.Item &

" - Quantity: " & apeep.Quantity & vbCrLf

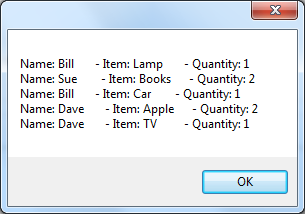
Next

MessageBox.Show(strOutput)

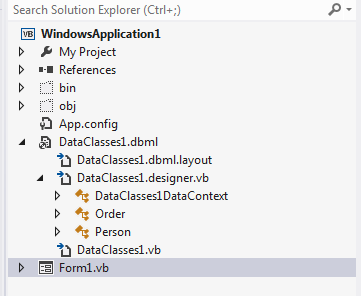
End Sub

End Class

Here’s the output:



Kind of a lot of work to be able to mess with non-SQL, but that’s your call. At least you learned how to create SQL Server files. Also, if you expand to Show All Files in your project, you can see the classes that were written to map the LINQ to SQL for you:



Here’s the code generated for the DataClasses1DataContext:

'------------------------------------------------------------------------------

' <auto-generated>

' This code was generated by a tool.

' Runtime Version:4.0.30319.17929

'

' Changes to this file may cause incorrect behavior and will be lost if

' the code is regenerated.

' </auto-generated>

'------------------------------------------------------------------------------

Option Strict On

Option Explicit On

Imports System

Imports System.Collections.Generic

Imports System.ComponentModel

Imports System.Data

Imports System.Data.Linq

Imports System.Data.Linq.Mapping

Imports System.Linq

Imports System.Linq.Expressions

Imports System.Reflection

<Global.System.Data.Linq.Mapping.DatabaseAttribute(Name:="Customers")> \_

Partial Public Class DataClasses1DataContext

Inherits System.Data.Linq.DataContext

Private Shared mappingSource As System.Data.Linq.Mapping.MappingSource =

New AttributeMappingSource()

#Region "Extensibility Method Definitions"

Partial Private Sub OnCreated()

End Sub

Partial Private Sub InsertOrder(instance As [Order])

End Sub

Partial Private Sub UpdateOrder(instance As [Order])

End Sub

Partial Private Sub DeleteOrder(instance As [Order])

End Sub

Partial Private Sub InsertPerson(instance As Person)

End Sub

Partial Private Sub UpdatePerson(instance As Person)

End Sub

Partial Private Sub DeletePerson(instance As Person)

End Sub

#End Region

Public Sub New()

MyBase.New(Global.WindowsApplication1.My.MySettings.Default.CustomersConnectionString, mappingSource)

OnCreated

End Sub

Public Sub New(ByVal connection As String)

MyBase.New(connection, mappingSource)

OnCreated

End Sub

Public Sub New(ByVal connection As System.Data.IDbConnection)

MyBase.New(connection, mappingSource)

OnCreated

End Sub

Public Sub New(ByVal connection As String, ByVal mappingSource As

System.Data.Linq.Mapping.MappingSource)

MyBase.New(connection, mappingSource)

OnCreated

End Sub

Public Sub New(ByVal connection As System.Data.IDbConnection, ByVal

mappingSource As System.Data.Linq.Mapping.MappingSource)

MyBase.New(connection, mappingSource)

OnCreated

End Sub

Public ReadOnly Property Orders() As System.Data.Linq.Table(Of [Order])

Get

Return Me.GetTable(Of [Order])

End Get

End Property

Public ReadOnly Property Persons() As System.Data.Linq.Table(Of Person)

Get

Return Me.GetTable(Of Person)

End Get

End Property

End Class

Here’s the Order class (still in the same physical file as above):

<Global.System.Data.Linq.Mapping.TableAttribute(Name:="dbo.[Order]")> \_

Partial Public Class [Order]

Implements System.ComponentModel.INotifyPropertyChanging,

System.ComponentModel.INotifyPropertyChanged

Private Shared emptyChangingEventArgs As PropertyChangingEventArgs = New

PropertyChangingEventArgs(String.Empty)

Private \_TUID As Integer

Private \_Item As String

Private \_Quantity As Integer

#Region "Extensibility Method Definitions"

Partial Private Sub OnLoaded()

End Sub

Partial Private Sub OnValidate(action As System.Data.Linq.ChangeAction)

End Sub

Partial Private Sub OnCreated()

End Sub

Partial Private Sub OnTUIDChanging(value As Integer)

End Sub

Partial Private Sub OnTUIDChanged()

End Sub

Partial Private Sub OnItemChanging(value As String)

End Sub

Partial Private Sub OnItemChanged()

End Sub

Partial Private Sub OnQuantityChanging(value As Integer)

End Sub

Partial Private Sub OnQuantityChanged()

End Sub

#End Region

Public Sub New()

MyBase.New

OnCreated

End Sub

<Global.System.Data.Linq.Mapping.ColumnAttribute(Storage:="\_TUID",

DbType:="Int NOT NULL", IsPrimaryKey:=true)> \_

Public Property TUID() As Integer

Get

Return Me.\_TUID

End Get

Set

If ((Me.\_TUID = value) \_

= false) Then

Me.OnTUIDChanging(value)

Me.SendPropertyChanging

Me.\_TUID = value

Me.SendPropertyChanged("TUID")

Me.OnTUIDChanged

End If

End Set

End Property

<Global.System.Data.Linq.Mapping.ColumnAttribute(Storage:="\_Item",

DbType:="NChar(10)")> \_

Public Property Item() As String

Get

Return Me.\_Item

End Get

Set

If (String.Equals(Me.\_Item, value) = false) Then

Me.OnItemChanging(value)

Me.SendPropertyChanging

Me.\_Item = value

Me.SendPropertyChanged("Item")

Me.OnItemChanged

End If

End Set

End Property

<Global.System.Data.Linq.Mapping.ColumnAttribute(Storage:="\_Quantity",

DbType:="Int NOT NULL")> \_

Public Property Quantity() As Integer

Get

Return Me.\_Quantity

End Get

Set

If ((Me.\_Quantity = value) \_

= false) Then

Me.OnQuantityChanging(value)

Me.SendPropertyChanging

Me.\_Quantity = value

Me.SendPropertyChanged("Quantity")

Me.OnQuantityChanged

End If

End Set

End Property

Public Event PropertyChanging As PropertyChangingEventHandler Implements

System.ComponentModel.INotifyPropertyChanging.PropertyChanging

Public Event PropertyChanged As PropertyChangedEventHandler Implements

System.ComponentModel.INotifyPropertyChanged.PropertyChanged

Protected Overridable Sub SendPropertyChanging()

If ((Me.PropertyChangingEvent Is Nothing) \_

= false) Then

RaiseEvent PropertyChanging(Me, emptyChangingEventArgs)

End If

End Sub

Protected Overridable Sub SendPropertyChanged(ByVal propertyName As

[String])

If ((Me.PropertyChangedEvent Is Nothing) \_

= false) Then

RaiseEvent PropertyChanged(Me, New

PropertyChangedEventArgs(propertyName))

End If

End Sub

End Class

Here’s the Person class (still in the same physical file as above):

<Global.System.Data.Linq.Mapping.TableAttribute(Name:="dbo.Person")> \_

Partial Public Class Person

Implements System.ComponentModel.INotifyPropertyChanging,

System.ComponentModel.INotifyPropertyChanged

Private Shared emptyChangingEventArgs As PropertyChangingEventArgs = New

PropertyChangingEventArgs(String.Empty)

Private \_TUID As Integer

Private \_Name As String

Private \_OrderID As String

#Region "Extensibility Method Definitions"

Partial Private Sub OnLoaded()

End Sub

Partial Private Sub OnValidate(action As System.Data.Linq.ChangeAction)

End Sub

Partial Private Sub OnCreated()

End Sub

Partial Private Sub OnTUIDChanging(value As Integer)

End Sub

Partial Private Sub OnTUIDChanged()

End Sub

Partial Private Sub OnNameChanging(value As String)

End Sub

Partial Private Sub OnNameChanged()

End Sub

Partial Private Sub OnOrderIDChanging(value As String)

End Sub

Partial Private Sub OnOrderIDChanged()

End Sub

#End Region

Public Sub New()

MyBase.New

OnCreated

End Sub

<Global.System.Data.Linq.Mapping.ColumnAttribute(Storage:="\_TUID",

DbType:="Int NOT NULL", IsPrimaryKey:=true)> \_

Public Property TUID() As Integer

Get

Return Me.\_TUID

End Get

Set

If ((Me.\_TUID = value) \_

= false) Then

Me.OnTUIDChanging(value)

Me.SendPropertyChanging

Me.\_TUID = value

Me.SendPropertyChanged("TUID")

Me.OnTUIDChanged

End If

End Set

End Property

<Global.System.Data.Linq.Mapping.ColumnAttribute(Storage:="\_Name",

DbType:="NChar(10)")> \_

Public Property Name() As String

Get

Return Me.\_Name

End Get

Set

If (String.Equals(Me.\_Name, value) = false) Then

Me.OnNameChanging(value)

Me.SendPropertyChanging

Me.\_Name = value

Me.SendPropertyChanged("Name")

Me.OnNameChanged

End If

End Set

End Property

<Global.System.Data.Linq.Mapping.ColumnAttribute(Storage:="\_OrderID",

DbType:="NChar(10)")> \_

Public Property OrderID() As String

Get

Return Me.\_OrderID

End Get

Set

If (String.Equals(Me.\_OrderID, value) = false) Then

Me.OnOrderIDChanging(value)

Me.SendPropertyChanging

Me.\_OrderID = value

Me.SendPropertyChanged("OrderID")

Me.OnOrderIDChanged

End If

End Set

End Property

Public Event PropertyChanging As PropertyChangingEventHandler Implements

System.ComponentModel.INotifyPropertyChanging.PropertyChanging

Public Event PropertyChanged As PropertyChangedEventHandler Implements

System.ComponentModel.INotifyPropertyChanged.PropertyChanged

Protected Overridable Sub SendPropertyChanging()

If ((Me.PropertyChangingEvent Is Nothing) \_

= false) Then

RaiseEvent PropertyChanging(Me, emptyChangingEventArgs)

End If

End Sub

Protected Overridable Sub SendPropertyChanged(ByVal propertyName As

[String])

If ((Me.PropertyChangedEvent Is Nothing) \_

= false) Then

RaiseEvent PropertyChanged(Me, New

PropertyChangedEventArgs(propertyName))

End If

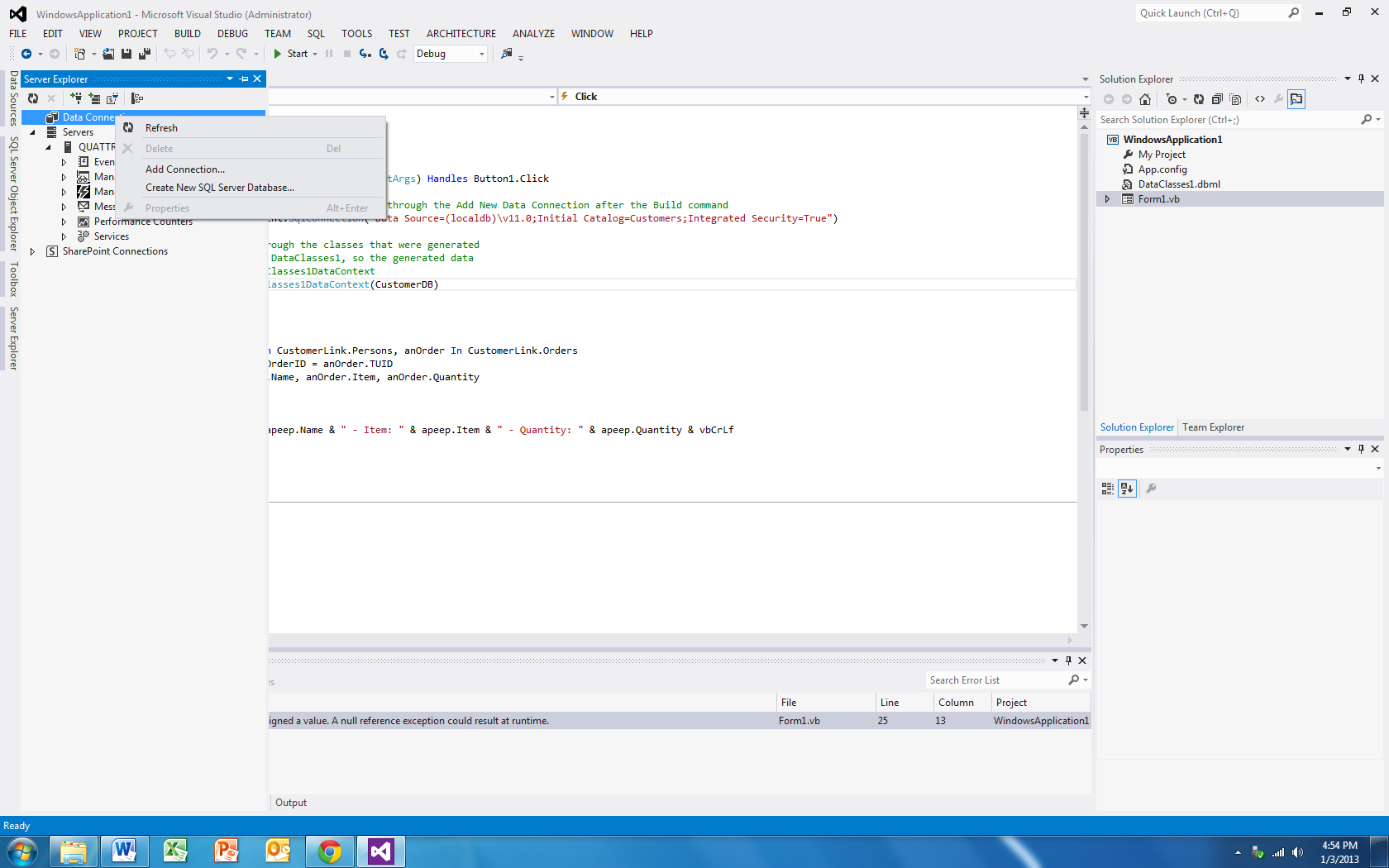
End Sub

End Class

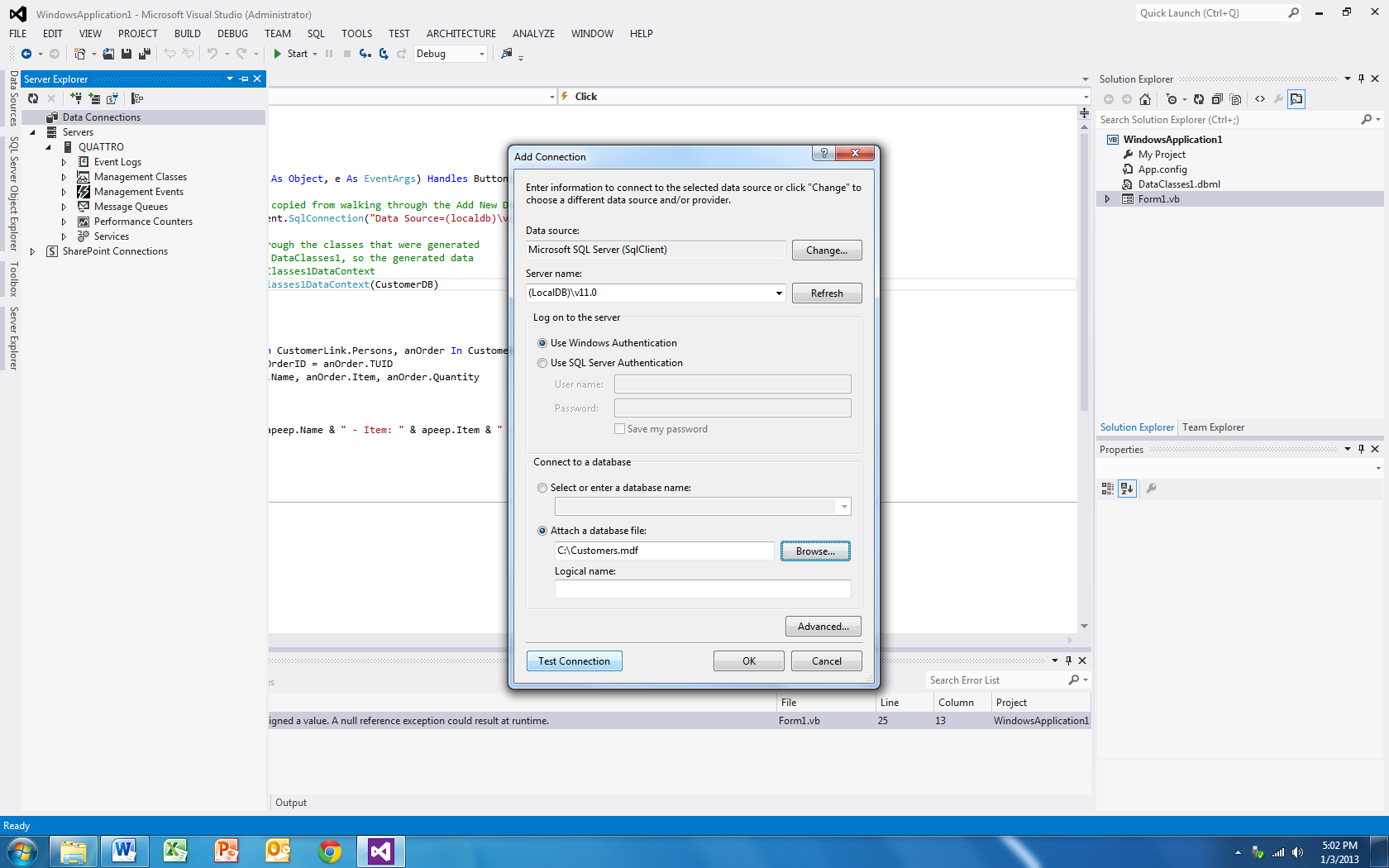
Sometimes once you see the magic, you kind of wish you hadn’t… Remember, nobody is making you use the Wizard related stuff – sometimes it helps out, sometimes it doesn’t; you’ll have to be the judge. I still don’t think there’s a whole lot of benefit to using LINQ to SQL.

## Reattaching a SQL Server File

To attach a SQL Server database file (.mdf), start off in the Server Explorer window:



Right click on the Data Connections and choose Add Connection. The Add Connection dialog box appears. We need to change some things on the dialog starting with the Change Data Source which we will change to Microsoft SQL Server and choose the OK button.



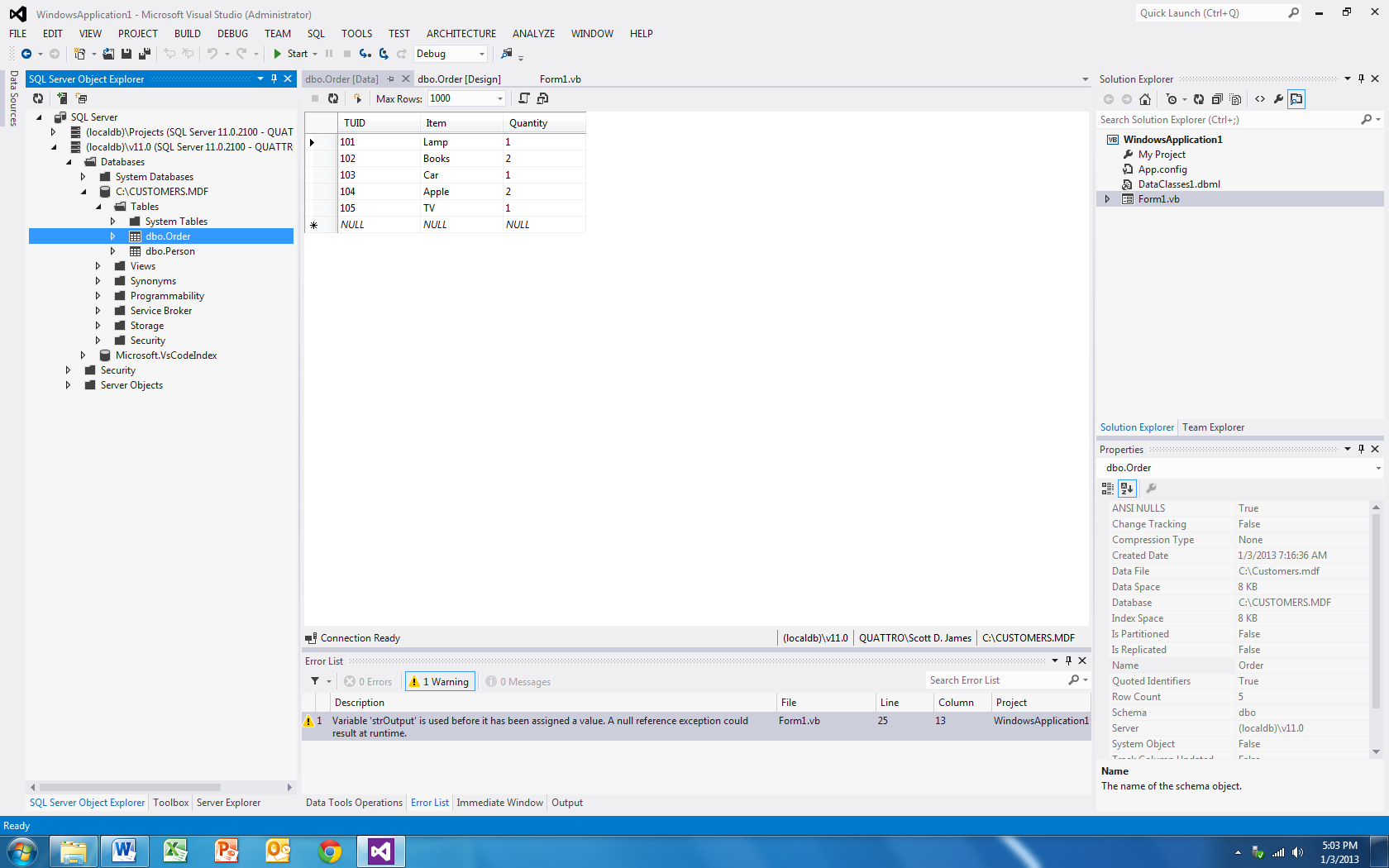
In the Server Name box, type or browse to the path to the local instance of SQL Server. You can type the following, depending on the particular SQL Server installation you have:

* "." for the default instance on your computer.
* "(LocalDB)\v11.0" for the default instance of SQL Server Express LocalDB.
* ".\SQLEXPRESS" for the default instance of SQL Server Express.

Just so you know, the default path for any of the LocalDB files will be located at:

C:\Users\*username*\AppData\Local\Microsoft\Microsoft SQL Server Local DB\Instances\v11.0

Once you’ve got things set up, choose the OK button. The new database appears in Server Explorer. It will now remain connected to SQL Server until you explicitly detach it.



# Review of the Database Related Classes

As I previously mentioned, we didn't hardly touch any of the classes, methods or properties that ADO.NET has for you to use. It is quite impressive that we can write applications as complicated as those that we looked at in this chapter with the few pieces that we did example.

This section is simply going to serve as a summary of the various classes that you encountered through the database examples. We will note any methods or properties that we used in the code examples. You’ll also notice other classes that you might want to look at within each set.

### BindingContext

|  |  |
| --- | --- |
| Method or Property | Purpose |
| AddNew | Adds a new record to the DataSet that the item is bound to |
| CancelCurrentEdit | Tells the bound DataSet to cancel any changes made to any records currently marked with changes |
| EndCurrentEdit | Tells the bound DataSet to end any records that may currently be being edited |
| Position | Returns the current record location within a DataSet |
| RemoveAt | Deletes a record from the DataSet at a given position |
| ResumeBinding | Resumes a data binding |
| SuspendBinding | Suspends a data binding |
| UpdateIsBinding | Updates a data binding |

### Controls

|  |  |
| --- | --- |
| Method or Property | Purpose |
| DataBindings.Add | Adds a new data binding to a control from a field in a DataSet |
| SetDataBinding | Specifies the field to bind to in a DataSet |

### DataSet

|  |  |
| --- | --- |
| Method or Property | Purpose |
| AcceptChanges | Accepts changes to the DataSet |
| CaseSensitive | Gets or sets a value indicating whether string comparisons within DataTable are case sensitive |
| Clear | Empties the DataSet |
| CreateDataReader | Returns a data reader |
| DataSetName | Gets or sets the name of the current dataset |
| DefaultViewManager | Gets a custom view of the data contained in the dataset for filtering, searching and navigating using a custom DataViewManager |
| EnforceConstraints | Gets or sets a value indicating whether constraint rules are followed when attempting any update operation |
| GetChanges | Gets a copy of the dataset containing all changes made to it since it was loaded or since AcceptChanges was called |
| HasChanges | Gets a value indicating whether the dataset has new, deleted or modified rows |
| HasErrors | Indicates whether there are errors in any of the DataTables object in the data set |
| Load | Fills a dataset with values from a data source using a DataReader |
| Merge | Merges an entity with the current dataset |
| Relations | The collection of relations that link tables and allow navigation from parent to child |
| RejectChanges | Rolls back all the changes made to the dataset since it was created or since the last time AcceptChanges was called |
| Reset | Clears all tables, relations, and so forth from the dataset |
| Tables | The collection of tables contained in the dataset |

### OleDbCommand

|  |  |
| --- | --- |
| Method or Property | Purpose |
| CommandText | A string containing the text of a SQL command |
| CommandTimeout | How long the command waits before terminating an attempt to execute a command and generating an error |
| Connection | The connection over which the OleDbCommand object is to be executed |
| ExecuteReader | Executes commands that return rows which cannot be changed |
| ExecuteNonQuery | Used to execute any OleDbCommand containing any SQL statement other than a SELECT: INSERT, DELETE and UPDATE statements |
| ExecuteScalar | Retrieves a single value, like an aggregate (MIN, MAX, COUNT) from a database |
| Parameters | Gets the OleDbParameter collection |
| Transaction | Gets or sets the OleDbTransaction within which the OleDbCommand executes |
| UpdatedRowSource | Gets or sets how command results are applied to the DataRow when used by the Update method |

### OleDbConnection

|  |  |
| --- | --- |
| Method or Property | Purpose |
| ChangeDatabase | Changes the current database for an open OleDbConnection |
| Close | Closes a connection to a database |
| ConnectionString | Gets the string used to open a database |
| ConnectionTimeout | How long to wait trying to establish a connection before terminating and generating and error |
| CreateCommand | Creates an returns an OleDbCommand object associated with the OleDbConnection |
| Database | Gets the name of the current database or the database to be used after a connection is opened |
| DataSource | The server name or file name of the data source |
| Open | Opens a connection to a database |
| Provider | Gets the name of the OLE DB provider specified in the connection string |
| ResetState | Updates the state property of the OleDbConnection |
| ServerVersion | Returns the server version that the client is connection to |
| State | Gets the current state of the connection |

### OleDbDataAdapter

|  |  |
| --- | --- |
| Method or Property | Purpose |
| AcceptChangesDuringFill | Indicates whether AcceptChanges is called on a DataRow after it is added to the DataTable during a Fill |
| AccceptChangesDuringUpdate | Indicates whether AcceptChanges is called during an Update |
| ContinueUpdateOnError | Specified whether to generate an exception when an error is encountered during a row update |
| DeleteCommand.CommandText | DeleteCommand is an OleDbCommand object. In particular this object holds the text string designating how to SQL DELETE information from a database. |
| Fill | Loads data into a DataSet |
| FillLoadOption | Determines how the adapter fills the DataTable from DbDataReader |
| FillSchema | Adds a DataTable to a DataSet and configures the schema to match that in the data source |
| GetFillParameters | Gets the parameters set by the user when executing a SQL SELECT statement |
| InsertCommand.CommandText | InsertCommand is an OleDbCommand object. In particular this object holds the text string designating how to SQL INSERT information into a database. |
| MissingMappingAction | Determines the action to take when incoming data does not have a matching table or column |
| MissingSchemaAction | Determines the action to take when an existing DataSet schema does not match incoming data |
| ResetFillLoadOption | Resets FillLoadOption back to its default state and causes Fill to again AcceptChangesDuringFill |
| SelectCommand.CommandText | SelectCommand is an OleDbCommand object. In particular this object holds the text string designating how to SQL SELECT information from a database. |
| TableMappings | The collection that provides the master mapping between a source table and a DataTable |
| Update | Sends information containing in a DataSet back to its underlying database |
| UpdateBatchSize | The value that enables or disables batch processing support and specifies the number of commands that can be executed in a batch |
| UpdateCommand.CommandText | UpdateCommand is an OleDbCommand object. In particular this object holds the text string designating how to SQL UPDATE information in a database. |

### DataReader

|  |  |
| --- | --- |
| Method or Property | Purpose |
| Close | Closes the OleDbDataReader |
| FieldCount | The number of columns in the current row |
| GetXXX | Gets the value of the specified column as type XXX (for example GetBoolean, GetBytes …) |
| GetDataTypeName | Gets the name of the source data type |
| GetFieldType | Gets the type that is in the data type of the object |
| GetName | Gets the name of the specified column |
| GetOrdinal | Gets the column ordinal given a column name |
| GetSchemaTable | Returns a DataTable that describes the column metadata of the OleDbDataReader |
| HasRows | Indicates whether the OleDbDataReader contains one or more rows |
| IsClosed | Indicates whether the data reader is closed |
| Item | Gets the value of a column in its native format |
| RecordsAffected | Gets the number of rows changes, inserted or deleted by the execution of the SQL statement |
| VisibleFieldCount | Gets the number of fields in the OleDbDataReader that are not hidden |

# Using SQL

This section of the chapter is going to provide a very brief introduction to using Structured Query Language (SQL). I strongly recommend that you take some time and learn this language since it will open up all sorts of database avenues for you.

We will work through the following examples with an Addresses table that has the following fields: FirstName, LastName, Address, City, State and ZipCode. Five records have been added to the table, which are displayed below for your information:

| **FirstName** | **LastName** | **Address** | **City** | **State** | **ZipCode** |
| --- | --- | --- | --- | --- | --- |
| Bill | Jones | 1523 Maple | Birch Run | MI | 48415 |
| Sue | Franklin | 100 South Huron | Burt | MI | 48417 |
| Tim | McMann | 4876 Beyer Road | Birch Run | MI | 48415 |
| Dawn | Davis | P.O. Box 174 | University Center | MI | 48734 |
| Ron | Williams | 15463 Elms Road | Montrose | MI | 48457 |

The main SQL statements that you need to know for performing queries include SELECT, WHERE, FROM, GROUP BY and ORDER BY. SQL is not case-sensitive. I tend to put all SQL statements in uppercase simply to differentiate between what is a keyword and what isn't.

The purpose of the SELECT statement is to return rows of information from a database table. Let's take a look at the purpose of each of the selection related keywords.

### Select Statement

SQL SELECT related keywords

|  |  |
| --- | --- |
| *Keyword* | *Purpose* |
| SELECT | Retrieves specified fields from a table |
| WHERE | Provides criteria for selecting records |
| FROM | Used to name the table from where to retrieve records |
| GROUP BY | Specifies how to group records |
| ORDER BY | Specifies how to sort records |
| DISTINCT | Limits values returned to be unique |

Here are a series of queries which return information from the Addresses table. Following each query is a summary of what the query does and a copy of the information that would be returned from the Addresses table.

SELECT \* FROM Addresses

This query would bring back every field from the Addresses table. The purpose of the \* is to tell SQL bring back all fields contained in the named table. Since there is no WHERE statement present in this query, every field for every record would be returned.

| **FirstName** | **LastName** | **Address** | **City** | **State** | **ZipCode** |
| --- | --- | --- | --- | --- | --- |
| Bill | Jones | 1523 Maple | Birch Run | MI | 48415 |
| Sue | Franklin | 100 South Huron | Burt | MI | 48417 |
| Tim | McMann | 4876 Beyer Road | Birch Run | MI | 48415 |
| Dawn | Davis | P.O. Box 174 | University Center | MI | 48734 |
| Ron | Williams | 15463 Elms Road | Montrose | MI | 48457 |

SELECT LastName, City, State FROM Addresses

This query would bring back all records since there is no WHERE statement present. Notice that the \* was not provided after the SELECT statement; instead three fields were provided. This query will only retrieve the LastName, City and State fields for each record from the table.

| **LastName** | **City** | **State** |
| --- | --- | --- |
| Jones | Birch Run | MI |
| Franklin | Burt | MI |
| McMann | Birch Run | MI |
| Davis | University Center | MI |
| Williams | Montrose | MI |

SELECT \* FROM Addresses WHERE City = "Birch Run"

This will select all fields for records whose City field contains the value Birch Run.

| **FirstName** | **LastName** | **Address** | **City** | **State** | **ZipCode** |
| --- | --- | --- | --- | --- | --- |
| Bill | Jones | 1523 Maple | Birch Run | MI | 48415 |
| Tim | McMann | 4876 Beyer Road | Birch Run | MI | 48415 |

SELECT FirstName FROM Addresses WHERE FirstName > "Jim"

This query will print the FirstName for all records in the Addresses table where the FirstName is greater than “Jim.”

| **FirstName** |
| --- |
| Sue |
| Tim |
| Ron |

SELECT City FROM Addresses WHERE City LIKE "B\*"

This will select all cities from the table which start with the letter B. Notice that both instances of "Birch Run" are returned.

| **City** |
| --- |
| Birch Run |
| Burt |
| Birch Run |

SELECT DISTINCT City FROM Addresses WHERE City LIKE "B\*"

This will select all cities from the table which start with the letter B. Notice that the DISTINCT keyword returns only the first occurrence of "Birch Run."

| **City** |
| --- |
| Birch Run |
| Burt |

SELECT ZipCode FROM Addresses WHERE ZipCode LIKE "4841?"

This will select all zipcodes from the table which start with 4841 and have only one more character after the 1.

| **ZipCode** |
| --- |
| 48415 |
| 48417 |
| 48415 |

SELECT \* FROM Addresses ORDER BY LastName ASC

This will sort all records in ascending order (ASC) by the person's last name.

| **FirstName** | **LastName** | **Address** | **City** | **State** | **ZipCode** |
| --- | --- | --- | --- | --- | --- |
| Dawn | Davis | P.O. Box 174 | University Center | MI | 48734 |
| Sue | Franklin | 100 South Huron | Burt | MI | 48417 |
| Bill | Jones | 1523 Maple | Birch Run | MI | 48415 |
| Tim | McMann | 4876 Beyer Road | Birch Run | MI | 48415 |
| Ron | Williams | 15463 Elms Road | Montrose | MI | 48457 |

SELECT \* FROM Addresses ORDER BY City, Lastname ASC

This will sort all records by city and then by last name within city in ascending order.

| **FirstName** | **LastName** | **Address** | **City** | **State** | **ZipCode** |
| --- | --- | --- | --- | --- | --- |
| Bill | Jones | 1523 Maple | Birch Run | MI | 48415 |
| Tim | McMann | 4876 Beyer Road | Birch Run | MI | 48415 |
| Sue | Franklin | 100 South Huron | Burt | MI | 48417 |
| Ron | Williams | 15463 Elms Road | Montrose | MI | 48457 |
| Dawn | Davis | P.O. Box 174 | University Center | MI | 48734 |

Here are some other examples of the SELECT statement using some additional keywords that we didn't cover in detail. I also didn't print the details of the results of these queries out since you can probably figure them out.

SELECT \* FROM Addresses WHERE City BETWEEN "A\*" and "C\*"

This would return all fields from the Addresses table where the City name starts with an A, B or C.

SELECT \* FROM Addresses WHERE City IN ("Birch Run", "Burt", "Escanaba")

This would return all fields from the Addresses table where the City name is in the set of Birch Run, Burt or Escanaba.

SELECT COUNT(EmployeeID) AS NumberEmployees, AVG(Age) AS AverageAge, MIN(Salary) AS SmallestSalary, MAX(Salary) AS LargestSalary FROM Employees

This query will count the number of Employee IDs (which tell us the number of employees that the company has) and print the results under the column NumberEmployees. The query will also use the SQL AVG function to calculate the AverageAge of all employees, which is printed under the column AverageAge. Likewise, the MIN and MAX functions are used to calculate the SmallestSalary and LargestSalary that any employee earns.

While the SELECT statement is probably the most common that you will use in your database processing, there are three other SQL commands that you should be aware of. Again, we will look at some fairly simple examples of each command and you should be able to figure out what each does from the previous discussion on the SELECT statement.

### The Insert Statement

The purpose of the SQL INSERT statement is to place new records into a table. The syntax for the Insert statement has the following form:

INSERT INTO tableName (fieldName1, fieldName2, … fieldNameN) VALUES (value1, value2, … valueN)

INSERT INTO Employees (EmployeeID, Age, Salary) VALUES (105, 22, 15.37)

This statement inserts a new record into the Employees table. The EmployeeID field receives the value 105, the Age field receives the value 22 and the Salary field receives the value 15.37.

### The Update Statement

The purpose of the SQL Update statement is to change values for records that already exist in a database. The syntax for the Update statement has the following form:

UPDATE tableName SET fieldName1 = value1, fieldName2 = value2, fieldNameN = valueN WHERE criteria

UPDATE Addresses SET City = "Marquette" WHERE ZipCode = 49855

This statement would set the City name to Marquette for all records in the Addresses table where the ZipCode is 49855.

### The Delete Statement

The DELETE statement is used to remove records from a table.

DELETE \* FROM Addresses

This would DELETE all records from the Addresses table.

DELETE \* FROM Addresses WHERE City NOT IN ("Birch Run", "University Center")

This statement would DELETE all records in the Addresses table that have cities other than Birch Run and University Center.

This was a quick introduction to the SQL language and is by no means even close to a complete coverage of the language. There are a number of great books on both databases and SQL available. If you haven't worked much with a database, it is really in your best interest to learn SQL since just about every database that's used today allows you to manipulate its data via the SQL language.