CS708 Lecture Notes

Visual Basic.NET Object-Oriented Programming

Implementing Business Objects (Part II)
Business Rules & Validation

Part (II of III)

(Lecture Notes 3B)

Professor: A. Rodriguez

Chapter 5 Business Rules & Validation

5.1 Business Objects Review & Status

5.1.1 Business Objects Requirements Status

- Ok, let's review where we are as far as Business Objects requirements, and what we have done to implement them.
- The following is a listing of the Business Object's requirements and the status of what we have accomplished:

Business Object Requirements	Status	Comments
Business Object Represents Real-World Business Entities – Business Objects contain the necessary *attributes* & *methods* to behave like their real-world counterparts.	▪ We added private data & properties to emulate real word logic to our objects, for example in the *clsPerson* class we added variables such as Name, Birthday, address, phone etc. Attributes which makes our person objects behave as a person. ▪ In addition we added a Shop() method that emulated the process of a person shopping. Also we implemented clsEmployee Class with Authentication(u,p) method to authenticate employees	▪ DONE
User Interface Support – The Business Objects should contain the following logic to support the User Interface (UI): - Contain all the *features* and *functionality* the UI-Developer will need to make communication between the User & the application effective. - The *Business Objects* are the core of the application and must be designed in a way that is very easy to change the *UI Layer* without risking the business logic stored in the business objects	▪ Begun to design our applications by using the 5-tier Application Architecture. Thus separating the Presentation/User Interface layer from the business processing. ▪ We have placed all processing code in the Business Object Layer. In our examples, all processing code is done within the classes (***clsPerson)*** & collection classes (***lsCustomerManager***).	▪ DONE
Business Object contain all Business Logic & Rules – contain the necessary *Business Logic* & *Rules* to perform their business process & support the data access	▪ Business object has to have the logic and intelligence required to support all the methods and data access. ▪ We need an object that contains logic and automation of functionalities. More on this later	▪ OPEN REQUIREMENT ❖ COVERED IN THIS COURSE
BO Manage their own *data* & *database access* – Business Objects should contain logic to handle data access: - The Business Object should contain all the code to manage the data access or interact with the database. Operations such as *searching*, *inserting*, *updating*, *deleting* the database should be done by the business objects. - Database access should NOT be performed in the User Interface Layer. Only from the Business Object Layer.	▪ These features have not yet been implemented in this course.	▪ OPEN REQUIREMENT ❖ COVERED IN THIS COURSE

Business Object Requirements	Status	Comments
Scalable & Reusable – Business Objects should be design with the following logic: - Can **<u>evolve</u>** & gain new data, properties & methods to support more functionality	▪ Design application using the 5-tier Application Architecture. ▪ Created **Class Library or DLL** to encapsulate our classes, thus enabling them to be placed in the 5-tier Application Architecture ***Business Object & Data Access Business Objects layers***. ▪ Discussed & implemented technologies to implement scalability such as: - DLL	▪ DONE - WE PACKAGED OUR CLASSES IN A **DLL** OR **COMPONENT**
Validation or Enforcement & Status Tracking – Business Objects should contain the following logic: - Business Objects should contain the logic to verify that the data being **<u>set</u>** by the user is valid, correct data type, length etc - The *Business Objects* should be able to keep track of it's current status - The *Business Objects* should be able to keep track of the business rules that are broken. - Business Objects should protect itself from unauthorized or unwanted, harmful access	▪ These feature have not yet been implemented in this course so far.	▪ OPEN REQUIREMENT ❖ LECTURE AVAILABLE IN THE NOTES, BUT WILL BE PARTIALLY COVERED. NO TIME!!!
Distributed Business Objects – Business Objects should be design base with the following network distribution scheme in mind: - Business Objects should contain the technology to allow them to be distributed across processes and application. - Distributed Objects are about sending the object (smart data) from one machine to another, rather than sending raw data and hoping that the business logic on each machine is being kept in sync.	▪ Discussed technologies to implement ***distributed or unanchored objects*** as well as ***non-distributed or anchored objects***. ▪ Implemented ***distributed or unanchored objects*** by using the **`<Serializable()> _`** attribute statement in our ***clsPerson*** Business Object. ▪ Discussed technologies to implement business objects, such as: - DLL - Serialization - Remoting	OPEN REQUIREMENT: - Have not yet implemented ***Serialization*** - Not yet implemented ***Anchored Objects*** - Not yet implemented ***Remoting*** ❖ WILL DISCUSS IN CLASS AND PREPARE FOR IT, BUT NOT IMPLEMENT OR COVERED IN THIS COURSE

5.1.2 Key Technologies to Implement Business Object Review

- We have discussed and address the following key technologies to implement Business Objects:

1. ***Local objects***:
   - Default designation of object when created. Can only be accessed by components within its process (In-Process Communication)
   - Local Classes are NOT available to the technology or **Remoting**, which enables objects to communicate across networks and processes.
   - We have implemented this technology by default since CS608.

2. ***Anchored objects***:
   - These objects are stuck on the process or machine in which they were created and are important, because we can guarantee that they will always run on a specific machine only.
   - Communication with these types of objects is via ***Pass-By-Reference*** or a *pointer* is passed to other processes that wish to communicate with the Anchored Objects.
   - ***Data Access BO Layer*** will be created as Anchored Objects since they need to run on a specific machine with access to the Database Layer.
   - To implement we need to inherit our classes from the **MarshalByRefObject** class as follows.

```
Public Class MyClass

    Inherits MarshalByRefObject

End Class
```

   - Anchored objects are available to the **Remoting** subsystem.

3. ***Unanchored Objects*** or ***Distributed Objects***:
   - Distributed Objects can be passed from one process to another process or from one machine to another, **By-Value**. By value means that a *copy* of the original object is placed on the target machine.
   - The ***Business Objects Layer*** is a candidate as *Distributed* or *Unanchored Objects*.
   - To implement, you need to use the **<Serializable()> _** attribute statement.
   - We begun to implement this feature in our last examples as follows.

```
<Serializable()> _
Public Class clsPerson

End Class
```

   - Unanchored objects are available to the **Remoting** subsystem.

4. The ***Anchored*** and ***Unanchored*** Objects require the following technologies:

   - **Class Library Project (DLL)** – Business Objects need to be packaged as a Class Library or DLL (Dynamic-Link-Library).
   - **Remoting** – .NET Subsystem that handles communication between objects across a Network. Either ***Pass-By-Reference*** or ***Pass-By-Value***.

5.1.3 Business Objects Requirements Overview & Summary

- So far we have made some accomplishments in the pursuit of implementing Business Objects.
- In this lecture notes, we will focus on completing the following requirement of adding validation code and making our business object more intelligent and the object protect itself.
- Because this topic is quite involved, we will keep our implementation basic and limited to only a few rules. We don't have the time in this course to cover many of the required logic.

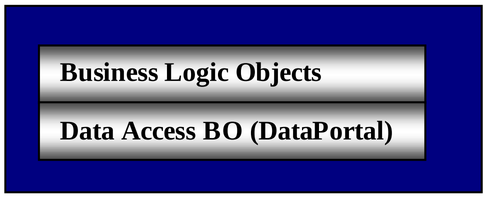
5.2 Business Objects – Data Access Requirements

5.2.1 Overview

- The next requirement we must address is Data Access.
- Since Business Objects need to handle their own Data Access, we will now cover the methods required to do so.

Data Access Objectives:

- Our objectives is to implement the following two layers:



  - The Business Object Layer will contain the business rules
  - The Data Access Business Objects will interact with the data base on our behalf. We will start calling this layer the **DataPortal** Layer.

Implementing the Data Access Objectives:

- It is important to decide where to place the **Data Access code** or ***SQL Statements*** that will ***Load***, ***update***, ***insert*** and ***delete*** the *Objects* to the database.
- These operations are actually performed on the Object's private data. In other words when an Object performs data access, it's actually taking it's private data and saving, updating , inserting or deleting it to the database
- There are several approaches we can take:

  - **METHOD 1:** ***Business Objects that perform Data Access (Execute Queries) themselves:***
    - The *Unanchored* or *Distributed* Business Objects save, update insert & delete themselves to the database.

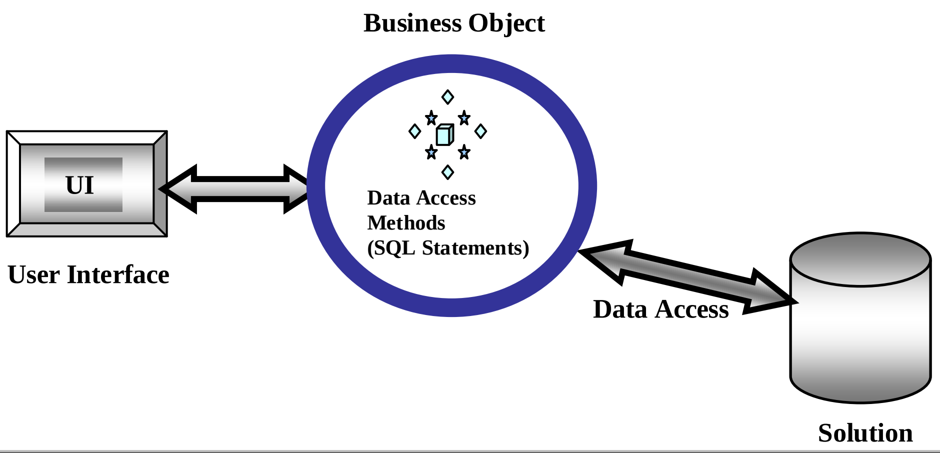
  - **METHOD 2:** ***Specialized Business Objects whose purpose is to Manage the Data Access (Execute Queries) for other objects:***
    - Objects or Business Object rely on another specialize Business Object to manage or save, update insert & delete their data access.
    - You will need one Data Access BO for every type business object.
    - Can be ***Anchored*** objects

  - **METHOD 3:** ***General Purpose DATAPORTAL Object (Not Business Objects) whose purpose is to Manage the Data Access (Execute Queries) for the Business Objects:***
    - Objects or Business Object rely on a *DATAPORTAL* Object(s) which perform and manage or save, update insert & delete.
    - *DATAPORTAL* Object contains all the *SQL statements* to manage the data access for all objects.
    - There may be more than one **Dataportal**, usually one for every type of database we are going to access, *SQL Server*, *Oracle* etc.

  - **METHOD 4:** ***Specialized SERVER-SIDE DATAPORTAL Objects (Not Business Objects) whose purpose is to manage the Business Objects manage and let them do their OWN Data Access:***
    - Unanchored Business Objects are SENT to the *DATAPORTAL* Object(s). The DATAPORTAL simply calls the Business Object's *Data Access methods* so the business Objects will save themselves.
    - *DATAPORTAL* Object contains NO *SQL statements*. The SQL Statements are inside the Business Objects.
    - The Business Objects actually save themselves.
    - This is a new approach that can be implemented due to VB.NET Remoting and serialization techniques.

Method I – Business Objects Perform Their Own Data Access

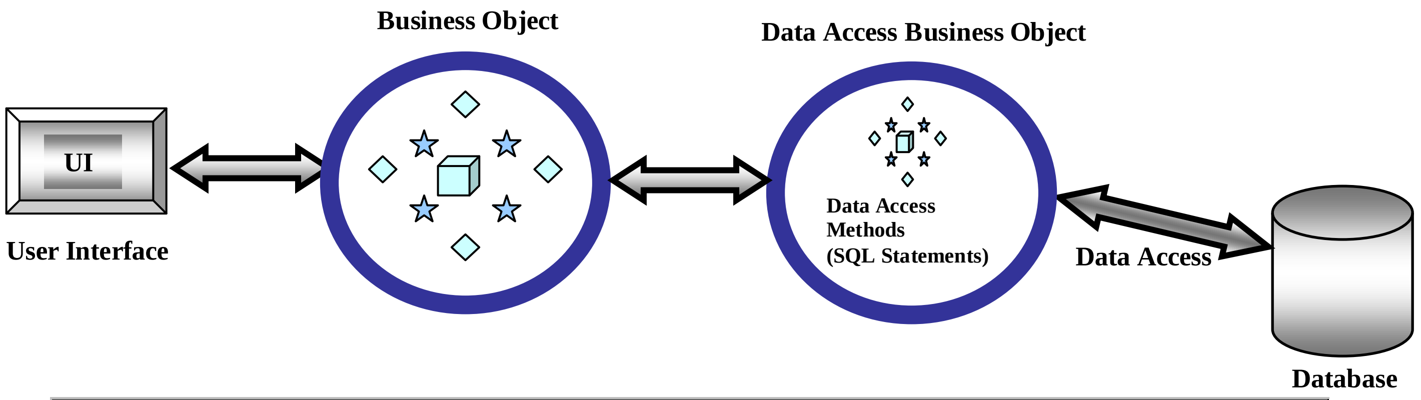
- In this method it is the Business Objects that handle their own data access
- The *Unanchored* or *Distributed* Business Objects save, update insert & delete themselves to the database. They contain the queries and interact with the database:



Advantages/Characteristics	Disadvantages
▪ ***Simple***. BO handle themselves ▪ Object is one package with everything we need, thus we have full encapsulation.	▪ Not scalable for our multi-tiered Client/Server architectures.

Method II – Data Access Business Objects Handle the Data Access

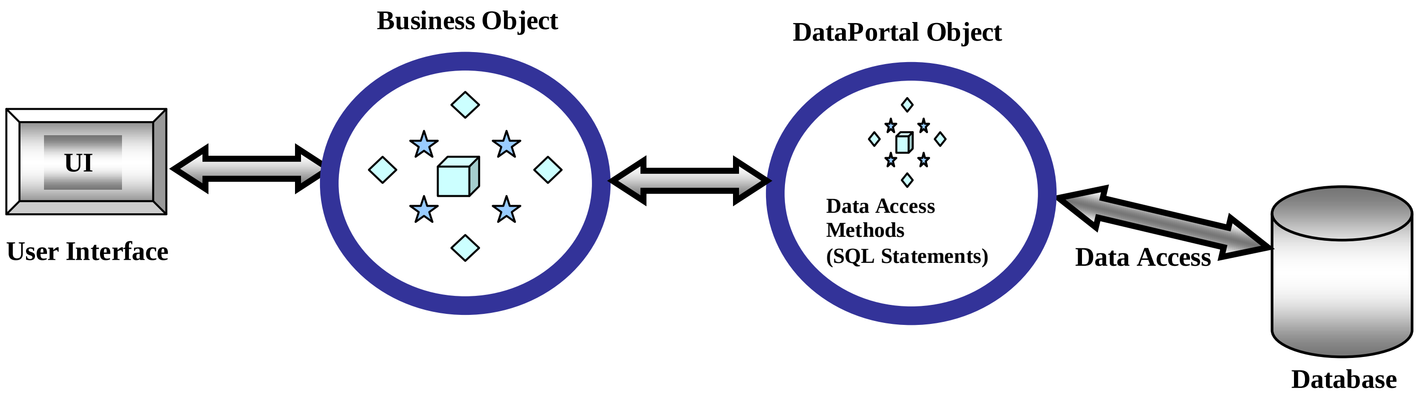
- In this method the Business Object rely on another specialize Business Object to manage or save, update insert & delete their data access
- These Data Access Business Objects can be A*nchored* and contain the SQL Statements or queries and interact with the database:


Business Object
Data Access Business Object
UI
User Interface
Data Access Methods (SQL Statements)
Data Access
Database

Advantages/Characteristics	Disadvantages
▪ Business Objects are light-weight. Less complex since Data Access BOs contain queries ▪ ***Scalable***. Fits our client/server architectures	▪ Object not one single package but broken up into two separate entities. No more full encapsulation ▪ Will need one for every type of business objects ▪ Business Object rely on Data Access BOs

Method III – General Purpose DataPortal Layer Handle the Data Access (Common Practice)

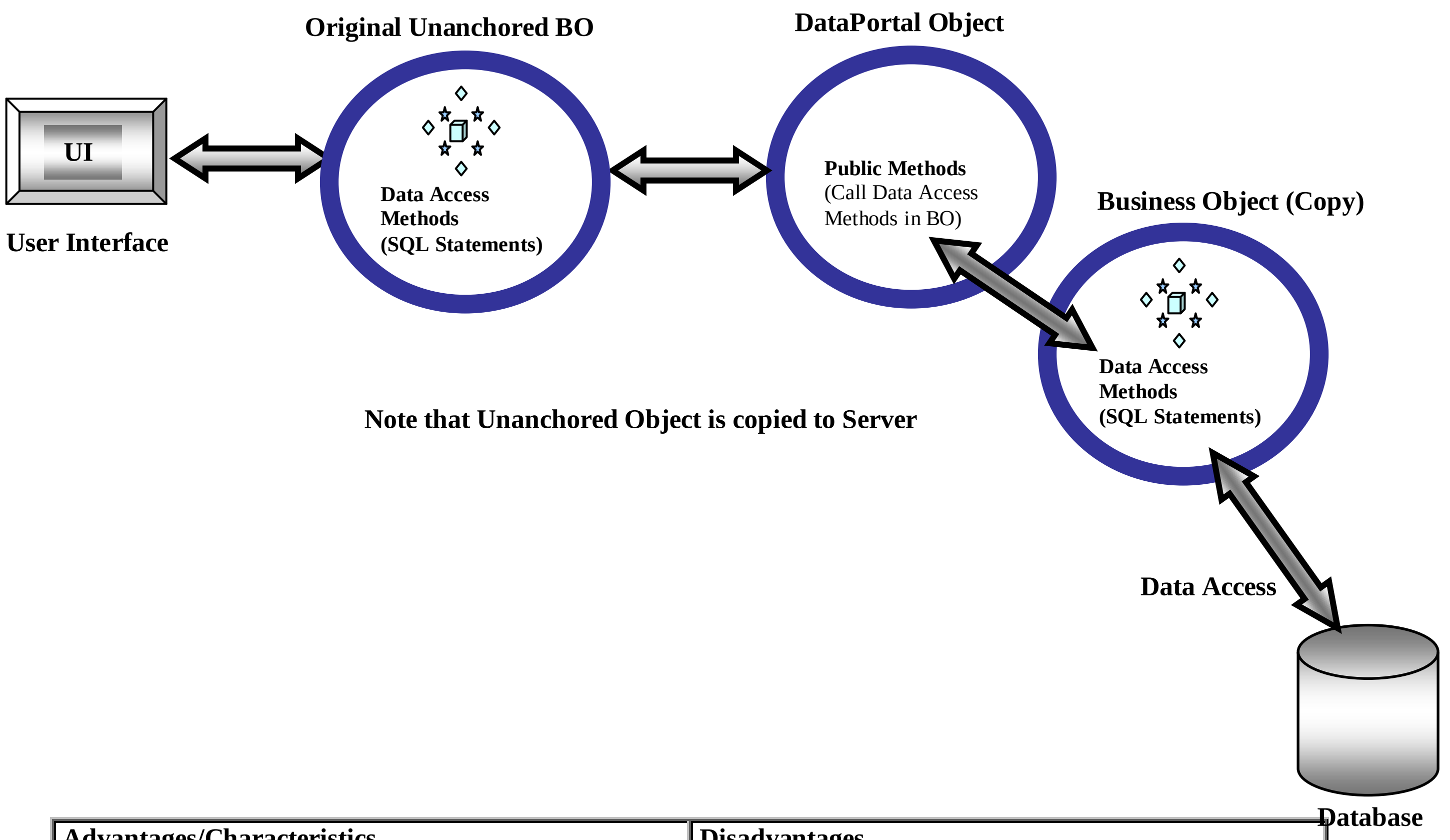
- ❑ In this method the Business Object rely on a general DATAPORTAL Object or Layer to manage or save, update insert & delete their data access
- ❑ The DATAPORTAL is usually ***Anchored*** and contain the SQL Statements or queries and interact with the database:



Advantages/Characteristics	Disadvantages
▪ Business Objects are light-weight. Less complex since DataPortal contains queries ▪ ***<u>Scalable</u>***. Fits our client/server architectures ▪ Object partially a single package and encapsulated ▪ One **DataPortal** for all BO objects. ▪ Could have a **DataPortal** for each type of Database SQL, Oracle etc.	▪ No data access code in Business Objects. ▪ Business Objects will always rely on DataPortal

Method IV – General Purpose Server-Side DataPortal Layer allows Business Objects to Handle the Data Access (New Method – Preferred Method for this Course)

- In this method the Unanchored or Distributed Business Object perform their OWN data access.
- But they rely on a general DATAPORTAL Object or Layer to manage the process by CALLING the Business Objects Data Access Methods on behalf of the Business Objects.
- The key here is that the Business Objects save themselves and contain the SQL Statements or queries and interact with the database, but is the DATAPORTAL that is telling them when and how to do it.
- The DATAPORTAL is ***Anchored*** but the Business Objects must be Unanchored using using .NET technologies such as **Remoting** and **Serialization** etc.


Original Unanchored BO
DataPortal Object
UI
User Interface
Data Access Methods (SQL Statements)
Public Methods (Call Data Access Methods in BO)
Business Object (Copy)
Data Access Methods (SQL Statements)
Note that Unanchored Object is copied to Server
Data Access
Database

Advantages/Characteristics	Disadvantages
▪ Business Objects are a complete single package and contain data access code. ▪ ***Scalable***. Fits our client/server architectures ▪ One **DataPortal** for all BO objects since BO save themselves ▪ Could have a **DataPortal** for each type of Database SQL, Oracle etc ▪ Business Object don't need to always rely on **Dataportal**, they can be configured to save themselves.	▪ Business Objects will always rely on DataPortal ▪ May be more difficult to implement

5.2.2 Implementation Overview

- Base on our discussion of the four methods, in this course we will use the following options:

  - **Option I – Business Object will perform their own data access:**
    - We will use this option for the first semester project and to implement our *Single-Tier Client/Server*

  - **Option IV – DATAPORTAL will manage Data Access but Business Object will perform their own data access:**
    - We will use this option for the to upgrade our semester project and to implement a *three-Tier* & *Web-based Client/Server*

Data Access Methods Details:

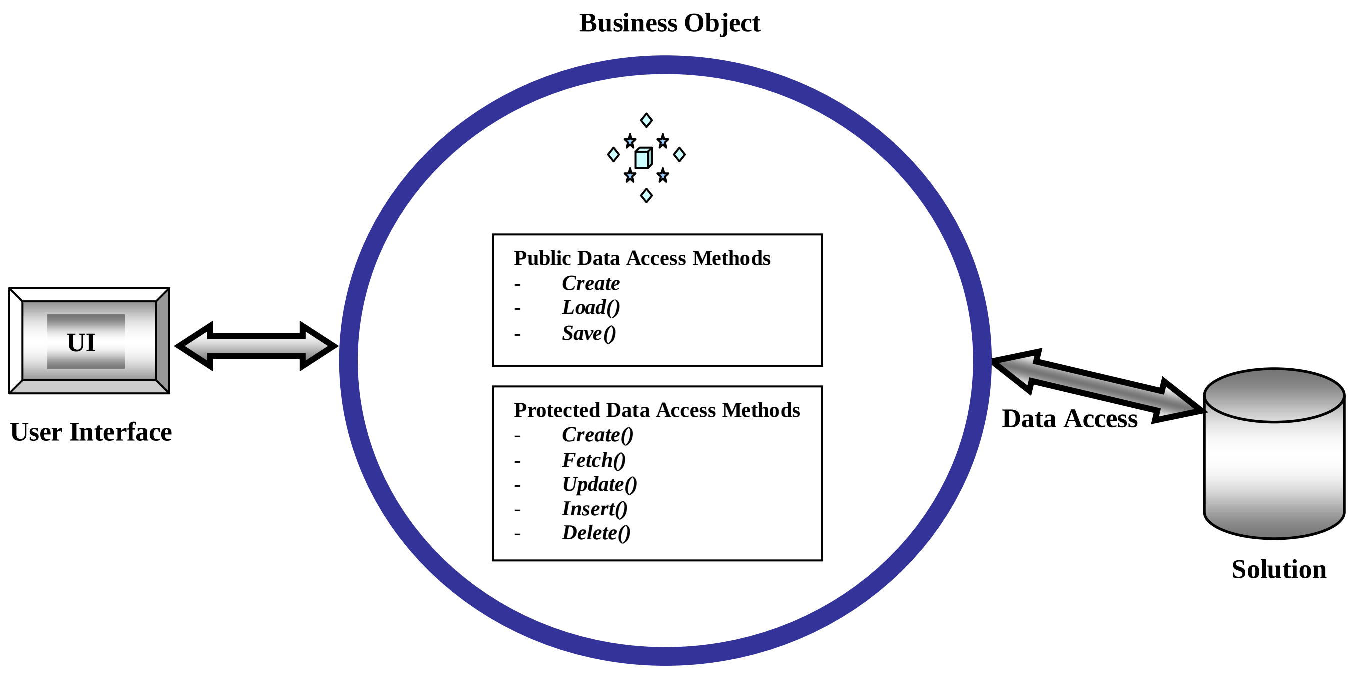
- Since Business Objects need to handle their own Data Access, we will now cover the methods required to do so.
- First we break up the data access methods into two sections, PUBLIC DATA ACCESS METHODS AND PROTECTED OR PRIVATE DATA ACCESS METHODS:

  - ***Public*** **Data Access Methods** – These methods are Public and assessable to the User-Interface or clients. These methods will be declared and implemented in our ***Business Classes***.
    - Note that these methods will be implemented in our ***Business Classes***, and will be forced upon the Business Class by the ***BusinessBase*** class. Therefore these methods will appear in the *BusinessBase* as well as *MustOverride*.

  - ***Protected***, ***Private*** **Data Access methods** – These methods can only be accessed internally within the class and its inherited children. These methods will actually perform the data access and contain the *SQL queries* or *Stored Procedures*. These methods are called by the ***Public Data Access Methods***.

    - Note that these methods will be implemented in our ***Business Classes***, and will be forced upon the Business Class by the ***BusinessBase*** class. Therefore these methods will appear in the *BusinessBase* as well as *MustOverride*.

- The idea here is that there will be data access methods available to the outside world or user interface, and internal private methods that will perform the actual Data Access.



5.3 Creating the Business Logic & Rules. Business Classes, & BusinessBase Class Templates

5.3.1 Overview

Business Rules

- Our focus in this section is to implement all non-data access code required in a Business Object.
- Because of the short time frame for this course, we will only implement a few Business rules.
- In this section we will implement the following business logic:

  - Tracking the Status of an object for NEW, OLD or MODIFIED
    i. Track whether the object is new or has just been created
    ii. Track whether it's data has been changed

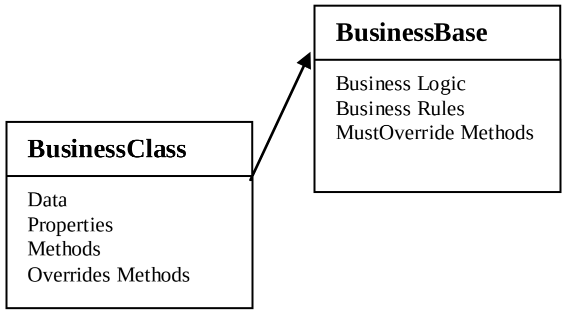
  - Validation – Enforcement of business rules, such data being set by the user is valid, correct data type, length etc

Business Class

- Our Business objects will contain all the business logic and data access code.
- The Business Objects are objects created from a ***Business Class***. So our *Business Classes* are the classes we are going to create for our ***customers***, ***employees***, **autos**, ***videos***, ***checking accounts***, etc.
- All our business classes need to have the mechanism to implement the required logic, data access and rules.

BusinessBase Class

- Our ***Business Classes***, require business rules for tracking, validation and data access etc.
- We need these rules in EVERY BUSINESS CLASS, so what we are going to do is create a **BASE CLASS** that will contain the mechanism to FORCE our classes to implement the business rules and data access methods. We will call this class ***BusinessBase***.
- We will derive our ***Business Classes*** from ***BusinessBase*** in order to inherit all the business rules, tracking etc.
- First thing we need to do is create the **BusinessBase** Class.



5.3.2 Implementing BusinessBase Class

- Our first objective is to create a Base Class named ***BusinessBase***. This Base class will contain all the Business Objects tracking, validation mechanism & logic required. Therefore we can simply derive our classes from this base class and inherit all the Business rules.
- So the code we will implement will be with Inheritance in mind, therefore we will use inheritance concepts like *Overloading*, *Overriding*, *Shadowing* will apply here.
- The first thing we will create is the class header. This class will only be used as a base class so we will use the keyword ***MustInherit***.
- In addition the Business Object will be an unanchored object or distributed so, we will use the **<Serializable()> _** attribute

5.3.3 IMPORT Required Libraries

- Then First thing we need to do is IMPORT ALL THE REQUIRED LIBRARIES. These include the following:

  - ADO.NET Data Access Libraries
  - Serialization Libraries
  - Remoting Libraries
  - Other necessary libraries, for example, I will include the System.IO for any file access I may need in my projects.

```
Option Explicit On
Option Strict On
```

```
Imports System.IO                                        'File/IO
Imports System.Data                                      'Data Access (DataSet)
Imports System.Data.OleDb                                'OLEDB Provider
Imports System.Configuration                             'Configuration File for DB
Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                          'Remoting
'Imports System.Runtime.Remoting.Channels                 'Remoting
'Imports System.Runtime.Remoting.Channels.Http            'Remoting
```

5.5.4 Convert Class into Distribute Object/Unanchored Class

- Now we convert the class into an UNANCHORED CLASS, using the following TAG, just before the class declaration.
- The class header will look as follows:

```
<Serializable()> _
Public MustInherit Class BusinessBasee


End Class
```

5.3.5 Tracking Dirty Object

- We need to keep track if the object has changed. If so, we will designate this object as being DIRTY.
- A DIRTY object has the following definition:

    - **A DIRTY Object is an Object whose data or private variables have been modified**.
        - When an object changes it means that any of the data or private variables have been modified thus DIRTY.

    - **BECAUSE IT HAS BEEN CHANGED, A DIRTY Object does not match the data in the DATABASE**.
        - It is important that you understand this concept clearly. When we refer to an object being dirty, we not only mean that the data has changed, but that it has changed in reference to its copy in the database.
        - Think of it this way, when we populate an object with data from the database, once we change the data in the object, the data no longer is the same as the data in the database. Therefore, the object is dirty and does not represent what is in the database

    - HERE IS THE MAIN POINT, if the object is dirty, then we need to perform some kind of **UPDATE** or **INSERT** operation on the database base on this status. This will also be determined by the NEW Rule which will be explained in the next section.

Implementing Dirty Object

- To implement dirty objects, declare:

I. Declare the Boolean flag **flgDirty**, which when TRUE the object is dirty, when FALSE, the object is not dirty. .
   - This Boolean flag will be set to TRUE by default. This makes sense since when an object is created it does NOT match any data in the database therefore it is dirty by our definition

```
Private mflgIsDirty As Boolean = True
```

II. We need to expose the dirty flag to the User-Interface & Business Logic to be able to *retrieve* this value ONLY. We will declare a READ-ONLY *Property IsDirty()*. We will make the property ***Overridable*** reason being that derived objects of this class may want to override this property for special situation (more on this later). Create the property as follows.

```
    Public Overridable ReadOnly Property IsDirty() As Boolean
        Get
            Return mflgIsDirty
        End Get
    End Property
```

III. This flag also needs to be set by the Business Object. But the code outside the class should NOT be able to alter this flag. On the other hand, derived children should be able to set this flag. With this in mind, we will implement a ***Protected*** Method named ***MarkDirty()*** as follows:

```
    Protected Sub MarkDirty()
        mflgIsDirty = True
    End Sub
```

❖ **IMPORTANT!** – ANY PROPERTY (SET portion only) OR METHOD WHICH ***MODIFIES*** DATA MUST CONTAIN A CALL TO **MarkDirty()**

IV. Finally we need a way to mark the object as clean when the data is saved or updated to the database. This is done by implementing a ***Private*** Method named ***MarkClean()***. This sub procedure is created private because it will only be called from within the Base Class, no need for child object to have access to this method. In the mean time, this method is implemented as follows:

```
    Private Sub MarkClean()
        mflgIsDirty = False
    End Sub
```

5.3.6 Tracking New Object

- The third status-tracking mechanism we will implement is the concept of a NEW object
- A new object has the following definition:

  - **A NEW Object is an Object that was just created and in Local MEMORY ONLY**.
    - Every time we create an object using the basic object creation syntax, the object is NEW
    - The following creation of an object signifies that this object is NEW:

```
        Public objCustomer As New clsCustomer
```

  - **A NEW Object exists in memory but NOT in DATABASE**.
    - Objects that are newly created, exists in the memory of the computer, but have not been committed to database. They DO NOT exist in the database therefore are NEW.

  - **A NEW Object is also DIRTY since Data in the Object does not match ANY Data in the DATABASE**
    - A new object is marked dirty because it does not match any data in the database, therefore when committing an object we can determine whether to perform an **UPDATE** or **INSERT** operation on the database base on the **IRTY** status.
  - **An OLD Object is an object committed to DATABASE** and no longer **NEW**
    - Once a NEW object has been SAVED or **INSERT** to Database it is classified as **OLD**.

- We now implement a mechanism that will track if the object is NEW (not in database) or OLD (exists in database).

Implementing NEW Objects

- To implement the NEW object feature perform the following:

I. Declare the Boolean flag **flgIsNew**. When TRUE, this flag indicates Object has just been created or does not exist in the database. A FALSE indicates object already contains a record in the database.
   - This Boolean flag is set to TRUE by default. When an object is created it is NEW since NOT EXIST in database.

```
Private mflgIsNew As Boolean = True
```

II. We need to expose the deleted flag to the User-Interface & Business Logic to be able to *retrieve* this value ONLY. We will declare a READ-ONLY *Property IsNew()*. The property is declared as follows:

```
    Public ReadOnly Property IsNew() As Boolean
        Get
            Return mflgIsNew
        End Get
    End Property
```

III. This flag also needs to be set by the Business Object logic. In addition, derived children should be able to set this flag. Note that when we mark an object as NEW, we need to set the **IsDirty** flag to True by calling the *MarkDirty()* method, because the data in a new object does not match any data in the database, therefore it is dirty. With this in mind, we will implement a ***Protected*** Method named ***MarkNew()*** as follows:

```
    Protected Sub MarkNew()
        mflgIsNew = True
       MarkDirty()
    End Sub
```

IV. Finally, we need to provide a method that will mark the object as OLD, to indicate the object has been SAVED to DATABASE. Therefore we implement a ***MarkOld()*** method. We also need to set the dirty flag to True, indicating that the data in the object matches the data in the database. This is implemented as follows:

```
    Protected Sub MarkOld()
        mflgIsNew = False
       MarkClean()
    End Sub
```

5.3.7 Adding Tracking Mechanism to BusinessBase Class

- So far we have implemented the following basic required tracking mechanism that will add to a Business Base Class:
  - Track whether the object is NEW & DIRTY.

- At this point, the ***BusinessBase*** Class looks as follows:

```
<Serializable()> _
Public MustInherit Class BusinessBase

#Region "Business Rules IsNew, IsDirty"
    Private mflgIsDirty As Boolean = True
    Private mflgIsNew As Boolean = True

    Public ReadOnly Property IsNew() As Boolean
        Get
            Return mflgIsNew
        End Get
    End Property

    Public Overridable ReadOnly Property IsDirty() As Boolean
        Get
            Return mflgIsDirty
        End Get
    End Property

    Protected Sub MarkDirty()
        mflgIsDirty = True
    End Sub

    Private Sub MarkClean()
        mflgIsDirty = False
    End Sub

   Protected Sub MarkNew()
        mflgIsNew = True
        MarkDirty()
    End Sub

  Protected Sub MarkOld()
        mflgIsNew = False
        MarkClean()
    End Sub

#End Region
End Class
```

5.3.8 MustOverride Data Access Methods – Declared in BusinessBase

- Now we address the data access code that will perform the actual database retrieval, update, insert or delete.
- These methods will be declared in the ***BusinessBase*** Class and FORCED upon the ***Business Class***.

BusinesBase & MustOverride

- Again we will declare these methods **MustOverride** in our ***BusinessBase*** class, thus forcing the derived classes (***Business Class***) to have to implement them.
- THESE METHODS ARE NOT IMPLEMENTED IN BUSINESSBASE, BUT ONLY DECLARED MUSTOVERRIDE. THEY MUST BE IMPLEMENTED IN THE DERIVED CLASSES.
- With this in mind, we will ONLY declare these methods in the Base Class as ***MustOverride*** methods. If you remember in inheritance a ***MustOverride*** method is declared in the Base class, but MUST be implemented in the derived class. The derived class MUST implement this method otherwise you cannot compile the application.

Declaring Public & Protected Data Access Methods in BusinessBase

- To implement these methods we make the following declarations in the ***BusinessBase*** Class:

I. **Public MustOverride Create(), Load(Key), Save() & DeleteObject** – These methods are *MustOverride*, therefore CANNOT be implemented in the Base Class, but the derived class will be FORCED to implement tem. Declare methods here as follows:

```
    'Public Shared Data Access Methods Declarations
    ''' Override these Public Methods in SubClass to perform Data Access
    ''' These methods are the public interface provided by the class
    ''' for data access
    Public MustOverride Sub Create()
    Public MustOverride Sub Load(ByVal Key As Object)
    Public MustOverride Sub Save()
    Public MustOverride Sub DeleteObject(ByVal Key As Object)
```

II. **Protected MustOverride DataAccess Methods** – These methods are *MustOverride*, therefore CANNOT be implemented in the Base Class, but the derived class will be FORCED to implement them. Implement this method as follows:

```
    ''' Override these methods in SubClass or Business Classes to
    ''' actually perform data access. SQL Queries & Stored Procedures
    ''' are handled by these methods
    Protected MustOverride Sub DataPortal_Create()
    Protected MustOverride Sub DataPortal_Fetch(ByVal Key As Object)
    Protected MustOverride Sub DataPortal_Update()
    Protected MustOverride Sub DataPortal_Insert()
    Protected MustOverride Sub DataPortal_DeleteObject(ByVal Key As Object)
```

5.3.8 Other Data Access Helper Methods (BusinessBase)

- When performing data access from a database, we need to establish a DATABASE CONNECTION.
- This connection, also know as a CONNECTION STRING.
- There are several options to creating a connection string. I will show three:

  - **METHOD 1:** ***Create or hard-code Connection String inside class either in BusinessBase or Business Class itself:***

    Advantage:
    - Simple and effective.
    - Objects don't need to go anywhere to get the connection string; it is available inside the object.
    - Secured. No one can see connection string, since it is compiled within the class

    Disadvantage:
    - Creating the connection string inside the class is perfectly fine, but what happens if we change database? Now we need to go inside each class and make the change manually.
    - Thus Difficult to maintain and update. Must recompile program.

  - **METHOD 1I:** ***Create or hard-code Connection String in an external Configuration File*** – All objects can retrieve the connection string from one file.

    Advantage:
    - Easy to create and write
    - Central location where string could be found. Can be XML file
    - Easy to maintain and change. Change one location, all objects get the change.

    Disadvantage:
    - Not Secured. Configuration file is a text file and can be seen by anyone with access to server
    - You may be able to encrypt the file, but would need encryption/decryption code inside every business object.

  - **METHOD 1II:** ***Create or hard-code Connection String in the Computer Registry*** – All objects can retrieve the connection string from the registry.

    Advantage:
    - Central location where string could be found.
    - Easy to maintain and change. Change one location, all objects get the change.

    Disadvantage:
    - Must create code to write to Registry the connection string details.
    - Not secured. Not as available as configuration file, but registry can still be read
    - You can encrypt the entries in registry, but would need encryption/decryption code inside every business object.

- We will implement both Method I & II.
- We will hard-code the connection string the class for the FIRST DATA ACCESS EXAMPLE, then implement method II for the SECOND DATA ACCESS SAMPLE PROJECT.
- Nevertheless, we will prepare our **BusinessBase** Class to contain code for implementing METHOD II or reading from configuration file.
- In the ***BusinessBase*** Class add the following code:

I. **Protected Function DBConnectionString()** – This is where the code and queries or stored procedures are listed for creating new objects and populating them with data from database:

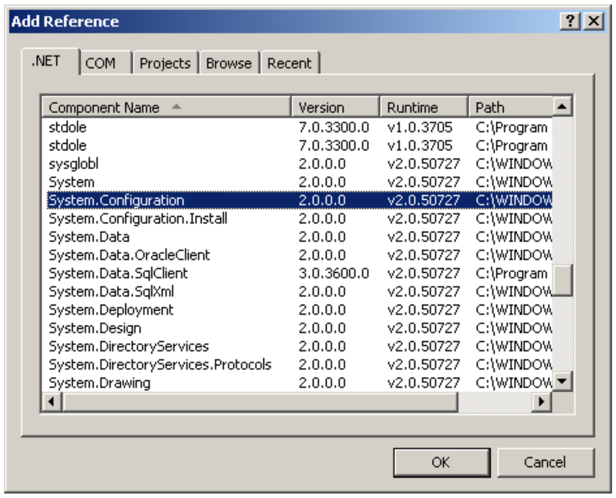
```
'Method will return the Database Connection string from Configuration File
'Assumes the database name is prefixed with "DB"
Protected Function DBConnectionString(ByVal sDatabaseName As String) As String
    Return ConfigurationManager.AppSettings("DB:" & sDatabaseName)
End Function
```

II. Imported Library – In order for this to work, we need to perform the following steps:

1. Add reference to the System.Configuration Library

a. In solution explorer, SELECT the **Project**, then RIGHT-CLICK, select ***Add Reference…*** you will invoke the ***"Add Reference"*** dialog box

b. Select the .NET tab and scroll and select the ***System Configuration*** Library, then click ***OK*** :



Component Name	Version	Runtime	Path
stdole	7.0.3300.0	v1.0.3705	C:\Program
stdole	7.0.3300.0	v1.0.3705	C:\Program
sysglobl	2.0.0.0	v2.0.50727	C:\WINDOW
System	2.0.0.0	v2.0.50727	C:\WINDOW
System.Configuration	2.0.0.0	v2.0.50727	C:\WINDOW
System.Configuration.Install	2.0.0.0	v2.0.50727	C:\WINDOW
System.Data	2.0.0.0	v2.0.50727	C:\WINDOW
System.Data.OracleClient	2.0.0.0	v2.0.50727	C:\WINDOW
System.Data.SqlClient	3.0.3600.0	v2.0.50727	C:\Program
System.Data.SqlXml	2.0.0.0	v2.0.50727	C:\WINDOW
System.Deployment	2.0.0.0	v2.0.50727	C:\WINDOW
System.Design	2.0.0.0	v2.0.50727	C:\WINDOW
System.DirectoryServices	2.0.0.0	v2.0.50727	C:\WINDOW
System.DirectoryServices.Protocols	2.0.0.0	v2.0.50727	C:\WINDOW
System.Drawing	2.0.0.0	v2.0.50727	C:\WINDOW

2. Import the System.Configuration Library into your code:

```
Imports System.Configuration      'Configuration File for DB Connection
```

5.4 Business Base Template – Putting the Base Class Together

5.4.1 Implementing Business Base

- Now we at will put all the code together to create our ***BusinessBase*** Class.
- This is the Base Class we will used to derive all our ***Business Classes*** from.
- Keep in mind that this in ONLY a base class, we still need to create the Business Classes (employees, customers, etc.) that will be used to create the Business Objects themselves.
- So far we have implemented the following basic required tracking mechanism that will add to a ***BusinessBase*** Class:

  - Track whether the object is new or has just been created or NEW
    - Private ***flgIsNew***, Property ***IsNew*** & ***MarkNew()***, ***MarkOld()*** Methods

  - Track whether it's data has been changed or DIRTY
    - Private ***flgIsDirty***, Property ***IsDirty*** & ***MarkDirty()***, ***MarkClean()*** Methods

- In addition we need to DECLARE ONLY the <u>***MustOverride***</u> protected Data Access methods that we are imposing upon our derived classes or children:

  - **Public MustOverride Create()**
  - **Public MustOverride Load()**
  - **Public MustOverride Save()**
  - **Public MustOverride DeleteObject()**

  - **Protected MustOverride DataPorta_Create()**
  - **Protected MustOverride DataPorta_Fetch()**
  - **Protected MustOverride DataPorta_Update()**
  - **Protected MustOverride DataPorta_Insert()**
  - **Protected MustOverride DataPorta_DeleteObject()**
  - **Protected Function DBConnectionString()**

- In addition, there are .NET namespace libraries which must be included for these mechanisms to work. Therefore we will add the required libraries for the following:

  - **ADO.NET Library**
    I/O Library for any file access requirements
  - **Configuration File library to use and manage configuration files storing our connection strings**
  - **Remoting Libraries**
  - **Serialization Library**

5.4.2 Sample Program #1 – Creating the BusinessBase Class

- We now implement the ***BusinessBase*** class that will server as the basis for creating the ***Bussiness Classes***

Example 5.1 – Creating a BusinessBase Class

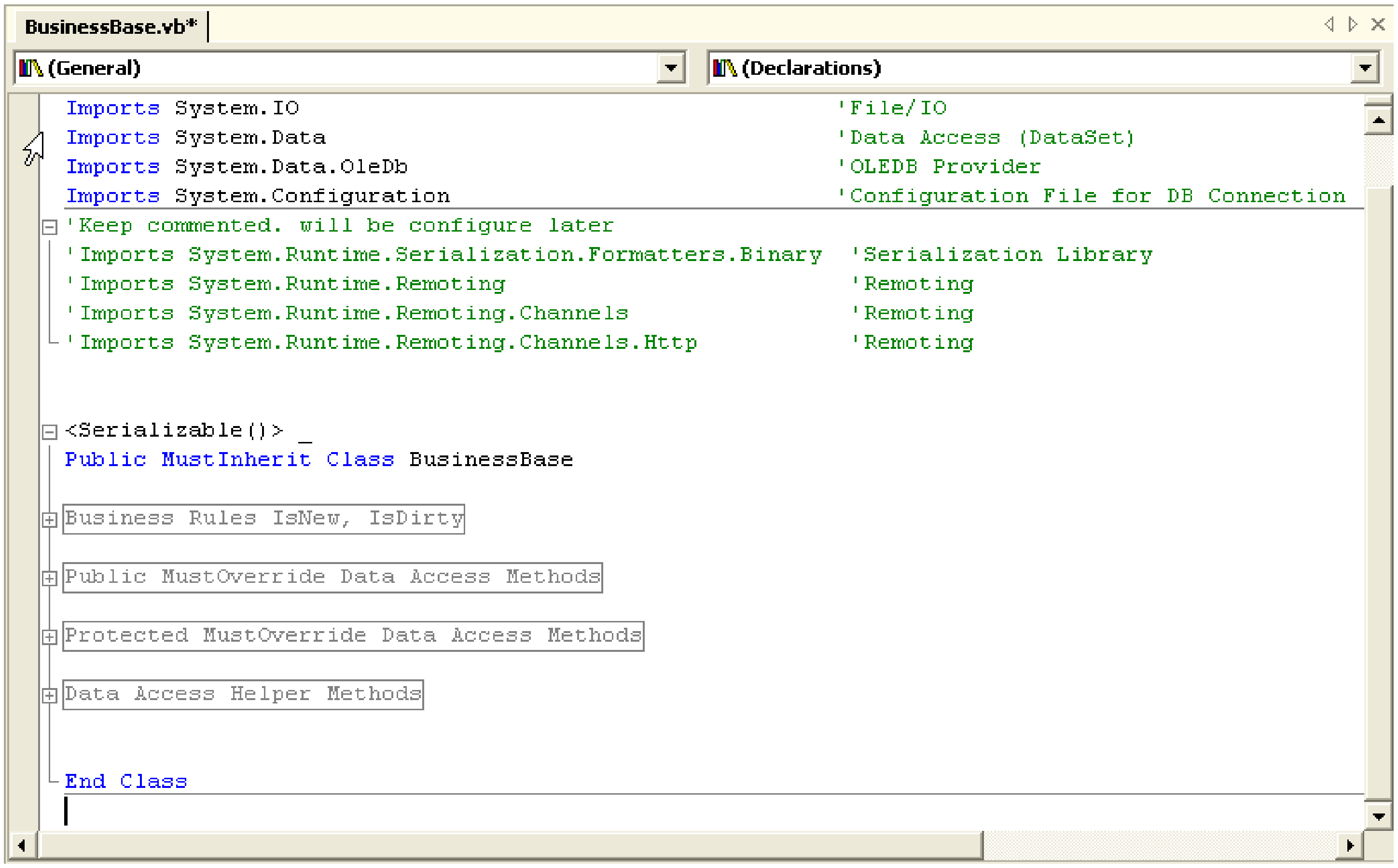
Problem statement:

- Create the ***BusinessBase*** class using all the rules covered in the lecture.

Business Object Layer – Business Class & DLL Requirements

- Implement the ***BusinessBase*** in a DLL project
- We now go through the steps of creating our **BusinessBase** class.
- The diagram below shows the **Regions** that make up the **BusinessBase** Class Format.

Region view of BusinessBase Format



```
Imports System.IO                                         'File/IO
Imports System.Data                                       'Data Access (DataSet)
Imports System.Data.OleDb                                 'OLEDB Provider
Imports System.Configuration                              'Configuration File for DB Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary   'Serialization Library
'Imports System.Runtime.Remoting                          'Remoting
'Imports System.Runtime.Remoting.Channels                 'Remoting
'Imports System.Runtime.Remoting.Channels.Http            'Remoting


<Serializable()> _
Public MustInherit Class BusinessBase

Business Rules IsNew, IsDirty

Public MustOverride Data Access Methods

Protected MustOverride Data Access Methods

Data Access Helper Methods


End Class
```

Step 0: Create an Empty Solution and do the following:

1. Create Empty Solution
2. Create a Class Library Project
3. Add a Class, name it **BusinessBase**

Step 1: Imports and Class declarations:

- At this point, the ***BusinessBase*** Class looks as follows. Library declarations, Unanchored Object & Class declaration:

```
Imports System.IO                                              'File/IO
Imports System.Data                                            'Data Access (DataSet)
Imports System.Data.OleDb                                      'OLEDB Provider
Imports System.Configuration                                   'Config File DB Connection

'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary         'Serialization Library
'Imports System.Runtime.Remoting                                'Remoting
'Imports System.Runtime.Remoting.Channels                       'Remoting
'Imports System.Runtime.Remoting.Channels.Http                  'Remoting

<Serializable()> _
Public MustInherit Class BusinessBase
```

Step 2: Add the Business Rules Data, Property Procedures & Methods:

- Dirty and New mechanism:

```
#Region "Business Rules IsNew, IsDirty"

    Private mflgIsDirty As Boolean = True
    Private mflgIsNew As Boolean = True


      Public ReadOnly Property IsNew() As Boolean
        Get
            Return mflgIsNew
        End Get
    End Property

    Public Overridable ReadOnly Property IsDirty() As Boolean
        Get
            Return mflgIsDirty
        End Get
    End Property

    Protected Sub MarkDirty()
        mflgIsDirty = True
    End Sub

    Private Sub MarkClean()
        mflgIsDirty = False
    End Sub

    Protected Sub MarkNew()
        mflgIsNew = True
        MarkDirty()
    End Sub

    Protected Sub MarkOld()
        mflgIsNew = False
        MarkClean()
    End Sub
```

Step 3: Add the Public Data Access MustOverride Method Declarations:

- Protected Data Access declarations:

```
#Region "Public MustOverride Data Access Methods"
    '*********************************************************************
    'Public Shared Data Access Methods Declarations
    ''' Override these Public Methods in SubClass to perform Data Access
    ''' These methods are the public interface provided by the class
    ''' for data access
    Public MustOverride Sub Create()
    Public MustOverride Sub Load(ByVal Key As Object)
    Public MustOverride Sub Save()
    Public MustOverride Sub DeleteObject(ByVal Key As Object)

#End Region
```

Step 4: Add the Protected Data Access MustOverride Method Declarations:

- Protected Data Access declarations:

```
#Region "Protected MustOverride Data Access Methods"

    ''' Override these methods in SubClass or Business Classes to
    ''' actually perform data access. SQL Queries & Stored Procedures
    ''' are handled by these methods
    Protected MustOverride Sub DataPortal_Create()
    Protected MustOverride Sub DataPortal_Fetch(ByVal Key As Object)
    Protected MustOverride Sub DataPortal_Update()
    Protected MustOverride Sub DataPortal_Insert()
    Protected MustOverride Sub DataPortal_DeleteObject(ByVal Key As Object)
#End Region
```

Step 5: Add the Data Access Helper Methods Declarations:

- Helper method to allow CONNECTION STRING IN CONFIGURATION FILE.
- **IMPORTANT!** DON'T FORGET TO ADD A REFERENCED TO THE ***System.Configuration*** LIBRARY as follows:
  a. In Solution Explorer SELECT & RIGHT-CLICK PROJECT, in drop-down menu, select ***Add Reference.***
  b. In the reference DIALOG BOX, select .NET TAB,
  c. Scroll and select ***System.Configuration*** library, the click OK.

```
#Region "Data Access Helper Methods"

    'Method will return the Database Connection string from Configuration File
    'Assumes the database name is prefixed with "DB"
    Protected Function DBConnectionString(ByVal sDatabaseName As String) As String
        Return ConfigurationManager.AppSettings("DB:" & sDatabaseName)
    End Function

#End Region
```

5.5 Creating our Business Classes – Business Class Template

5.5.1 Implementing Business Class

- Now we focus on our ***Business Class***. REMEMBER THIS IS THE CLASS IN WHICH WE WILL BUILD OUR OBJECTS FROM (clsCustomer, clsEmployee etc.)
- DO NOT CONFUSE THE BUSINESS CLASS WITH BUSINESS OBJECTS, BUSINESS OBJECTS ARE INSTANCE OF A BUSINESS CLASS!
- OBJECT-ORIENTED RULES REVISED:

  I. CREATE BUSINESS CLASS
  II. CREATE BUSINESS OBJECT
  III. USE BUSINESS OBJECT

- We will now CREATE A BUSINESS CLASS TEMPLATE, that we can use to guide us in creating or modifying our classes.
- This class will be inherited from ***BusinessBase*** thus FORCING the business rules and data access methods upon the BUSINESS CLASS

5.5.2 IMPORT Required Libraries

- Then First thing we need to do is IMPORT ALL THE REQUIRED LIBRARIES. These include the following:

  - ADO.NET Data Access Libraries
  - Serialization Libraries
  - Remoting Libraries
  - Other necessary libraries, for example, I will include the System.IO for any file access I may need in my projects.

```
Option Explicit On
Option Strict On

Imports System.IO                                          'File/IO
Imports System.Data                                        'Data Access (DataSet)
Imports System.Data.OleDb                                  'OLEDB Provider
Imports System.Configuration                               'Configuration File for DB
Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary    'Serialization Library
'Imports System.Runtime.Remoting                           'Remoting
'Imports System.Runtime.Remoting.Channels                  'Remoting
'Imports System.Runtime.Remoting.Channels.Http             'Remoting
```

5.5.3 Convert Class into Distribute Object/Unanchored Class & Inherit from BusinessBase Class

- Now we convert the class into an UNANCHORED CLASS, using the following TAG, just before the class declaration:

```
<Serializable()> _
Public Class BusinessClassTemplate
```

- Then we inherit this class from BUSINESSBASE class, we can INHERIT THE BUSINESS RULES and METHOD forced upon the BUSINESS CLASS by BUSINESS BASE.

```
<Serializable()> _
Public Class BusinessClassTemplate
    Inherits BusinessBase  'Inherits from BusinessBase.
```

5.5.4 Implementing Data, Properties, Methods and Events

- Nothing new here. These are the components every class should have, private data, public properties, public methods etc.
- In addition, we have the default and parameterized constructors.

5.5.5 Public Data Access Methods – Forced Upon by BusinessBase

- We will look at the four Public Data Access Methods created in the ***Business Classes*** to be called by the **User-Interface**:

Public Data Access Methods Implementation Details

- We will implement these Public Data Access Methods in our **Business Classes**. Our Business Classes are the Business Objects themselves such as customers, videos, cars, employees etc.
- To implement we must use the keyword ***Overrides*** since they are FORCED by ***BusinessBase***
- In our Business Classes, we will use **ADO.NET** to implement our data access. In order to user **ADO.NET** we need to add the namespace libraries as follows:

```
Imports System.Data
Imports System.Data.OleDb  'Data Access Library OLEDB Provider
```

Implementing Create(), Load(key) and Save()

- NOTE THA THE IMPLEMENTATIONS SHOWN HERE ARE EXAMPLES ONLY. THERE ARE MANY WAYS TO DO THIS. YOU ARE WELCOME TO READ OTHER MATERIAL AND LEARN HOW IS DONE BY OTHER DEVELOPERS AND AUTHORS. To implement these methods we make the following declarations in the ***Business Class***:

I. **Public Sub Create()**

- Declared Override since it is forced by Base Class
- This method is OPTIONAL and we may not implement, but we will have it for future use. Most times, objects need to be created with default data from the DATABASE. If this is the case, the Business Object needs to be created by the DATAPORTAL Server and populated with data from the database and returned to the client for use. This method calls the ***Protected DataPortal_Create()*** method that will contain the necessary code to create the object and populated with defaults from the database

```
'Public interface to Create objects from database
Public Overrides Sub Create()
    DataPortal_Create()
End Sub
```

II. **Public Sub Load(key)** – Implement this method as follows:

- This method is labeled as Overrides, since we are forced to override the base class
- Calls the ***Protected DataPortal_Fetch(*****ByVal** *Key* **As** *Object***)** method to LOAD data from database
- The argument to this method **Load(***Key* **As** *Object***)** represents the database key and it is of type *Object*, which means that we can pass any object type as argument, string, customers, cars, videos etc

```
Public Overrides Sub Load(ByVal Key As Object)
    'Calls Local DatPortal_Fetch(Key) To do the work
    DataPortal_Fetch(Key)

End Sub
```

III. **Public Shared Sub Delete()** – Implement this method as follows:

```
Public Overrides Sub DeleteObject(ByVal Key As Object)
    'Calls Local DatPortal_DeleteObject() To do the work
    DataPortal_DeleteObject(Key)
End Sub
```

- This method is labeled as Overrides, since we are forced to override the base class
- Calls the ***Protected DataPortal_DeleteObject(*****ByVal** *Key* **As** *Object***)** method to DELETE object from database
- The argument to this method **Load(***Key* **As** *Object***)** represents the database key and it is of type *Object*, which means that we can pass any object type as argument, string, customers, cars, videos etc

IV. Public Function Save() – Implement this method as follows:

- This method is labeled as Overrides, since we are forced to override the base class
- Note that the decision to perform and update or insert is based on the status of the DIRTY & NEW flags
- Calls the ***Protected DataPortal_Insert()*** or ***Protected DataPortal_Insert()*** methods based on the status of the Dirty and New flags

```
    Public Overrides Sub Save()
        'Only save if dirty, otherwise do nothing in this method
        If Me.IsDirty Then
            If Me.IsNew Then
                'We are new and being inserted
                'Calls Local DataPortal_Insert()
                DataPortal_Insert()
            Else
                'We are OLD so we are being updated
                'Calls Local DataPortal_Update()
                DataPortal_Update()
            End If
        End If

    End Sub
```

5.5.6 Protected Data Access Methods – Implemented in Business Class

- Now we look at the implementation of the protected DATA ACCESS METHODS in the Business Classes.
- Since we declared these methods in the ***BusinessBase*** Class as ***MustOverride*** methods, we are forced to implement them here otherwise we cannot compile our class. To implement we must use the keyword ***Overrides***.
- There are two requirements for implementing the Data Access Methods:

1. Implement the **ADO.NET** Code to perform the Data Access

   - SINCE WE ARE NOT COVERING DATA ACCESS USING ADO.NET AT THIS TIME. I WILL NOT SHOW THE ACTUAL CODE HERE
   - THIS WILL BE DONE IN THE NEXT LECTURE NOTES

2. Incorporate the ***DIRTY*** & ***NEW*** mechanism in the Data Access Methods for database operations such as loading records (**SELECT**), inserting record (**INSERT**), updating records (**UPDATE**), deleting records (**DELETE**) and finally in some circumstances we create an object with default data from database (**CREATE**).

   - The logic is as follows:

     - **CREATE:**
       - MARKS OBJECT AS ***NEW***, WHEN CREATING A NEW OBJECT WITH DEFAULT DATA FROM DB

     - **SELECT:**
       - MARKS OBJECT AS ***OLD***, AFTER RETRIEVING RECORDS FROM DB

     - **INSERT:**
       - ONLY PERFORMED WHEN OBJECT IS ***DIRTY*** & ***NEW***.
       - MARKS OBJECT AS ***OLD*** AFTER INSERT

     - **UPDATE:**
       - ONLY PERFORMED WHEN OBJECT IS ***DIRTY*** & ***OLD***.
       - MARKS OBJECT AS ***OLD*** AFTER UPDATE

     - **DELETE:**
       - MARKS OBJECT AS ***NEW***, AFTER DELETE SINCE OBJECT DOES NOT EXIST IN DB.

- To implement these methods we make the following declarations in the ***Business Class***:

I. **Protected Overrides DataPortal_Create()** – This is where the code and queries or stored procedures are listed for creating new objects and populating them with data from database:
   - A business rule is applied that set the object as a NEW object since it was just created.

```
'Data Access Code for Creating a New Business Object
Protected Overrides Sub DataPortal_Create()
    'Create object and assign default values from database etc.

    'ADD DATA ACCESS CODE HERE USING ADO.NET

    'At the end, set New flag to True a new object is created
    MyBase.MarkNew()
End Sub
```

II. **Protected Overrides DataPortal_Fetch(**key **As** Object**)** – This is where the queries or stored procedures are listed for fetching an object from database base on the parameter key:

- A business rule is applied that set the object as a OLD object since it was just retrieved from database, thus it exists and is old.

```
    'Data Access Code to fetch an object from Database
    Protected Overrides Sub DataPortal_Fetch(ByVal Key As Object)
        'ADO.NET Queries for Fetching (Select/From/Where) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'At the end, set New flag to False.  NOT Dirty since found in database
        MyBase.MarkOld()
    End Sub
```

III. **Protected Overrides DataPortal_Update()** – This is where the queries or stored procedures are listed for updating an object’s data to database:

- A business rule is applied that set the object as a OLD object since it was just updated to database, thus it exists and is old.

```
    'Data Access Code to Update an Objects data to database
    Protected Overrides Sub DataPortal_Update()
        'ADO.NET Queries for updating (Update/Set/Where) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'Set New flag to False since exist in database/and is Not dirty any longer
        MyBase.MarkOld()
    End Sub
```

IV. **Protected Overrides DataPortal_Insert()** – This is where the queries or stored procedures are listed for inserting new objects to database:

- A business rule is applied that set the object as a OLD object since it was just INSERTED into the database, thus it exists and is old.

```
    'Data Access Code to insert a new object to database
    Protected Overrides Sub DataPortal_Insert()
        'ADO.NET Queries for Inserting (Insert/Into) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'Set New flag to False since exist in database/and is Not dirty any longer
        MyBase.MarkOld()
    End Sub
```

V. **Protected Overrides DataPortal_DeleteObject()** – This is where the queries or stored procedures are listed for deleting objects from database:

- A business rule is applied that set the object as a NEW object since it was just DELETED from database, thus it DOES NOT EXIT in database thus NEW.

```
    'Data Access Code to immediatly delete an object from database.
    Protected Overrides Sub DataPortal_DeleteObject(ByVal Key As Object)
        'ADO.NET Queries for deleting (Delete/From/Where) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'Object no longer in database, therefore reset our status to be a new object
        MyBase.MarkNew()
    End Sub
```

5.5.7 Implementing Business Class Template

- Now we focus on the ***Business Class***. This is the class we will create our business objects from.
- We need to derive this class from ***BusinessBase*** which imposes *MustOverride* methods on the ***Business Class***.
- We are going to create a template of what a Business Class requires.
- NOTE, THIS IS NOT A BASE CLASS BUT A TEMPLATE TO GUIDE YOU AS TO WHAT THE CLASSES REQUIRES.

Components of Business Class

- So far we have implemented the following basic requirements for the ***Business Class***.
- First we need to inherit from **BUSINESSBASE**
- We will look at the four ***<u>MustOverride</u>*** Public Data Access Methods imposed on us by the **BusinessBase** class and to be called by the **User-Interface**:

  - **Public Overrides Sub Create()**
  - **Public Overrides Sub Load(ByVal** *Key* **As** *Object***)**
  - **Public Overrides Sub DeleteObject (ByVal** *Key* **As** *Object***)**
  - **Public Overrides Sub Save()**

- In addition we need to CREATE the ***<u>MustOverride</u>*** protected Data Access methods imposed on us by the **BusinessBase** class:

  - **Protected Overrides DataPortal_Create()**
  - **Protected Overrides DataPortal_Fetch()**
  - **Protected Overrides DataPortal_Update()**
  - **Protected Overrides DataPortal_Insert()**
  - **Protected Overrides DataPortal_DeleteObject()**

- In addition, there are .NET namespace libraries which must be included for these mechanisms to work. Therefore we will add the required libraries for the following:

  - **ADO.NET Library**
  - **I/O Library for any file access requirements**
  - **Configuration File library to use and manage configuration files storing our connection strings**
  - **Remoting Libraries**
  - **Serialization Library**

- Finally we will add to our template regions for our standard class declarations such as:

  - **Private Data**
  - **Public Event Declarations**
  - **Public Properties**
  - **Constructor Methods**
  - **Regular Business Methods**
  - **Helper Methods** – These are other methods needed by the data access or any other methods to handle some maintenance or any required process that is not business related.

5.4.8 Sample Program #2 – Creating the Business Class Template

- We now implement a template or ***Business Class*** that will server as our templates for the ***Bussiness Objects***

Example 5.2 – Creating a Business Class

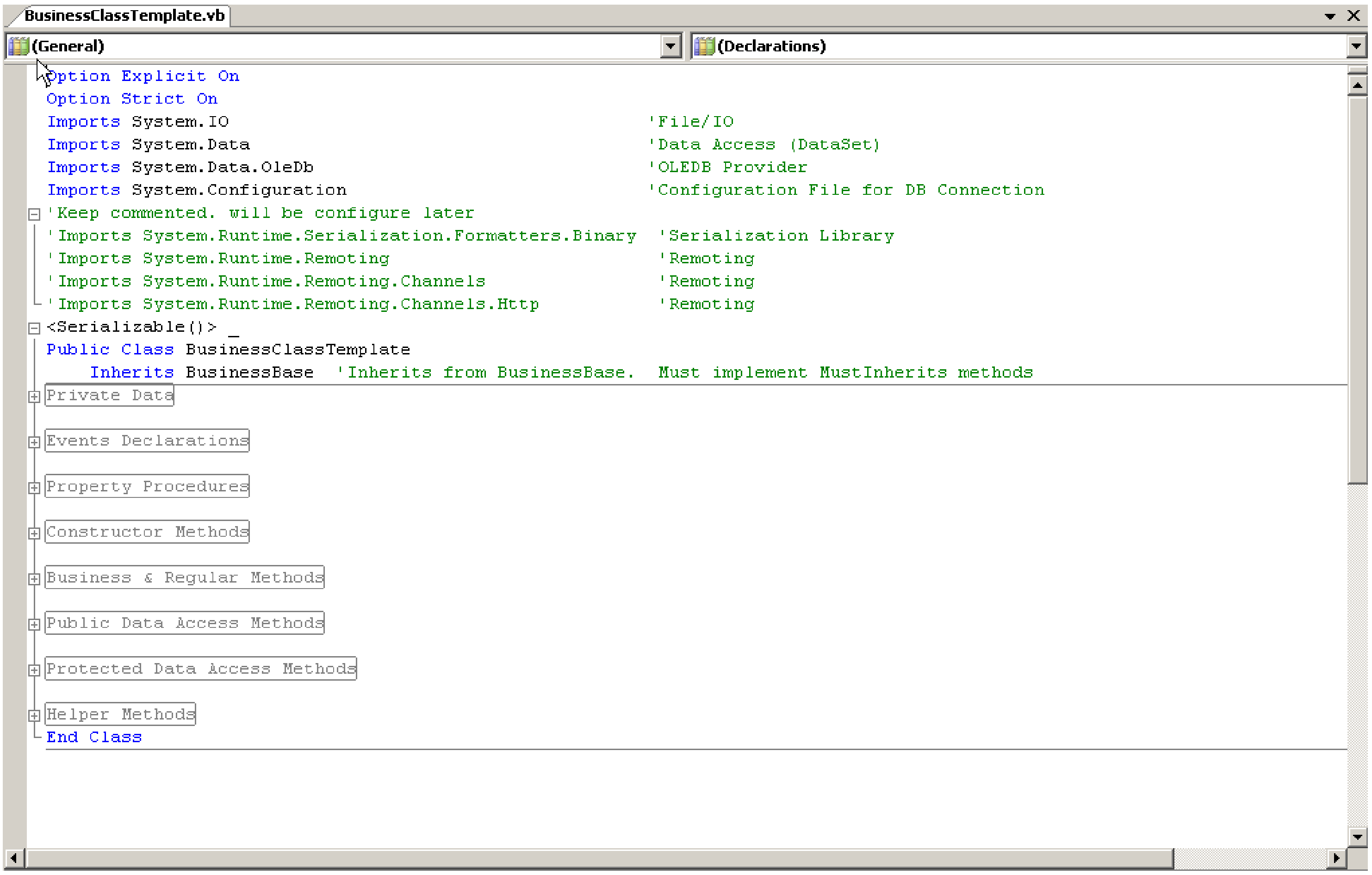
Problem statement:

- Create the ***Business Class Template*** class that we can use as a template to create our classes.

Business Object Layer – Business Class & DLL Requirements

- Add to the ***BusinessObjects*** DLL project
- The diagram below shows a Regions that make up the **Business Class Template** Format.

Region view of BusinessBase Format



BusinessClassTemplate.vb

(General) (Declarations)

```
Option Explicit On
Option Strict On
Imports System.IO                                       'File/IO
Imports System.Data                                     'Data Access (DataSet)
Imports System.Data.OleDb                               'OLEDB Provider
Imports System.Configuration                            'Configuration File for DB Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                         'Remoting
'Imports System.Runtime.Remoting.Channels                'Remoting
'Imports System.Runtime.Remoting.Channels.Http           'Remoting
<Serializable()> _
Public Class BusinessClassTemplate
    Inherits BusinessBase  'Inherits from BusinessBase.  Must implement MustInherits methods
Private Data

Events Declarations

Property Procedures

Constructor Methods

Business & Regular Methods

Public Data Access Methods

Protected Data Access Methods

Helper Methods
End Class
```

Step 0: Create an Empty Solution and do the following:

1. To the existing DLL project containing our *BusinessBase*, add a Class, name it **Business Class**

Step 1: Imports and Class declarations:

- At this point, the ***BusinessBase*** Class looks as follows. Library declarations, Unanchored Object & Class declaration:

```
Imports System.IO                                        'File/IO
Imports System.Data                                      'Data Access (DataSet)
Imports System.Data.OleDb                                'OLEDB Provider
Imports System.Configuration                             'Config File DB Connection

'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                         'Remoting
'Imports System.Runtime.Remoting.Channels                'Remoting
'Imports System.Runtime.Remoting.Channels.Http           'Remoting

<Serializable()> _
Public Class BusinessClass
    Inherits BusinessBase 'Must implement MustInherits methods
```

Step 2: Add The Common Class Components Regions:

- Add Private Data, Event declarations, Property, Constructors:

```
#Region "Events Declarations"
    '*********************************************************************
    'Event Declarations

#End Region


#Region "Property Procedures"
    '*********************************************************************
    'Class Properties declarations


#End Region


#Region "Constructor Methods"
    '*********************************************************************
    'Class Constructor declarations

#End Region


#Region "Business & Regular Methods"
    '*********************************************************************
    'Methods declarations

#End Region
```

Step 3: Add the Public Data Access Method Declarations:

- Public Shared Data Access declarations:

```
#Region "Public Shared Data Access Methods"
    '*********************************************************************
    'Public Shared Data Access Methods Declarations

    '*********************************************************************
    'Public interface to Create objects from database
    Public Overrides Sub Create()
        DataPortal_Create()
    End Sub

    Public Overrides Sub Load(ByVal Key As Object)
        'Calls Local DatPortal_Fetch(Key) To do the work
        DataPortal_Fetch(Key)

    End Sub

    Public Overrides Sub Save()
        'Only save if dirty, otherwise do nothing in this method
        If Me.IsDirty Then
            If Me.IsNew Then
                'We are new and being inserted
                'Calls Local DataPortal_Insert()
                DataPortal_Insert()
            Else
                'We are OLD so we are being updated
                'Calls Local DataPortal_Update()
                DataPortal_Update()
            End If
        End If

    End Sub

    Public Overrides Sub DeleteObject(ByVal Key As Object)
        'Calls Local DatPortal_DeleteObject() To do the work
        DataPortal_DeleteObject(Key)
    End Sub


#End Region
```

Step 4: Add the Protected Data Access Method Declarations:

- Protected Data Access Methods that contain the SQL Queries:

```
#Region "Protected Data Access Methods"
    '*********************************************************************
    'Protected Data Access Methods declarations

    'Data Access Code for Creating a New Business Object
    Protected Overrides Sub DataPortal_Create()
        'Create object and assign default values from database etc.

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'At the end, set New flag to True a new object is created
        MyBase.MarkNew()
    End Sub

    'Data Access Code to fetch an object from Database
    Protected Overrides Sub DataPortal_Fetch(ByVal Key As Object)
        'ADO.NET Queries for Fetching (Select/From/Where) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'At the end, set New flag to False.  NOT Dirty since found in database
        MyBase.MarkOld()
    End Sub

    'Data Access Code to Update an Objects data to database
    Protected Overrides Sub DataPortal_Update()
        'ADO.NET Queries for updating (Update/Set/Where) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'Set New flag to False since exist in database/and is Not dirty any longer
        MyBase.MarkOld()
    End Sub

    'Data Access Code to insert a new object to database
    Protected Overrides Sub DataPortal_Insert()
        'ADO.NET Queries for Inserting (Insert/Into) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'Set New flag to False since exist in database/and is Not dirty any longer
        MyBase.MarkOld()
    End Sub

    'Data Access Code to immediatly delete an object from database.
    Protected Overrides Sub DataPortal_DeleteObject(ByVal Key As Object)
        'ADO.NET Queries for deleting (Delete/From/Where) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'Object no longer in database, therefore reset our status to be a new object
        MyBase.MarkNew()
    End Sub

#End Region
```

Step 6: Helper Methods:

- Other non-business related methods:

```
#Region "Helper Methods"
    '*********************************************************************
    'Methods used to assist other methods or maintenance


#End Region


End Class
```

5.4.9 CONCLUSION

- WE NOW HAVE A TEMPLATE BUSINESS CLASS FROM WHICH WE CAN CREATE ALL OUR BUSINESS CLASSES!!!

5.5 BusinessCollectionBase class

5.5.1 Overview

- OK, in the previous section we created the **BusinessBase** & **Business Class** or template for our *Business Objects*.
- We also need to support for Collections of Business Objects, in other words ***Collection Classes***.
- We will now implement the **BusinessCollectionBase** Class that will serve as the base class for our Collection Classes. In addition we will create a template for our **BusinessCollection Classes** themselves.

- The ***BusinessCollectionBase*** class needs to support many of the functionality as ***BusinessBase***. They include the following only:

  - Track whether it's data has been changed or DIRTY. Note that in this case a dirty collection means that a child object or an object stored in the list has been changed.

- In addition we need to support Data Access for our Collection Classes.
- Finally we need to import the Collections Namespace into this class:

```
Imports System.Collections       'Collections Namespace
```

5.5.2 IMPORT Required Libraries

- Then First thing we need to do is IMPORT ALL THE REQUIRED LIBRARIES. These include the following:

  - ADO.NET Data Access Libraries
  - Serialization Libraries
  - Remoting Libraries
  - Other necessary libraries, for example, I will include the System.IO for any file access I may need in my projects.
  - Finally the Collection Library

```
Option Explicit On
Option Strict On

Imports System.IO                                       'File/IO
Imports System.Data                                     'Data Access (DataSet)
Imports System.Data.OleDb                               'OLEDB Provider
Imports System.Configuration                            'Configuration File Access
Imports System.Collections                              'Collection Library

'Configuration File for DB Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                         'Remoting
'Imports System.Runtime.Remoting.Channels                'Remoting
'Imports System.Runtime.Remoting.Channels.Http           'Remoting
```

5.5.3 Convert Class into Distribute Object/Unanchored Class & Inherit from DictionaryBase

- Now we convert the class into an UNANCHORED CLASS, using the following TAG, just before the class declaration:

```
<Serializable()> _
Public MustInherit Class BusinessCollectionBase
```

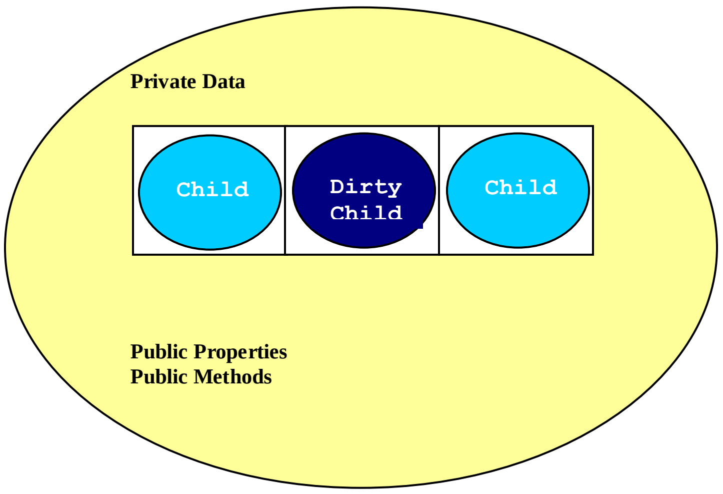
- Then we inherit this class from DICTIONARYBASE class since we are using the DICTIONARY LIBRARY.

```
<Serializable()> _
Public MustInherit Class BusinessCollectionBase
    Inherits DictionaryBase
```

5.5.4 Tracking Dirty Objects

- We need to keep track if the Collection has been modified or is DIRTY.
- A DIRTY Collection Object means that one of the CHILD objects stored in the collection has been modified, as shown in the diagram below:

Collection Business Object



- If any of the Child Object in the collection is modified, the Collection Object is DIRTY.
- So tracking simply means the following:
  1. Iterate through the Collection and ask each CHILD object if it's Dirty
  2. IF ANY OBJECT IS DIRTY, THE COLLECTION IS A DIRTY COLLECTION!

Implementing Dirty Collection Object

- To implement dirty objects, declare:

I. Iterate through the Collection and test if any of the Objects are DIRTY. As with the ***BusinessBase*** Class, the User-Interface & Business Logic needs to be able to determine if an object is DIRTY. Therefore, we will declare the READ-ONLY *Property IsDirty()*. Create the property as follows.

```
Public ReadOnly Property IsDirty() As Boolean
    Get
        'Any Dirty Object make the entire collection dirty
        Dim objDEntry As DictionaryEntry
        Dim objChild As BusinessBase
        For Each objDEntry In MyBase.Dictionary
            objChild = CType(objDEntry.Value, BusinessBase)
            If objChild.IsDirty Then Return True
        Next
        Return False
    End Get
End Property
```

5.5.5 Declared Data Access Methods

- As with *BusinessBase* & *Business Class*, the **BusinessCollectionBase** and **BusinessCollection Class** we also contain data access methods.
- As with the regular Business Objects, we will break up the data access methods into two sections, THOSE IN THAT ARE PUBLIC TO THE WORLD AND THOSE THAT ARE PROTECTED

  - ***Public*** **Data Access Methods** – These methods are Public and assessable to the User-Interface or clients.

  - ***Protected*** **Data Access methods** – These methods will actually perform the data access and contain the SQL queries or Stored Procedures. These classes are called by the ***Public Data Access Methods***.

- Again, the idea here is that there will be data access methods available to the outside world or user interface, and internal private methods that will perform the actual Data Access.
- These methods are MUSTOVERRIDE and only declared in the ***BusinessBase*** Class but implemented in the ***Business Class***.

MustOverride PUBLIC DATA ACCESS METHODS

- Again we will declare these methods **MustOverride** in our ***BusinessCollectionBase*** class, thus forcing the derived classes (***BusinessCollection Class***) to have to implement them.
- THESE METHODS ARE NOT IMPLEMENTED IN BUSINESSCOLLECTIONBASE, BUT ONLY DECLARED MUSTOVERRIDE. THEY MUST BE IMPLEMENTED IN THE DERIVED BUSINESSCOLLECTION CLASSES.
- In BusinessCollection Base, we will declare the following *MustOverride* methods:

Declaring Public & Protected Data Access Methods in BusinessBase

- To implement these methods we make the following declarations in the ***BusinessBase*** Class:

I. **Public MustOverride Create(), Load(), Save() & DeleteObject()** – These methods are *MustOverride*, therefore CANNOT be implemented in the Base Class, but the derived class will be FORCED to implement tem. Declare methods here as follows:

```
    'Public Shared Data Access Methods Declarations
    ''' Override these Public Methods in SubClass to perform Data Access
    Public MustOverride Sub Create()
    Public MustOverride Sub Load()
    Public MustOverride Sub Save()
    Public MustOverride Sub DeleteObject(ByVal Key As Object)
```

II. **Protected MustOverride Data Access Methods** – These methods are *MustOverride*, therefore CANNOT be implemented in the Base Class, but the derived class will be FORCED to implement them. Implement this method as follows:

```
    ''' Override this method in SubClass to create new Collection Object
    Protected MustOverride Sub DataPortal_Create()
    Protected MustOverride Sub DataPortal_Fetch()
    Protected MustOverride Sub DataPortal_Save()
    Protected MustOverride Sub DataPortal_DeleteObject(ByVal Key As Object)
```

5.5.6 Other Data Access Helper Methods (BusinessCollection Base)

- As with BusinessBase, we will implement in our Collection Base Class the ability to retrieve the DATABASE CONNECTION string from a configuration file:
- In the ***BusinessCollectionBase*** Class add the following code:

III. **Protected Function DBConnectionString()** – This is where the code and queries or stored procedures are listed for creating new objects and populating them with data from database:

```
    'Method will return the Database Connection string from Configuration File
    'Assumes the database name is prefixed with "DB"
    Protected Function DBConnectionString(ByVal sDatabaseName As String) As String
        Return ConfigurationManager.AppSettings("DB:" & sDatabaseName)
    End Function
```

IV. **Imported Library** – In order for this to work, we need to import the following library:

```
Imports System.Configuration      'Configuration File for DB Connection
```

5.6 BusinessCollection Base Implementation

5.6.1 Implementing BusinessCollection Base

- Now we at will put all the code together to create our ***BusinessCollectionBase*** and ***BusinessCollection Class*** Classes.
- First we focus on ***BusinessCollectionBase***
- The only required tracking mechanism is to determine if an object is dirty in the Collection:

  - Track whether it's data has been changed or DIRTY
    - Iterate through collection and ask each Business Child Object if it's dirty

- In addition we need to DECLARE ONLY the ***MustOverride*** protected Data Access methods that we are imposing upon our derived classes or children:

  - **Public MustOverride Create()**
  - **Public MustOverride Load()**
  - **Public MustOverride Save()**
  - **Public MustOverride DeleteObject(Key)**

  - **Protected MustOverride DataPorta_Create()**
  - **Protected MustOverride DataPorta_Fetch()**
  - **Protected MustOverride DataPorta_Save()**
  - **Protected MustOverride DataPorta_DeleteObject()**
  - **Protected Function DBConnectionString()**

- In addition, there are .NET namespace libraries which must be included for these mechanisms to work. Therefore we will add the required libraries for the following:

  - **ADO.NET Library**
  - **I/O Library for any file access requirements**
  - **Configuration File library to use and manage configuration files storing our connection strings**
  - **Remoting Libraries**
  - **Serialization Library**

5.6.2 Sample Program #3 – Creating the BusinessCollectionBase Class

- We now implement the ***BusinessCollectionBase*** class that will server as the basis for creating the ***Business Collection Classes***

Example 5.3 – Creating a BusinessCollectionBase Class

Problem statement:

- Create the ***BusinessCollectionBase*** class using all the rules covered in the lecture.

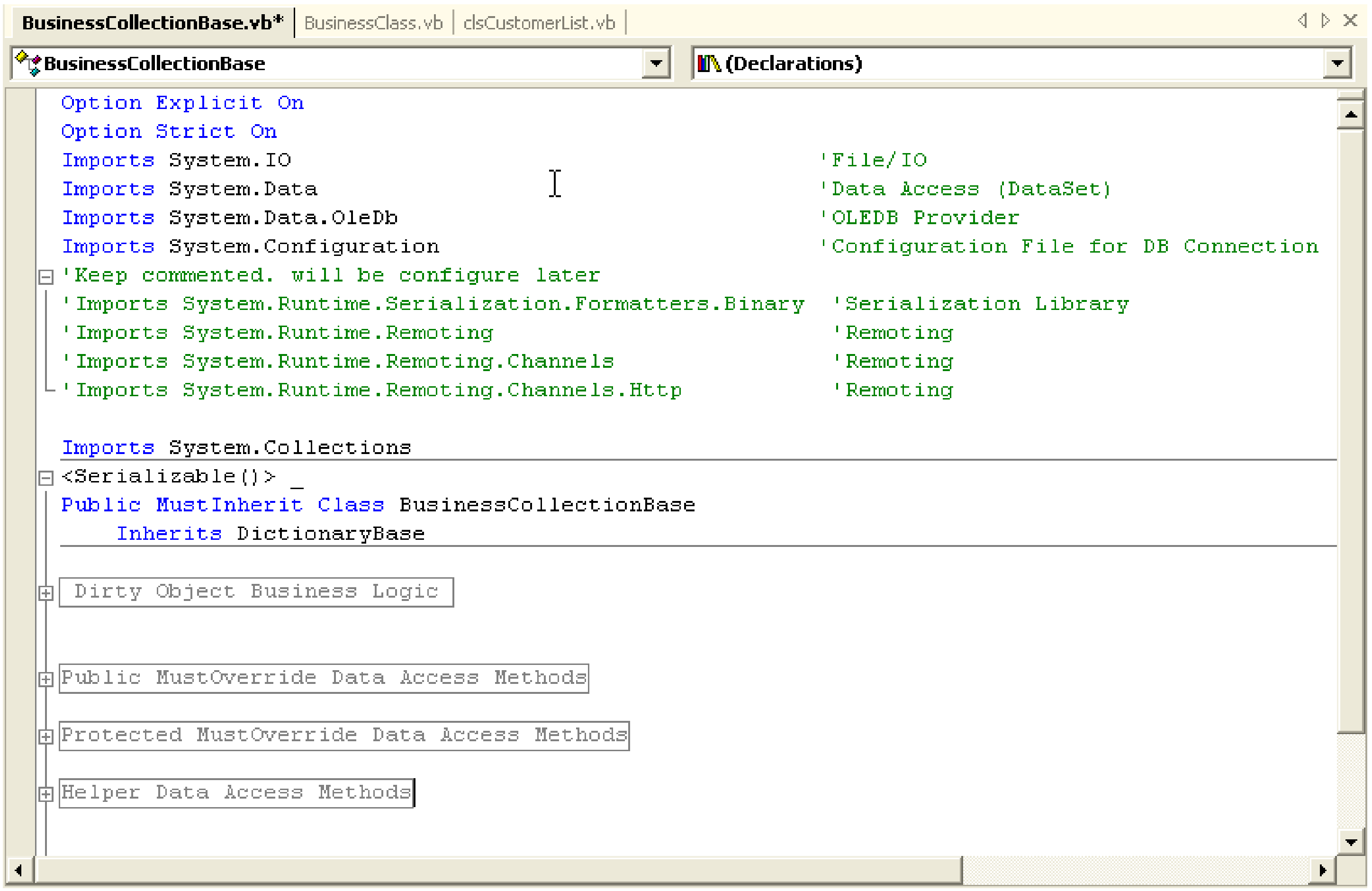
Business Object Layer – Business Class & DLL Requirements

- Implement the ***BusinessCollectionBase*** in a DLL project. Reuse the Solution/DLL project from example 5.1

Code to Implement BusinessCollectionBase Class

- Now we show all the code to implement the ***BusinessCollectionBase*** Class. Diagram below shows the view of the format for this class

Region view of BusinessCollectionBase Format



```
Option Explicit On
Option Strict On
Imports System.IO                                         'File/IO
Imports System.Data                                       'Data Access (DataSet)
Imports System.Data.OleDb                                 'OLEDB Provider
Imports System.Configuration                              'Configuration File for DB Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                          'Remoting
'Imports System.Runtime.Remoting.Channels                 'Remoting
'Imports System.Runtime.Remoting.Channels.Http            'Remoting

Imports System.Collections
<Serializable()> _
Public MustInherit Class BusinessCollectionBase
    Inherits DictionaryBase

 Dirty Object Business Logic 


Public MustOverride Data Access Methods

Protected MustOverride Data Access Methods

Helper Data Access Methods
```

Step 0: Create an Empty Solution and do the following:

1. Create a Class Library Project
2. Add a Class, name it **BusinessCollectionBase**

Step 1: Imports and Class declarations:

- At this point, the ***BusinessCollectionBase*** Class looks as follows. Library declarations, Unanchored Object & Class declaration:

```
Option Explicit On
Option Strict On

Imports System.IO                                 'File/IO
Imports System.Data                               'Data Access (DataSet)
Imports System.Data.OleDb                         'OLEDB Provider
Imports System.Configuration                      'Configuration File for DB Connection
Imports System.Collections                                'Collection Library

'Configuration File for DB Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                          'Remoting
'Imports System.Runtime.Remoting.Channels                 'Remoting
'Imports System.Runtime.Remoting.Channels.Http            'Remoting

<Serializable()> _
Public MustInherit Class BusinessCollectionBase
    Inherits DictionaryBase
```

Step 2: Add the Business Rules:

- Determine if Collection is Dirty:

```
#Region " Dirty Object Business Logic "
    'Search and Find the first Dirty Child Object
    Public ReadOnly Property IsDirty() As Boolean
        Get
            'Any Dirty Object make the entire collection dirty
            Dim objDEntry As DictionaryEntry
            Dim objChild As BusinessBase
            For Each objDEntry In MyBase.Dictionary
                objChild = CType(objDEntry.Value, BusinessBase)
                If objChild.IsDirty Then Return True
            Next
            Return False
        End Get
    End Property
#End Region
```

Step 3: Add the Public Data Access MustOverride Method Declarations:

- Public Data Access declarations:

```
#Region "Public MustOverride Data Access Methods"
    '*********************************************************************
    'Public Shared Data Access Methods Declarations
    ''' Override these Public Methods in SubClass to perform Data Access
    Public MustOverride Sub Create()
    Public MustOverride Sub Load()
    Public MustOverride Sub Save()
    Public MustOverride Sub DeleteObject(ByVal Key As Object)

#End Region
```

Step 4: Add the Protected Data Access MustOverride Method Declarations:

- Protected Data Access declarations:

```
#Region "Protected MustOverride Data Access Methods"

    ''' Override this method in SubClass to create new Collection Object
    Protected MustOverride Sub DataPortal_Create()
    Protected MustOverride Sub DataPortal_Fetch()
    Protected MustOverride Sub DataPortal_Save()
    Protected MustOverride Sub DataPortal_DeleteObject(ByVal Key As Object)

#End Region
```

Step 5: Add the Helper Data Access Method Declarations:

- Helper methods:

```
#Region "Helper Data Access Methods"

    'Method will return the Database Connection string from Configuration File
    'Assumes the database name is prefixed with "DB"
    Protected Function DBConnectionString(ByVal sDatabaseName As String) As String
        Return ConfigurationManager.AppSettings("DB:" & sDatabaseName)
    End Function


#End Region

End Class
```

5.6.3 CONCLUSION

- WE NOW HAVE A BASE CLASS TO ENFORCE BUSINESS RULES ON OUR COLLECTIONS CLASSES!!!

5.7 BusinessCollection Class Details

5.7.1 Overview

- We will now implement the **BusinessCollectionClass** Class that will serve as a template or model for us to create our Collection Classes.
- These are the actual collection classes that will do the work, such as CustomerList, EmployeeList etc. and will be derived from the base class ***BusinessCollectionBase***
- Our ***Collection Classes*** are imposed the MustOverride Data Access methods by the ***BusinessCollectionBase*** Class..

5.7.2 Business Class Requirements

- REMEMBER THIS IS THE CLASS IN WHICH WE WILL BUILD OUR COLLECTION CLASSES FROM (clsCustomerList, clsEmployeeList etc.)
- DO NOT CONFUSE THE BUSINESS COLLECTION CLASS WITH BUSINESS COLLECTION OBJECTS, BUSINESS COLLECTION OBJECTS ARE INSTANCE OF A BUSINESS CLASS!
- OBJECT-ORIENTED RULES REVISED:

  I. CREATE BUSINESS COLLECTION CLASS
  II. CREATE BUSINESS COLLECTION OBJECT
  III. USE BUSINESS COLLECTION OBJECT

- We will now CREATE A BUSINESS COLLECTION CLASS TEMPLATE, that we can use to guide us in creating or modifying our COLLECTION CLASSES.
- This class will be inherited from ***BusinessCollectionBase*** thus FORCING the business rules and data access methods

5.7.3 IMPORT Required Libraries

- Then First thing we need to do is IMPORT ALL THE REQUIRED LIBRARIES. These include the following:

  - ADO.NET Data Access Libraries
  - Serialization Libraries
  - Remoting Libraries
  - Other necessary libraries, for example, I will include the System.IO for any file access I may need in my projects.

```
Option Explicit On
Option Strict On

Imports System.IO                                'File/IO
Imports System.Data                              'Data Access (DataSet)
Imports System.Data.OleDb                        'OLEDB Provider
Imports System.Configuration                     'Configuration File for DB Connection

'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary   'Serialization Library
'Imports System.Runtime.Remoting                          'Remoting
'Imports System.Runtime.Remoting.Channels                 'Remoting
'Imports System.Runtime.Remoting.Channels.Http            'Remoting
```

5.7.4 Convert Class into Distribute Object/Unanchored Class & Inherit from BusinessCollectionBase Class

- Now we convert the class into an UNANCHORED CLASS, using the following TAG, just before the class declaration:

```
<Serializable()> _
Public Class BusinessCollectionClass
    Inherits BusinessCollectionBase
```

5.7.5 Implementing Data, Properties, Methods and Events

- Nothing new here. These are the components every COLLECTION CLASS HAS, such as public properties (Count, Item), public Wrapper methods (Add, Remove, Clear()), Regular methods (Edit, Print, PrintAll, Authenticate etc.)

5.7.6 Public Data Access Methods Forced upon us by BusinessCollectionBase

- We will look at the four Public Data Access Methods created in the ***BusinessCollection Classes*** to be called by the **User-Interface**
- These are similar to the ones used in the Business Classes, except that we are now using a Collection.

Public Data Access Methods Implementation Details

- In our Business Classes, we will use **ADO.NET** to implement our data access. In order to user **ADO.NET** we need to add the namespace libraries as follows:

```
Imports System.Data
Imports System.Data.OleDb  'Data Access Library OLEDB Provider
```

Implementing Create(), Load(key), DeleteObject(Key) and Save()

- NOTE THA THE IMPLEMENTATIONS SHOWN HERE ARE EXAMPLES ONLY. THERE ARE MANY WAYS TO DO THIS. YOU ARE WELCOME TO READ OTHER MATERIAL AND LEARN HOW IS DONE BY OTHER DEVELOPERS AND AUTHORS. To implement these methods we make the following declarations in the ***BusinessCollection Class***:

I. **Public Overrides Sub Create()** – CREATES A NEW COLLECTION OBJECT. Implement this method as follows:

```
    Public Overrides Sub Create()
        'Calls Local DatPortal_Create() To do the work
        DataPortal_Create()

    End Sub
```

II. **Public Overrides Sub Load()** – LOADS COLLECTION WITH OBJECTS. Implement this method as follows:

```
    Public Overrides Sub Load()
        'Calls Local DatPortal_Fetch() To do the work
        DataPortal_Fetch()

    End Sub
```

III. **Public Overrides Sub DeleteObject()** – DELETES A CHILD OBJECT. Implement this method as follows:

```
    Public Overrides Sub DeleteObject(ByVal Key As Object)
        'Calls Local DatPortal_DeleteObject() To do the work
        DataPortal_DeleteObject(Key)
    End Sub
```

IV. **Public Overrides Sub Save()** – SAVES COLLECTION TO DATABASE. Implement this method as follows:

- Note that the decision to perform and update or insert is based on the status of the DIRTY flags. No need to iterate through the collection and save every object if none of the objects are DIRTY!

```
Public Overrides Sub Save()
    'Verify there are dirty objects in Collection
    'Only modify if dirty, otherwise do nothing in this method
    If IsDirty Then
        'Dirty Collection, go ahead and update
        DataPortal_Save()
    End If

End Sub
```

5.7.7 BusinessCollection Class – Protected Data Access Methods

- We now focus on the Protected Data Access Methods imposed on us (MustOverride) by the ***BusinessCollectionBase*** Classe.
- These methods can only be called from within the ***BusinessCollection Class*** and it's children
- Since we declared these methods in the ***BusinessCollectionBase*** Class as ***MustOverride*** methods, we are forced to implement them here otherwise we cannot compile our class.
- To implement these methods we make the following declarations in the ***BusinessCollection Class***:

I. **Protected Overrides DataPortal_Create()** – This is where the code and queries or stored procedures are listed for creating new objects and populating them with data from database:

```
'Data Access or other Code for Creating a New Business COLLECTION Object
Protected Overrides Sub DataPortal_Create()
    'Create object and assign default values from database etc.
End Sub
```

II. **Protected Overrides DataPortal_Fetch()** – This method iterates through the collection and add the populated objects to collection. ADO.NET code and query or stored procedure will be required:

```
Protected Overrides Sub DataPortal_Fetch()
    'Iterates through Collection, Calling Each CHILD object.Load() method
    'CHILD Objects load themselves.  ADO.NET Queries may be required
    'for obtaininig key of every object for every object to load themselves

    'THIS CODE WILL BE IMPLEMENTED WHEN DURING THE ADO.NET LECTURES
End Sub
```

III. Protected Overrides DataPortal_Save() – Save is done by iterating through Collection and asking every object to save themselves:

```
'Data Access Code to Update an Objects data to database
'Simply iterate through collection and call each object's save method
Protected Overrides Sub DataPortal_Save()
    'Iterates through Collection, Calling Each CHILD object.Save() method
    'CHILD Objects save themselves
    'Step A- Begin Error trapping
    Try
        'Step 1-Step 1-Create Temporary Person and Dictionary object POINTERS
        Dim objDictionaryEntry As DictionaryEntry
        Dim objChild As BusinessClass

        'Step 2-Use For..Each loop to iterate through Collection
        For Each objDictionaryEntry In MyBase.Dictionary
            'Step 3-Convert DictionaryEntry pointer returned to Type Person
            objChild = CType(objDictionaryEntry.Value, BusinessClass)

            'Step 4-Call Child to Save itself
            objChild.Save()

        Next
        'Step B-Traps for general exceptions.
    Catch objE As Exception
        'Step C-Throw an general exceptions
        Throw New System.Exception("Save Error! " & objE.Message)
    End Try
End Sub
```

- NOTE THAT IN YOUR PROJECT, YOU NEED TO REPLACE THE NAME BusinessClass WITH THE CLASS TYPE OF THE CHILD OBJECTS YOU ARE STORING AND SAVING IN THE COLLECTION, For example, ***clsEmployee***, ***clsCustomer***, etc.

IV. **Protected Overrides DataPortal_DeleteObject()** – Iterates through collection, finds target object and tells object to delete itself. Optional, you can also delete the object from the collection or leave it to the UI programmer to do so.

```
'Data Access Code to immediatly delete an object from database.
Protected Overrides Sub DataPortal_DeleteObject(ByVal Key As Object)
    'Iterates through Collection, Calling Each CHILD object.Delete() method
    'CHILD Objects Delete themselves

    'Step A- Begin Error trapping
    Try
        'Step 1-Step 1-Create Temporary Person and Dictionary object POINTERS
        Dim objDictionaryEntry As DictionaryEntry
        Dim objChild As BusinessClass

        'Step 2-Use For..Each loop to iterate through Collection
        For Each objDictionaryEntry In MyBase.Dictionary
            'Step 3-Convert DictionaryEntry pointer returned to Type Person
            objChild = CType(objDictionaryEntry.Value, BusinessClass)

            'Step 4-Find target object based on key
            'YOU WILL NEED TO SELECT THE CORRECT PROPERTY
            'FOR objItem.Property, ALSO YOU NEED TO CONVERT THE
            'KEY PARAMETER USING CSTR OR CINT ETC. DEPENDING
            'ON THE DATATYPE OF THE ob
           If objItem.Property = CStr(Key) Then

                'Step 5-Object deletes itself
                objChild.DeleteObject(Key)

                ''Step 6-[OPTIONAL] Remove Object From Collection
                ''since no longer in DB
                'MyBase.Dictionary.Remove(Key)
            End If

        Next
        'Step B-Traps for general exceptions.
    Catch objE As Exception
        'Step C-Throw an general exceptions
        Throw New System.Exception("Save Error! " & objE.Message)
    End Try

End Sub
```

- AGAIN, HERE YOU NEED TO REPLACE THE NAME BusinessClass WITH THE CLASS TYPE OF THE CHILD OBJECTS YOU ARE STORING AND SAVING IN THE COLLECTION, For example, ***clsEmployee***, ***clsCustomer***, etc.

5.8 Creating the BusinessCollectionClass Template

5.8.1 Implementing BusinessCollection Class Template

- Now we focus on the ***BusinessCollection Class***. This is the class we will create our business COLLECTION objects from.
- We need to derive this class from ***BusinessCollectionBase*** which imposes *MustOverride* methods.
- We are going to create a template of what a Business Class requires.
- NOTE, THIS IS NOT A BASE CLASS BUT A TEMPLATE TO GUIDE YOU AS TO WHAT THE CLASSES REQUIRES.

Components of BusinessCollection Class

- First we need to inherit from **BUSINESSCOLLECTIONBASE**
- We will implement the four Public Data Access Methods created to be called by the **User-Interface**:

  - **Public Overrides Sub Create()** – Creates objects with default values from DB. Class the ***Protected DataPortal_Create()*** method to do the work, the queries etc.

  - **Public Overrides Sub Load()** – Fetches data from database all objects and populates COLLECTION. Calls the ***Protected DataPortal_Fetch()*** method to do the work.

  - **Public Overrides Sub DeleteObject (ByVal** *Key* **As** *Object***)** – Iterates through COLLECTION and Deletes object from database. Calls ***Protected DataPortal_DeleteObject(*****ByVal** *Key* **As** *Object***)** methods to do the work.

  - **Public Overrides Sub Save()** – Iterates through collection and saves each object. Calls ***Protected DataPortal_Save()*** to do the work.

- In addition we need to CREATE the ***MustOverride*** protected Data Access methods imposed on us by the **BusinessBase** class:

  - **Protected Overrides DataPorta_Create()**
  - **Protected Overrides DataPorta_Fetch()**
  - **Protected Overrides DataPorta_Save()**
  - **Protected Overrides DataPorta_DeleteObject()**

- In addition, there are .NET namespace libraries which must be included for these mechanisms to work. Therefore we will add the required libraries for the following:

  - **ADO.NET Library**
  - **I/O Library for any file access requirements**
  - **Configuration File library to use and manage configuration files storing our connection strings**
  - **Remoting Libraries**
  - **Serialization Library**

- Finally we will add to our template regions for our standard COLLECTION CLASS declarations such as:

  - **Public Properties** (Count, Item, etc.)
  - **Wrapper Methods**
  - **Regular Methods**
  - **Helper Methods** – These are other methods needed by the data access or any other methods to handle some maintenance or any required process that is not business related.

5.8.2 Sample Program #4 – Creating the BusinessCollection Class Template

- We now implement a template or ***BusinessCollection Class*** template that will server as our templates for the ***COLLECTION Objects***

Example 5.4 – Creating a BusinessCollection Class Template

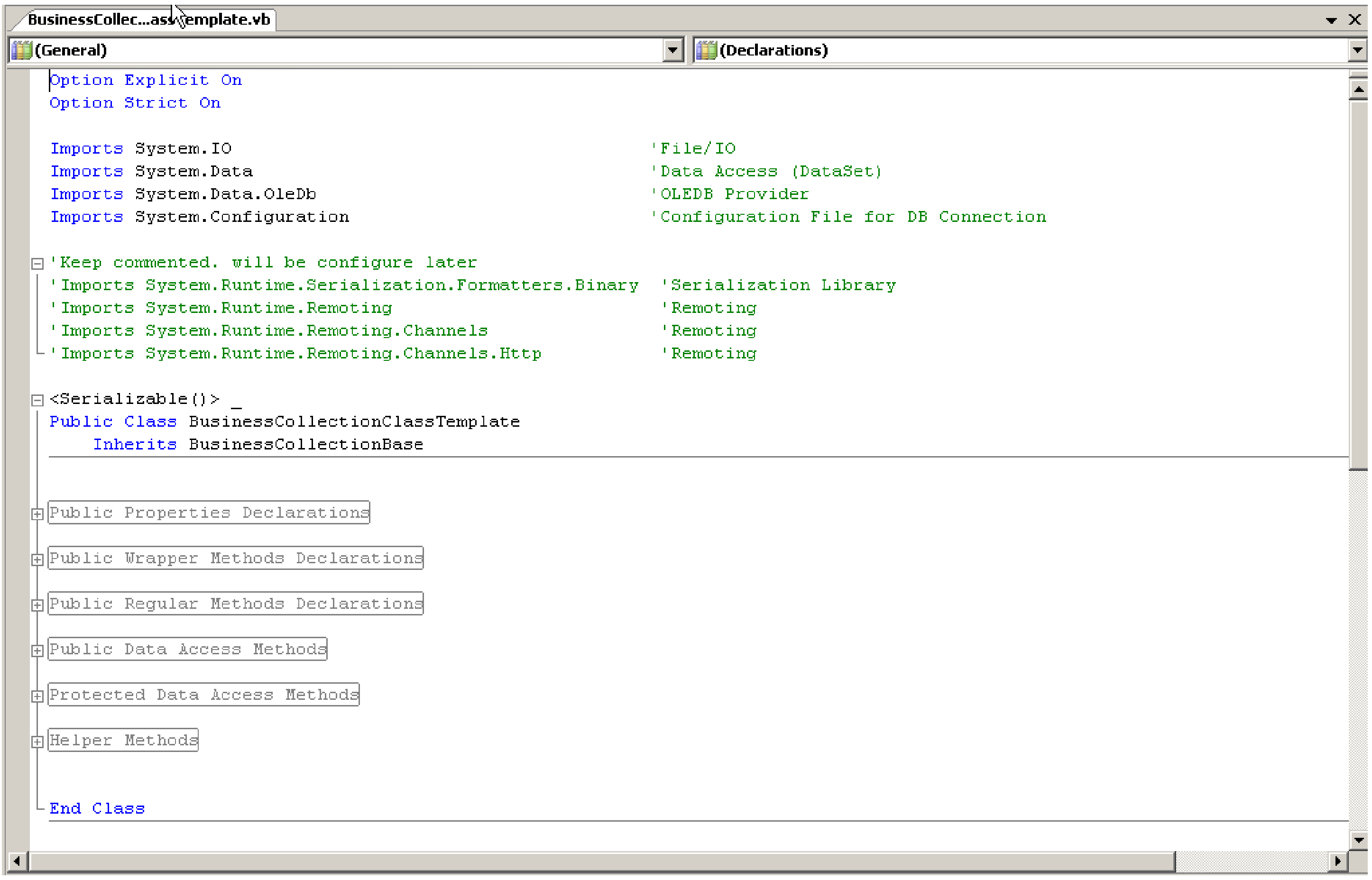
Problem statement:

- Create the ***BusinessCollection Class Template*** class that we can use as a template to create our classes.

Business Object Layer – Business Class & DLL Requirements

- Implement the ***BusinessCollection Class*** in a DLL project
- The diagram below shows the **Regions** that make up the **Business Class Template** Format.

Region view of BusinessCollection Class Format



```
Option Explicit On
Option Strict On

Imports System.IO                                          'File/IO
Imports System.Data                                        'Data Access (DataSet)
Imports System.Data.OleDb                                  'OLEDB Provider
Imports System.Configuration                               'Configuration File for DB Connection

'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary    'Serialization Library
'Imports System.Runtime.Remoting                           'Remoting
'Imports System.Runtime.Remoting.Channels                  'Remoting
'Imports System.Runtime.Remoting.Channels.Http             'Remoting

<Serializable()> _
Public Class BusinessCollectionClassTemplate
    Inherits BusinessCollectionBase


Public Properties Declarations

Public Wrapper Methods Declarations

Public Regular Methods Declarations

Public Data Access Methods

Protected Data Access Methods

Helper Methods


End Class
```

Step 0: Create an Empty Solution and do the following:

1. To the existing DLL project containing our *BusinessBase*, *BusinessClass*, & *BusinessCollectionBase* add a Class
2. Name the class **BusinessCollectionTemplate Class**

Step 1: Imports and Class declarations:

- At this point, the ***BusinessBase*** Class looks as follows. Library declarations, Unanchored Object & Class declaration:

```
Option Explicit On
Option Strict On

Imports System.IO                                'File/IO
Imports System.Data                              'Data Access (DataSet)
Imports System.Data.OleDb                        'OLEDB Provider
Imports System.Configuration                     'Configuration File for DB Connection

'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                          'Remoting
'Imports System.Runtime.Remoting.Channels                 'Remoting
'Imports System.Runtime.Remoting.Channels.Http            'Remoting

<Serializable()> _
Public Class BusinessCollectionClassTemplate
    Inherits BusinessCollectionBase
```

Step 2: Add The Common COLLECTION Class Properties Region:

- Add Properties, Wrapper Methods etc:

```
#Region "Public Properties Declarations"
    '*********************************************************************
    'Class Properties declarations, Example Count, Item etc.
    '*************************************************************************
    'Name:          Count Property                                          *
    'Purpose:       Returns the number of item in collection                *
    '***************************e*********************************************
    Public Shadows ReadOnly Property Count() As Integer
        Get
            Return MyBase.Dictionary.Count
        End Get
    End Property

    '*************************************************************************
    'Name:          Item(Key) Property                                      *
    'Purpose:       Sets or get the object specified by key                 *
    '***************************e*********************************************
    Public Property Item(ByVal key As Object) As BusinessClassTemplate
        Get
            'Step 1- Return POINTER of Object of associated key
            'Convert returned POINTER
            Return CType(MyBase.Dictionary.Item(key), BusinessClassTemplate)
        End Get
        Set(ByVal value As BusinessClassTemplate)

            'Step 1-Verify if key exists
            If MyBase.Dictionary.Contains(key) Then
                'Step 2-Set or overwrite object in collection
                MyBase.Dictionary.Item(key) = value
            Else
                'Step 3-Else throws an Argument Exeption to indicate not found.
                Throw New System.ArgumentException("Key Not found")
            End If
        End Set
    End Property
```

Step 3: Add The Common COLLECTION Class Wrapper Region:

- Add Wrapper Methods etc:

```
#Region "Public Wrapper Methods Declarations"

'**********************************************************************************
    ''' <summary>
    ''' Name: Add(Key, Object)Method
    ''' Purpose: Adds new object to the Collection.
    ''' Includes support for duplicate key
    ''' </summary>
    ''' <param name="key"></param>
    ''' <param name="objItem"></param>
    ''' <remarks></remarks>
    Public Sub Add(ByVal key As Object, ByVal objItem As BusinessClassTemplate)
        'Step A- Begin Error trapping
        Try
            'Step 1-Calls Collection.Add(Key,Object) Method to Add object
            MyBase.Dictionary.Add(key, objItem)

            'Step B-Traps argumentNullException when key is Nothing or null
        Catch objX As ArgumentNullException
            'Step C-ReThrow ArgumentNullException
            Throw New System.ArgumentNullException("Invalid Key: " & objX.Message)
            'Step D-Traps for ArgumentExecption when KEY is duplicate.
        Catch objY As ArgumentException
            'Step E-ReThrow an ArgumentExecption to calling programs
            Throw New System.ArgumentException("Duplicate Key: " & objY.Message)
            'Step F-Traps for general exceptions.
        Catch objE As Exception
            'Step G-ReThrow an general exceptions
            Throw New System.Exception("Add Method Error: " & objE.Message)
        End Try
    End Sub
```

- Continue Wrapper Methods etc:

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Function Remove(Key)Sub Method
    ''' Purpose: Remove object from collection based on key.
    ''' </summary>
    ''' <param name="key"></param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Function Remove(ByVal key As Object) As Boolean
        'Step A- Begin Error trapping
        Try

            'Step 1-Verify object exists
            If MyBase.Dictionary.Contains(key) Then
                'Step 2-Calls CollectionObject.Remove(Key) Method
                MyBase.Dictionary.Remove(key)
                'Step 3-Return True since found and removed
                Return True
            Else
                'Step 4-Return False since not found
                Return False
            End If

            'Step B-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step C-Throw Collection ArgumentNullException
            Throw New System.ArgumentNullException("Invalid Key: " & objX.Message)
            'Step D-Traps for general exceptions.
        Catch objE As Exception
            'Step E-Throw an general exceptions
            Throw New System.Exception("Remove Error: " & objE.Message)
        End Try
    End Function
```

❑ Continue Wrapper Methods etc:

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Clear()Method
    ''' Purpose: Remove all objects from collection
    ''' </summary>
    ''' <remarks></remarks>
    Public Shadows Sub Clear()
        'Step A- Begin Error trapping
        Try
            'Step 1-Calls Collection.Clear() Method
            MyBase.Dictionary.Clear()

            'Step B-Traps for General exceptions
        Catch objex As Exception
            'Step C-Throw an General Execption to calling programs.
            Throw New System.Exception("Unexpected error clear(). " & objex.Message)
        End Try
    End Sub

    '*********************************************************************
    'Name:          Contains()Method                                     *
    'Purpose:       Verify if object is in Collection                    *
    '*********************************************************************
    Public Shadows Function Contains(ByVal Key As Object) As Boolean
        'Step A- Begin Error trapping
        Try
            If MyBase.Dictionary.Contains(Key) Then
                Return True
            Else
                Return False
            End If

            'Step B-Traps for General exceptions
        Catch objex As Exception
            'Step C-Throw an General Execption
            Throw New System.Exception(objex.Message)
        End Try
    End Function

    '********************************************************************
    'Add other Overloaded Wrappers here as well, such as Add(x,y,z..)

#End Region
```

Step 5: Add Regular Method Declarations:

❑ Public Regular Methods or non Wrapper methods, such as Edit, Print, etc.:

```
#Region "Public Regular Methods Declarations"
    '********************************************************************
    'Class Regular Methods.  Ex:  EditItem(k,O), EditItem(x,y,z..), Print(X), etc.


#End Region
```

Step 6: Add the Public Shared Data Access Method Declarations:

- Public Shared Data Access declarations:

```
#Region "Public Data Access Methods"
    '*********************************************************************
    ''' <summary>
    ''' [Optional] Calls Data Portal_Create to create a Collection Object. This
    ''' Method is not used in this class, but can be used in the
    ''' future to create objects that need data from database upon Creation
    ''' </summary>
    ''' <remarks></remarks>
    Public Overrides Sub Create()
        'Calls Local DatPortal_Create() To do the work
        DataPortal_Create()

    End Sub

    '***************************************************************************
    ''' <summary>
    ''' Calls Data_Portal_Fetch to load all objects from database
    ''' </summary>
    ''' <remarks></remarks>
    Public Overrides Sub Load()
        'Calls Local DatPortal_Fetch() To do the work
        DataPortal_Fetch()

    End Sub

    '***************************************************************************
    ''' <summary>
    ''' Calls DataPortal_Save() to save all objects in collection to Database
    ''' </summary>
    ''' <remarks></remarks>
    Public Overrides Sub Save()
        'Verify there are dirty objects in Collection
        'Only modify if dirty, otherwise do nothing in this method
        If IsDirty Then
            'Dirty Collection, go ahead and update
            DataPortal_Save()
        End If

    End Sub

    '***************************************************************************
    ''' <summary>
    ''' Calls DataPortal_DeleteObject to delete an object residing
    '''  In the collection from the database
    ''' </summary>
    ''' <param name="Key"></param>
    ''' <remarks></remarks>
    Public Overrides Sub DeleteObject(ByVal Key As Object)
        'Calls Local DatPortal_DeleteObject() To do the work
        DataPortal_DeleteObject(Key)
    End Sub

#End Region
```

Step 7: Add the Protected Data Access Method Declarations:

- Protected Data Access Methods that contain the SQL Queries etc.:

```
#Region "Protected Data Access Methods"
    '*********************************************************************
    'Protected Data Access Methods declarations


    '***************************************************************************
    ''' <summary>
    ''' Data Access or other Code for Creating a New Business COLLECTION Object
    ''' Used when object requires data from db upon creation
    ''' </summary>
    ''' <remarks></remarks>
    Protected Overrides Sub DataPortal_Create()
        'Create object and assign default values from database etc.

    End Sub

    '***************************************************************************
    ''' <summary>
    ''' Loads all objects from database by Iterating through Collection, and
    ''' calling Each ITEM object LOAD() method so each Item loads itself
    ''' </summary>
    ''' <remarks></remarks>
    Protected Overrides Sub DataPortal_Fetch()
        'Iterates through Collection, Calling Each CHILD object.Load() method
        'CHILD Objects load themselves.  ADO.NET Queries may be required
        'for obtaininig key of every object for every object to load themselves

        'THIS CODE WILL BE IMPLEMENTED WHEN DURING THE ADO.NET LECTURES

    End Sub
```

- Continue Protected Data Access Methods:

```
    '************************************************************************
    ''' <summary>
    ''' SAVES all objects from database by Iterating through Collection, and
    ''' calling Each ITEM object SAVE() method so each Item saves itself
    ''' </summary>
    ''' <remarks></remarks>
    Protected Overrides Sub DataPortal_Save()
        'Iterates through Collection, Calling Each CHILD object.Save() method
        'CHILD Objects save themselves
        'Step A- Begin Error trapping
        Try
            'Step 1-Step 1-Create Temporary Person and Dictionary object POINTERS
            Dim objDictionaryEntry As DictionaryEntry
            Dim objChild As BusinessClassTemplate

            'Step 2-Use For..Each loop to iterate through Collection
            For Each objDictionaryEntry In MyBase.Dictionary
                'Step 3-Convert DictionaryEntry pointer returned to Type Person
                objChild = CType(objDictionaryEntry.Value, BusinessClassTemplate)

                'Step 4-Call Child to Save itself
                objChild.Save()

            Next
            'Step B-Traps for general exceptions.
        Catch objE As Exception
            'Step C-Throw an general exceptions
            Throw New System.Exception("Save Error! " & objE.Message)
        End Try
    End Sub
```

- IMPORTANT! YOU NEED TO REPLACE **BusinessClassTemplate** STATEMENT BY THE CLASS OF THE ITEMS BEING STORED IN THE COLLECTION, FOR EXAMPLE, clsCustomer, clsEmployee etc.

- Continue Protected Data Access Methods:

```
    '*************************************************************************
    ''' <summary>
    ''' DELETES AN OBJECT BY ID from database by Iterating through Collection
    ''' and calling Each ITEM object DELETE(ID) method so each Item delete itself
    ''' </summary>
    ''' <param name="Key"></param>
    ''' <remarks></remarks>
    Protected Overrides Sub DataPortal_DeleteObject(ByVal Key As Object)
        'Iterates through Collection, Calling Each CHILD object.Delete() method
        'CHILD Objects Delete themselves

        'Step A- Begin Error trapping
        Try
            'Step 1-Step 1-Create Temporary Person and Dictionary object POINTERS
            Dim objDictionaryEntry As DictionaryEntry
            Dim objItem As BusinessClassTemplate

            'Step 2-Use For..Each loop to iterate through Collection
            For Each objDictionaryEntry In MyBase.Dictionary
                'Step 3-Convert DictionaryEntry pointer returned to Type Person
                objItem = CType(objDictionaryEntry.Value, BusinessClassTemplate)

                'Step 4-Find target object based on key
                'YOU WILL NEED TO SELECT THE CORRECT PROPERTY
                'FOR objItem.Property, ALSO YOU NEED TO CONVERT THE
                'KEY PARAMETER USING CSTR OR CINT ETC. DEPENDING
                'ON THE DATATYPE OF THE objItem.Property
                If objItem.Property = CStr(Key) Then
                    'Step 5-Object deletes itself
                    objItem.DeleteObject(Key)

                    ''Step 6-[OPTIONAL] Remove Object From Collection
                    ''since no longer in DB
                    'MyBase.Dictionary.Remove(Key)
                End If
            Next
            'Step B-Traps for general exceptions.
        Catch objE As Exception
            'Step C-Throw an general exceptions
            Throw New System.Exception("Save Error! " & objE.Message)
        End Try

    End Sub
```

- IMPORTANT! Note that in the following code, **Property** represents the ID number, SS number or whatever is the ID property of the Item Object. YOU NEED TO MODIFY THIS CODE, REPLACE THE PROPERTY BY THE CORRECT PROPERTY OF THE OBJECT. ALSO YOU NEED TO USE THE CORRECT DATA TYPE CONVERSION FUNCTION INSTEAD OF **CStr()**

```
                If objItem.Property = CStr(Key) Then
                    'Step 5-Object deletes itself
                    objItem.DeleteObject(Key)
```

- ALSO, YOU NEED TO REPLACE TO REPLACE `BusinessClassTemplate` STATEMENT BY THE CLASS OF THE ITEMS BEING STORED IN THE COLLECTION, FOR EXAMPLE, clsCustomer, clsEmployee etc

Step 7: Helper Methods:

- ❑ Other non-business related methods:

```
#Region "Helper Methods"
    '*******************************************************************
    'Methods used to assist other methods or maintenance


#End Region


End Class
```

5.8.3 CONCLUSION

- ❑ WE NOW HAVE A TEMPLATE BUSINESS COLLECTION CLASS FROM WHICH WE CAN CREATE ALL OUR COLLECTION CLASSES!!!

5.9 Business Rules and Validation (Business Object Requirements)

- Now we address how to use some of the business rules we've implemented so far.
- In addition we implement another requirement for our Business Objects, and that is that they must validate themselves.

5.9.1 Implementing Dirty & NEW Business Rule In Properties & Methods

- We implemented several Business Rules and logic into our templates, such as NEW & DIRTY Objects.
- We now look at how to implement these rules.

Implementing Dirty Objects in Property Methods

- Every time an object is SET with data via properties, the object is DIRTY!.
- Therefore we need to MARK EVERY SET portion of a property by calling the Business Rule **MARKDIRTY()** method
- For example, lets look at the following Name Property:

```
    Public Property Name() As String
        Get
            Return m_Name
        End Get
        Set(ByVal value As String)

            m_Name = value

            'Mark Ojbect as dirty it has been modified
            MyBase.MarkDirty()

       End Set
    End Property
```

- Another example:

```
    Public Property IDNumber() As Integer
        Get
            Return m_IDNumber
        End Get
        Set(ByVal value As Integer)

                m_IDNumber = value

         MyBase.MarkDirty()   'Now DIRTY! Must be in Every Set Property

        End Set
    End Property
```

- **IMPORTANT! EVERY PROPERTY SET MUST HAVE THE CALL TO MARKDIRTY()**

Implementing Dirty Objects in Regular Methods

- ❑ As usual, you need to add you're the regular methods that make the object behave like its real world counterpart.
- ❑ Nevertheless, if a Method makes any modification to the data, then we need to mark the object as **DIRTY** once the method executes.
- ❑ For example, in the following Shop() method, modifies the private data Therefore it must be marked DIRTY

```
    'Methods modifies data, object must be marked as Dirty
    Public Sub Shop(ByVal intItems As Integer)
        'Data is modified
        intTotalItemsPurchased = intTotalItemsPurchased + intItems

        MyBase.MarkDirty() 'Must Mark Dirty since private data is being modified

        'Raise or trigger event & send information with the event
        RaiseEvent OnShopping(intTotalItemsPurchased)

    End Sub
```

- ❑ Note that if a method makes no kind of modification to the data, then we DO NOT need to mark it as dirty
- ❑ **ONLY METHODS THAT MODIFY DATA MUST CALL THE MARKDIRTY() METHOD!**

Dirty Objects & Public Data Access Methods

- ❑ Our ***BusinessClasses*** & ***BusinessCollectionClasses*** contain Public Data Access Methods.
- ❑ These include:

  - **Public** Create()
  - **Public** Load()
  - **Public** Save()
  - **Public** DeleteObject (Key)

- ❑ These Public methods don't require that we mark them DIRTY since these methods simply call the Protected DataPortal_XXX classes to do the work. It is inside the Protected Classes were changes are made and we need to apply these rules

Implementing Dirty Objects in Protected Data Access Methods

- Because these are the classes that actually perform the Data Access and modify the object, we need to implement our DIRTY AND NEW LOGIC.
- This applies only to the ***BusinessClass*** and NOT the COLLECTION ***BusinessCollectionClass***.
- The COLLECTION CLASSES, don't really modify the CHILD Business Objects, they rely on these object to do their own DIRTY WORK, therefore collection classes don't require that we add DIRTY or NEW logic to the Data Access Methods.
- With that said lets focus on the ***BusinessClass*** Protected Data Access Methods
- The protected methods include:

  - **Protected Overrides DataPorta_Create()**
  - **Protected Overrides DataPorta_Fetch()**
  - **Protected Overrides DataPorta_Update()**
  - **Protected Overrides DataPorta_Insert()**
  - **Protected Overrides DataPorta_DeleteObject()**

Business Rules & DataPortal_Create() method

- This method loads creates new object and populates them with default values from database etc.
- **IMPORTANT!** – Business Rules dictate that newly create objects are NEW. With this in mind, we need to call the ***MarkNew()*** method at the end of the method as follows:

```
    'Data Access Code for Creating a New Business Object
    Protected Overrides Sub DataPortal_Create()
        'Create object and assign default values from database etc.

        'At the end, set New flag to True a new object is created
        MarkNew()
    End Sub
```

Business Rules & DataPortal_Fetch(Key) method

- This method loads the object with data from the database based on the key. Using **ADO.NET**.
- **IMPORTANT!** – Business Rules dictate that an object loaded from database is marked **OLD** since it does exist in the database. With this in mind, we need to call the ***MarkOld()*** method at the end of the method as follows:

```
    'Data Access Code to fetch an object from Database
    Protected Overrides Sub DataPortal_Fetch(ByVal Key As Object)
        'ADO.NET Queries for Fetching (Select/From/Where) or Stored Procedures

        'Data Access Code Here!


        'At the end, set New flag to False.  NOT Dirty since found in database
        MarkOld()
    End Sub
```

Business Rules & DataPortal_Update() method

- This method UPDATES the object in the database using **ADO.NET**.
- **IMPORTANT!** – After updating, since this object exists in the database, we need to mark it **OLD**. Remember that marking and object ***OLD*** also marks it ***CLEAN***. Call the ***MarkOld()*** method at the end of the method.
- Implementation is as follows:

```
    'Data Access Code to Update an Objects data to database
    Protected Overrides Sub DataPortal_Update()
        'ADO.NET Queries for updating (Update/Set/Where) or Stored Procedures

        'Data Access Code Here!


        'Set New flag to False since exist in database/and is Not dirty any longer
        MarkOld()
    End Sub
```

Business Rules & DataPortal_Insert() method

- This method INSERTS a new record to the database using **ADO.NET**.
- IMPORTANT! – Since this object was just inserted and NOW exists in the database, we need to mark it OLD. Call the ***MarkOld()*** method at the end of the method.
- Implementation is as follows:

```
    'Data Access Code to insert a new object to database
    Protected Overrides Sub DataPortal_Insert()
        'ADO.NET Queries for Inserting (Insert/Into) or Stored Procedures

        'Data Access Code Here!

        'Set New flag to False since exist in database/and is Not dirty any longer
        MarkOld()
    End Sub
```

Business Rules & DataPortal_DeleteObject() method

- This method DELETES a record from the database using **ADO.NET**.
- **IMPORTANT!** – Deleting an object from the database, means that the Object is new **NEW**, since it does not exist in the database any more, we need to mark it NEW. Call the ***MarkNew()*** method at the end of the method.
- Implementation is as follows:

```
    'Data Access Code to immediatly delete an object from database.
    Protected Overrides Sub DataPortal_DeleteObject(ByVal Key As Object)
        'ADO.NET Queries for deleting (Delete/From/Where) or Stored Procedures

        'Data Access Code Here!

        'Object no longer in database, therefore reset our status to be a new object
        MarkNew()
    End Sub
```

5.9.2 Implementing Validation Business Rule

- In this section we implement the validation rules.
- Validation is performed in the PROPERTY methods of the object.
- The validation process usually occurs in the **SET** portion of a property where modification takes place.
- Validation involves using program code to verify that the value passed into a Property SET is within the expected data type, length, size, not empty etc.
- Validation usually involves the following:

  - Use *If/Else* and other *VB.NET statements* to accomplish the test and perform and action based on the results
  - The action usually involves ***Throwing and Exception***.

- Examples of validation business rules are:

  - **BLANK** Property – A property is left blank or empty. For example, in a School Management Program, the student's SS Number can never be blank, therefore we need to validate for this rule.
  - **MAXIMUN-LENTH** Property – Some properties may require that the string be kept within a certain length.
  - **EXACT-LENTH** Property – Property where the length must be exact. Example SSNUmber etc.
  - **WRITE-ONCE** Property – Some properties require that the value can only be set once and can never change. Example, SSNumber, LicenceID etc.

- Again, the idea is that when any of these rules are broken, we need to do handle this and let the User-interface that a rule was broken.
- Due to time constraints, we will NOT be implementing a more sophisticated mechanism, so we will simply raise exceptions.
- We will show the code required for the ***Class Developer*** as well as what the ***User-Interface Developer*** needs to do.

Maximum-Length String Business Rule

- ***Maximum-Length String Properties*** refers to a property that cannot exceed the length of a particular value. For example if the maximum value we want the Name property to contain under 50 characters, then we need to test for this length. If the length is exceeded, then we *Throw* a ***NotSupportedException***.

Implementing Max-Length inside Class Property:

- Example of this code is as follows:

```
    Public Property Name() As String
        Get
            Return strName
        End Get
        Set(ByVal Value As String)
            'Maximum-lengh property
            If Len(Value) > 50 Then
                Throw New NotSupportedException("Name too long")
            End If
            strName = Value

            MyBase.MarkDirty()  'Mark Ojbect as dirty
        End Set
    End Property
```

Handling Max-Length in User-Interface or Client:

- Now we need to know how to code the MAX-LENGTH rule in the User Interface (Forms, Clients etc).
- Since what the rule does is throw a ***NotSupportedException***, we need to trap for this exception in the client program and display the error message returned from the Business Object.
- Example of this is as follows:

```
        Try

            'Step x-Traps for Business Rule violations & Display Error Message
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation!  " & objNSE.Message)

        End Try
```

Implementing Write-Once Properties

- ***Write -Once Properties*** refers to a property that is only written once and cannot be changed once is written.

- This is an excellent technique to use for unique key values that identify an object and once entered can no longer be changed.
- For example a ***CustomerID*** value or ***SSN* number**, ***LicenseNum***, etc.

- ***Write -Once Properties*** are implemented by testing the new flag = flgIsNew, if this flag is TRUE, then we can allow the Set portion of the property to execute, otherwise we cannot allow this property to run if this object is NOT NEW, which means the value has been already set.
- **IF A PROPERTY IS GIVEN A WRITE-ONCE RULE, IN YOUR CODE, YOU CANNOT ATTEMPT TO SET THAT PROPERTY ANYWHERE IN YOUR CODE WHERE THE OBJECT IS OLD. FOR EXAMPLE** THE Edit() method.

Implementing Write-Once inside Class:

- **Write-Once Properties** are implemented as follows:

  I. In the **Property Set** portion of a Property statement, we test the status of the ***IsNew*** flag to implement this logic :

```
    'Write-Once Property
    Public Property IDNumber() As Integer
        Get
            Return intIDNumber
        End Get
        Set(ByVal intTheID As Integer)
            If Not Me.IsNew Then
                Throw New NotSupportedException("Write-Once Property already set")
            Else
                intIDNumber = intTheID

         MyBase.MarkDirty()   'Must be in Every Set Property
            End If
        End Set
    End Property
```

Handling WRITE-ONCE in User-Interface or Client:

- We need to know how to handle this Business Rule in the UI.
- Since the rule throw a ***NotSupportedException***, we need to trap for this exception in the client program and display the error message returned from the Business Object.
- Again, is the same code as before:

```
        Try
            'Step x-Traps for Business Rule violations & Display Error Message
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation!  " & objNSE.Message)
        End Try
```

Implementing NO BLANK/EMPTY String Rule

- ***No Blank/Empty Properties*** refers to a property that cannot be left blank or 0 in an Object.

- Examples of this rule such as the SSN or CustomerID which cannot be left blank, they must be populated since they usually represent a Primary Key in the database.

Implementing NO BLANK inside Class:

- ***No Blank or Empty Properties*** are implemented by verifying if the length of the string is empty:

I. In the **Property Set** portion of a Property statement, enter code to verify the length = 0:

```
    Public Property Address() As String
        Get
            Return strAddress
        End Get
        Set(ByVal Value As String)

            If Len(Trim(Value)) = 0 Then
                Throw New NotSupportedException("Value is empty")
            End If

            strAddress = Value

          MarkDirty()  'Must be in Every Set Property
        End Set
    End Property
```

Handling NO-BLANK/EMPTY Rule in User-Interface or Client:

- Again, we need to trap for a ***NotSupportedException***, and display the error message:

```
        Try


            'Step x-Traps for Business Rule violations & Display Error Message
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation!  " & objNSE.Message)

        End Try
```

Implementing EXACT-LENGTH Rule

Implementing the EXACT-LENGTH Rule

- ***Exact-Length Properties*** refers to a property that length of the string must be of an exact size.

- Examples of this rule such as the *SSN* which the size must be exactly 11 characters (including – character) or a Phone number which must be say 14 characters: (718)-260-5000.

Implementing EXACT-LENGTH inside Class:

- ***Exact-Length Properties*** are implemented by comparing the length is within a range
- This mechanism is implemented as follows.

```
    Public Property Phone() As String
        Get
            Return strPhone
        End Get
        Set(ByVal Value As String)
            'Enforce exact-lenght: (212)-555-1212
            If (Len(Trim(Value)) <> 14) Then
                Throw New NotSupportedException("Value not exact Lenght")
            End If

            strPhone = Value
         MarkDirty()  'Must be in Every Set Property
        End Set
    End Property
```

Implementing EXACT-LENGTH Rule in User-Interface or Client:

- Again, we need to trap for a ***NotSupportedException***, and display the error message:

```
        Try

            'Step x-Traps for Business Rule violations & Display Error Message
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation!  " & objNSE.Message)

        End Try
```

5.9.3 Constructor Methods & Business Rules

- As we know, when we create an object, constructors execute, such as default and parameterized constructors.
- These constructors modify data! They either set the data to default values or assign data to the parameters
- We have to options:

  - Modify via the Private Data – Modifies private data directly, but we have no way of knowing or checking if the data modified satisfy our validations rules. This is more of a concern when this data is being passed as parameters to the parameterized constructor.

  - Modify via Public Properties – Using Public Properties guarantees that the property validation mechanism catches any issues.

- With this said, we will do the following:

  1. Assign the ***Default constructor*** to Private Data directly – We don't have to concern ourselves with the default since we control it from within the class.
  2. Assign the ***Parameterized Constructor*** to the PROPERTY PROCEDURES. We don't have control of what the UI developer will pass as arguments to objects so we need to make sure they are within our validation rules.

Implementing the Default Constructor method

- No changes required here, if you are using the Private Data to initialize the default constructor.

```
Public Sub New()
    'Note that private data members are being initialized
    strName = ""
    intIDNumber = 0
    dBirthDate = #1/1/1900#
    strAddress = ""
    strPhone = "(000)-000-0000"
    intTotalItemsPurchased = 0

End Sub
```

- Note that if you decide to use the Properties instead of the private data directly, the default data that you enter, must satisfy the Business Rules dictated by the property otherwise you will yield errors.

Implementing the Parameterized Constructor method

- In this case we will assign the argument parameters to the Properties instead of the private data.
- By doing this we make sure that when an object is created and data is passed to the object upon creation, that data must satisfy the Business rules.
- Implementation is as follows:

```
Public Sub New(ByVal strN As String, ByVal intIDNum As Integer, ByVal bBDate As Date,
_
 ByVal strAdr As String, ByVal strPh As String)
    'Note that we are NOT using the private data but the Property Procedures instead
    Me.Name = strN
    Me.IDNumber = intIDNum
    Me.BirthDate = bBDate
    Me.Address = strAdr
    Me.Phone = strPh
    Me.TotalItemsPurchased = 0

End Sub
```

5.9.4 Listing of all Base Classes & Templates (Summary)

- So this is what we have so far:
  - **BusinessBase** – Base Class for our Business Classes. **BusinessCollectionBase** – Base Class for Business Collection Classes:

Imports
<Serializable()> _
Class MustInherit clsBusinessBase

Private Business Rules data:
mflgIsDirty, mflgIsNew

Public Business Rules Properties:
IsNew, IsDirty

Public *MustOverride* Data Access Methods:
Create()
Load(Key)
DeleteObject(Key)
Save()

Protected *MustOverride* Data Access Methods:
DataPortal_Create()
DataPortal_Fetch(Key)
DataPortal_Update()
DataPortal_Insert()
DataPortal_DeleteObject(Key)

Public Helper Data Access Methods:
DBConnectionString(DBName)

Imports
Class MustInherit clsBusinessCollectionBase
Inherits DictionaryBase

Public Business Rule Properties:
IsDirty

Public *MustOverride* Data Access Methods:
Create()
Load()
DeleteObject(Key)
Save()

Protected MustOverride Methods:
DataPortal_Create()
DataPortal_Fetch()
DataPortal_Save()
DataPortal_DeleteObject(Key)

Public Helper Data Access Methods:
DBConnectionString(DBName)

  - **BusinessClass Template** & **BusinessCollectionClass Template** – INHERITED from *BUSINESSBASE* for creating our REGULAR CLASSES & *BUSINESSCOLLECTIONBASE* for our COLLECTION CLASSES.

Imports
<Serializable()> _
Class clsBusinessClass
Inherits clsBusinessBase

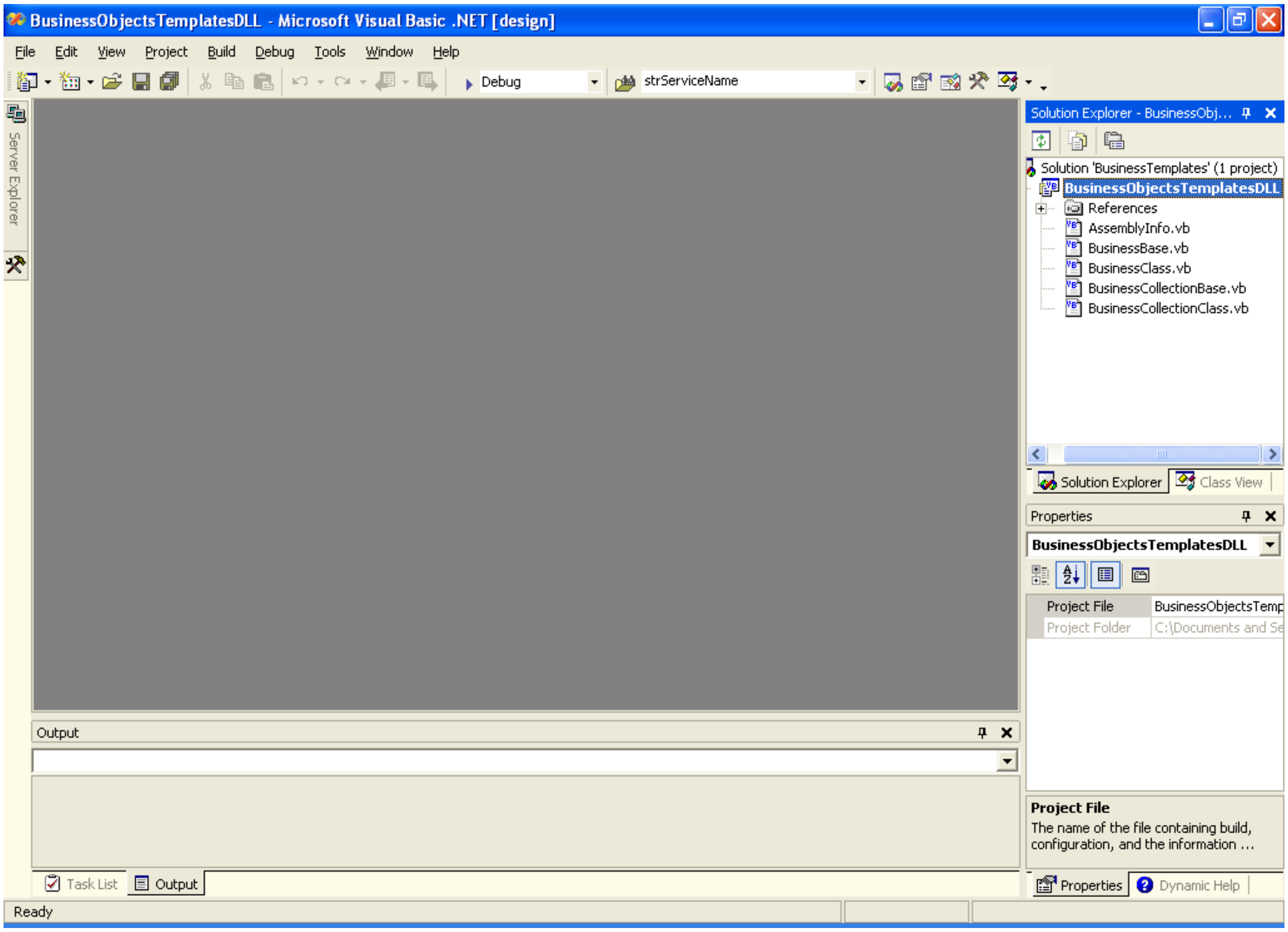
Private data:
Public Event Declarations:
Public Properties:
Public Constructors:
Public Methods:
Public *Shared* Data Access Methods:
Create()
Load(Key)
DeleteObject(Key)
Save()
Protected *Override* Data Access Methods:
DataPortal_Create()
DataPortal_Fetch(Key)
DataPortal_Update()
DataPortal_Insert()
DataPortal_DeleteObject(Key)
Public Helper Methods:

Imports
<Serializable()> _
Class clsBusinessCollectionClass
Inherits clsBusinessCollectionBase

Public Properties:
Public Wrapper Methods:
Public Regular Methods:
Public *Shared* Data Access Methods:
Create()
Load()
DeleteObject(Key)
Save()

Protected *Override* Data Access Methods:
DataPortal_Create()
DataPortal_Fetch()
DataPortal_Save()
DataPortal_DeleteObject(Key)
Public Helper Methods:

- At this point, we implemented a DLL component Project that contains our Base Classes & Templates, for us to use in our programs:


BusinessObjectsTemplatesDLL - Microsoft Visual Basic .NET [design]
File Edit View Project Build Debug Tools Window Help
Debug
strServiceName
Server Explorer
Solution Explorer - BusinessObj...
Solution 'BusinessTemplates' (1 project)
BusinessObjectsTemplatesDLL
References
AssemblyInfo.vb
BusinessBase.vb
BusinessClass.vb
BusinessCollectionBase.vb
BusinessCollectionClass.vb
Solution Explorer
Class View
Properties
BusinessObjectsTemplatesDLL
Output
Task List
Output
Properties
Dynamic Help
Ready

Project File	BusinessObjectsTemp
Project Folder	C:\Documents and Se

Project File
The name of the file containing build, configuration, and the information ...

- Going forward, when we create applications, we can use these base classes and templates for our Business Objects projects:

5.9 User-Interface Support for Business Objects

5.9.1 Overview

- Ok, now that we have gone thought the Business object Layer. We need to address the ***User-Interface Layer***.
- What we need to know is what needs to be done in our Forms or UI to support the Business Objects.
- What does our User-Interface Developer needs to know so they can use our Business Objects.

5.9.2 Programming the UI to use the Business Objects

- For starters we know the following:

1. UI will create Business Objects and use them.
2. UI will call Regular Public Methods & Properties to make the object behave as its real-world counterpart. Some of these methods modify data.
3. UI will also call Business Rules Public Properties to track the STATUS of the Business Object, such as **IsDirty** & **IsNew**
4. UI will also call Business Rules Public Data Access Methods: **Create()**, **Load()**, **Save()** & **DeleteObject()**

- So now let's address each one of these tasks at a time, see what needs to be done:

1. **UI will create Business Objects and use them.**

How is done: Create Object using default or Parameterized values

How Business Object React:
- BO will throw a **NotSupportedException** if the values passed to the parameterized constructor are in violation of Validation Business Rules: NO-BLANK, MAXIMUM-LENGTH, and WRITE-ONCE etc.

How User-Interface Should React:
- Trap for a **NotSupportedException**.

2. **UI will call Regular Public Methods & Properties to make the object behave as its real-world counterpart.**

How is done: Call Properties or Methods using normal syntax: Object.Property or Object.Method()

How Business Object React:
- If the Property or Method creates temporary BO's and uses them, BO will throw a **NotSupportedException** if the values assigned to the temporary objects are in violation of Validation Business Rules: NO-BLANK, MAXIMUM-LENGTH, and WRITE-ONCE etc.
- If the Property or Method MODIFIES the OBJECT, BO's will mark the Object as Dirty.

How User-Interface Should React:
- Trap for a **NotSupportedException**.

3. **UI will call Business Rules Public Properties to track the STATUS of the Business Object, such as IsDirty & IsNew.**

How is done: Call Properties using normal syntax: Object.Property

How Business Object React:

- Returns a TRUE or FALSE depending on the Status of the Object.

How User-Interface Should React:

- Take any necessary action based on these the True/False results.

4. **UI will also call Business Rules Public Data Access Methods: Load(), Save() & DeleteObject()**

How is done: Call Methods using normal syntax: Object.Method()

How Business Object React:

- Perform the data access
- Marks the Object as Dirty, New etc based on the data access method called.

How User-Interface Should React:

- Nothing or may need to trap for Exceptions generated by ADO.NET code.

Final Summary

- From our analysis of how the UI performs the operations listed and how the Business Object reacts we can conclude the following:

  - UI uses the object (Properties & method calls) and let's the object perform the requested operation
  - UI needs to trap for the **NotSupportedException** in case the UI violates the rules.
  - UI can use a Try-catch Block to trap for this exception and Handle the exception as required.

- So the UI developer needs to be aware of the exception and use a Try-Catch Block to trap and handle appropriately.

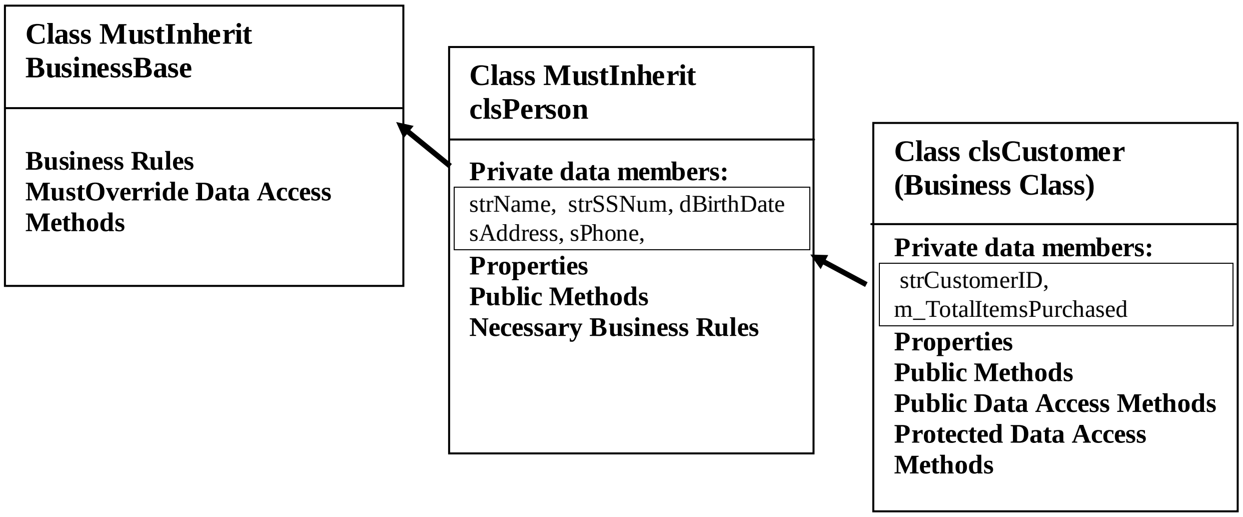
6.1 Sample Program #5 – Customer Management Business Objects Program

6.1.1 Overview

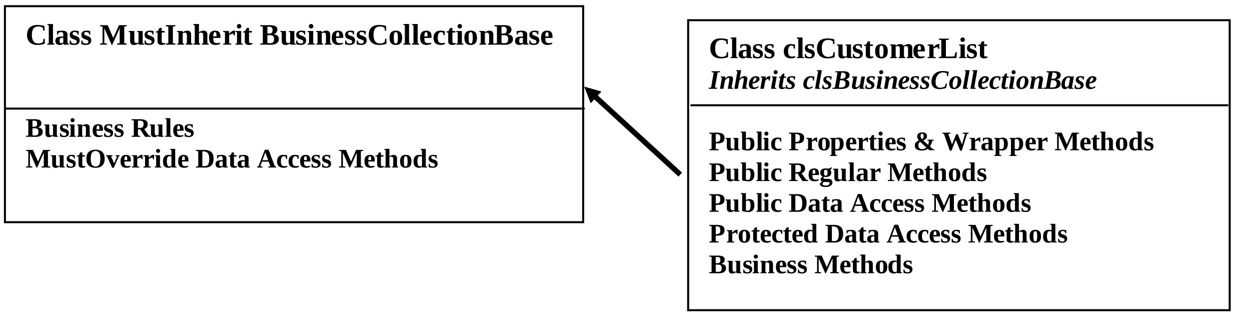
- We will now upgrade the *Customer Management Application* from previous lecture, which resembles the class project. We will inherit from ***BusinessBase***, ***BussinessCollectionBase*** and implement our Business Classes following the rules and format of the ***BusinessClass***, ***BussinessCollectionClass*** templates.
- In summary we will add the following new functionality:

1. Inheritance & Business Object requirements using ***BusinessBase***, ***BussinessCollectionBase***, and ***BusinessClass***, ***BussinessCollectionClass*** templates:

- The ***clsPerson*** Class will now **inherit** from ***BusinessBase*** class.
- Continue to **Inherit** the ***clsCustomer*** from ***clsPerson*** class.
- Modify ***clsCustomer*** to adhere to the ***BusinessClass*** template
- The ***clsCustomerList*** will now **Inherit** from ***BussinessCollectionBase***.
- Modify ***clsCustomerListManager*** to adhere to the ***BussinessCollectionClass*** template
- Maintain all Business Template logic within this new inheritance scheme.
- The new object model should look as follows for the Business Classes:


Class MustInherit BusinessBase
Business Rules
MustOverride Data Access Methods
Class MustInherit clsPerson
Private data members:
strName, strSSNum, dBirthDate sAddress, sPhone,
Properties
Public Methods
Necessary Business Rules
Class clsCustomer (Business Class)
Private data members:
strCustomerID, m_TotalItemsPurchased
Properties
Public Methods
Public Data Access Methods
Protected Data Access Methods

- The Collection Class hierarchy looks as follows:


Class MustInherit BusinessCollectionBase
Business Rules
MustOverride Data Access Methods
Class clsCustomerList
Inherits clsBusinessCollectionBase
Public Properties & Wrapper Methods
Public Regular Methods
Public Data Access Methods
Protected Data Access Methods
Business Methods

1. We will enforced **Dirty Objects** to ALL OUR PROPERTY SET:

   - Customer Name: Call MARK-DIRTY()
   - Social Security & Customer ID Number – Call MARK-DIRTY()
   - Address, & Phone – Call MARK-DIRTY().

2. We will enforced the following **Field-Level Validation** to our Properties:

   - Customer Name – NO-BLANK & MAX-LENGTH.
   - Social Security & Customer ID Number – WRITE-ONCE, EXACT LENGTH & NO-BLANK/EMPTY
   - Address, & Phone – NO-BLANK/EMPTY.

3. In addition we will CUT/PASTE **FILE ACCESS CODE** from the current load() & save() to the ***clsCustomerList*** DATA ACCESS METHODS, ***DataPortal_Fetch()*** & ***DataPortal_Save()*** in order to permanently store our data and simulate the database partially:

   - In the *CustomerList* Collection we include File Access code to Load & Save the Customer Child Objects with data from a comma-delimited file.
   - A file named ***Customers.txt*** is used to store the data.
   - NOTE! We will keep all Business Object structure as is. The Business Methods and properties should not be modified in any way.

6.1.2 Problem Statement

- ❑ The requirements for Sample program #5. are as follows:

Example #5 – Business Object Customer Management Application (Version 2)

Problem statement:

- ❑ Upgrade the Customer Management application as described in previous Overview section.

Business Object Layer – Business Class & DLL Requirements

- ❑ Implement the following classes:
  - **clsPerson Class** – MustInherit Class that inherits from ***BusinessBase***. Details in code to follow
  - **clsCustomer Class** – Inherit from ***clsPerson***. Details in code to follow
  - **clsCustomerList Collection Class** – Inherits from ***BusinessCollectionBase***:

    - Derive this class from ***BusinessCollectionBase***.
    - In the **DataPortal_Fetch()** Add File Access Code to load data from the ***Customer.txt*** file and populate the collection with data read from file.
    - In the **DataPortal_Save()**, add File Access Code from current Load() & Save() method to the new ***DataPortal_Fetch()*** & ***DataPortal_Save()*** in order to permanently store the data to ***Customer.txt*** file.

Presentation/UI Layer – Client Process requirements:

- Same as previous Customer Manager Example

HOW IT'S DONE:

The Component or DLL

Part I – View The Class Library Project:

Step 1: Open the Customer Management Application from Previous DLL Example

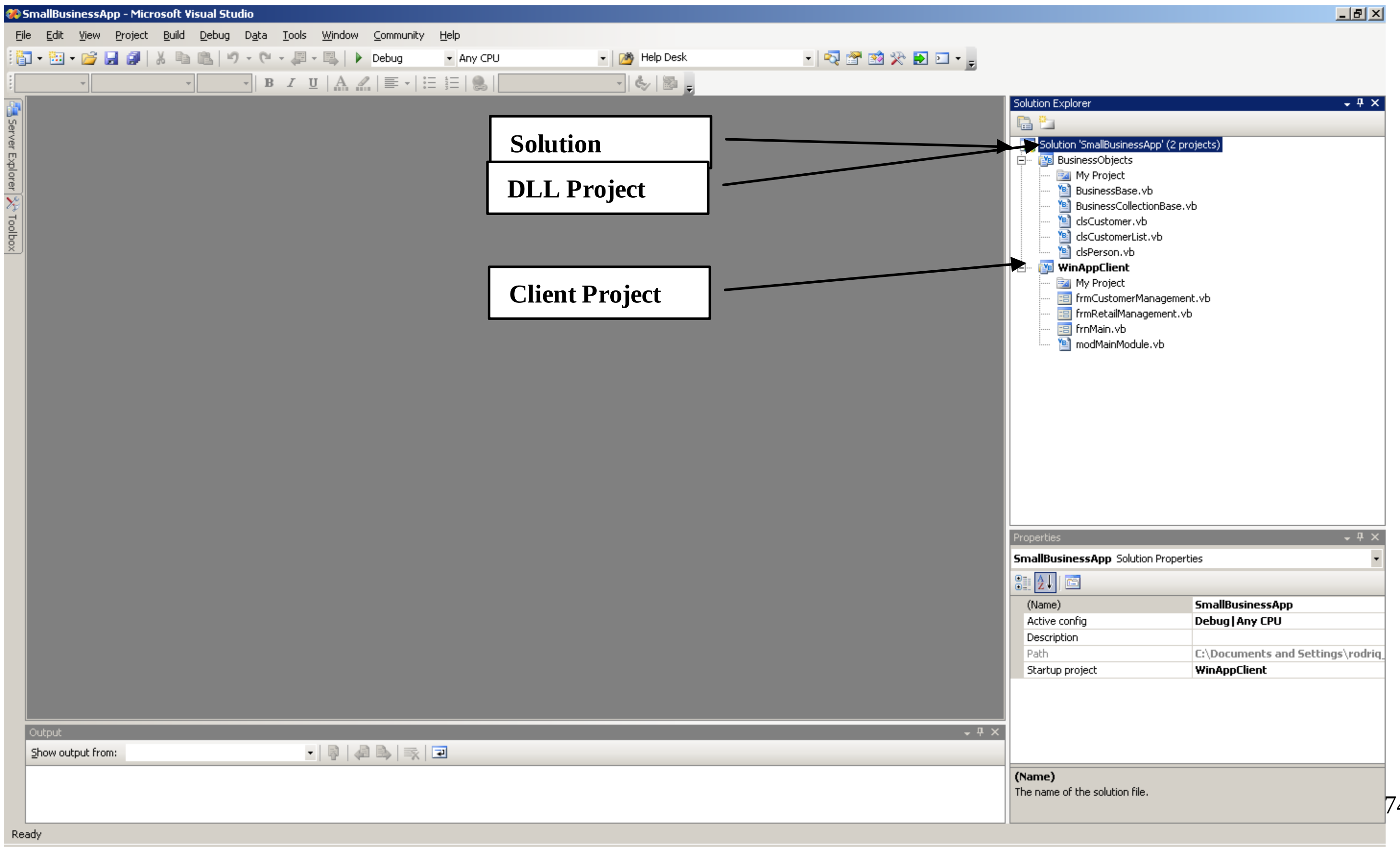
- In the previous example in ***Lecture 2B Sample Program #23*** on page 29, we converted the CUSTOMER RETAIL MANAGEMENT APPLICATION TO USE A DLL COMPONENT.
- The high-level steps are as follows:

  1. Created a **Blank Solution & added a NEW DLL Project**
  2. Copied the CUSTOMER RETAIL APPLICATION or **Client Project** FOLDER from previous application into this Blank Solution FOLDER STRUCTURE.
  3. We then ADDED CUSTOMER MANAGEMENT client into the Solution.
  4. We renamed the solution to WinAppClient
  5. Made the WinAppClient the STARTUP OBJECT, since it is an executable, it will now control the application
  6. We MOVED ALL CLASSES TO THE DLL PROJECT
  7. Set REFERENCE on the WinAppClient to POINT TO THE DLL COMPONENT
  8. Modified all code in the application were the CLASSES were being used to take into account that the classes NOW RESIDE INSIDE THE DLL using the syntax: **DLL.CLASS**, example: ***BusinessObjects.clsCustomer***

- If you have not done so, follow the steps to convert the Customer Management application to use a Class Library from our previous example and notes ***Lecture 2B Sample Program # 2*** on page 29.

Step 1: View of Solution at this point:

- The entire solution looks as follows:



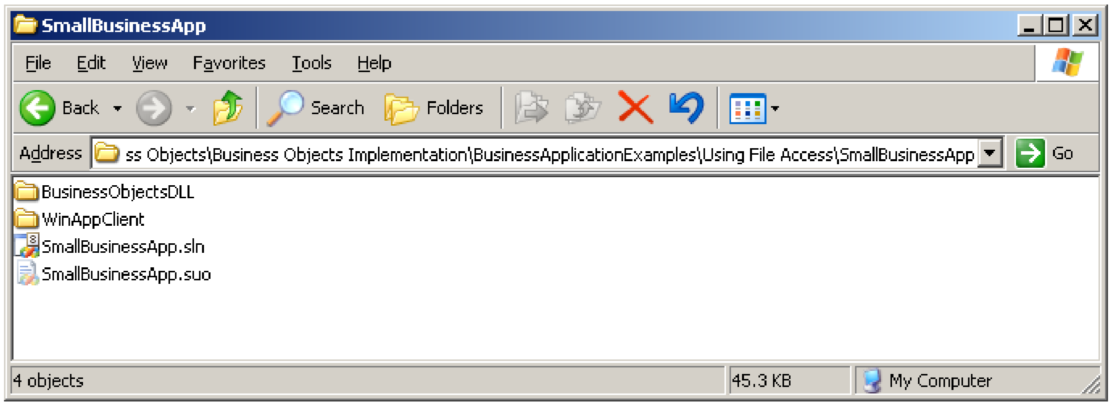
```
Solution 'SmallBusinessApp' (2 projects)
  BusinessObjects
    My Project
    BusinessBase.vb
    BusinessCollectionBase.vb
    clsCustomer.vb
    clsCustomerList.vb
    clsPerson.vb
  WinAppClient
    My Project
    frmCustomerManagement.vb
    frmRetailManagement.vb
    frmMain.vb
    modMainModule.vb
```

SmallBusinessApp Solution Properties

(Name)	SmallBusinessApp
Active config	Debug|Any CPU
Description	
Path	C:\Documents and Settings\rodriq
Startup project	WinAppClient

(Name)
The name of the solution file.

- The file structure looks as follows:


SmallBusinessApp
File Edit View Favorites Tools Help
Back Search Folders
Address ss Objects\Business Objects Implementation\BusinessApplicationExamples\Using File Access\SmallBusinessApp
Go

BusinessObjectsDLL
WinAppClient
SmallBusinessApp.sln
SmallBusinessApp.suo

4 objects 45.3 KB My Computer

Business Object Layer (Business Classes)

Overview

- We need to add the ***BusinessBase*** & ***BusinessCollectionBase*** Classes so our Business Classes (clsPerson, clsCustomer & clsCustomerList) can inherit all the Business Rules.
- We also need the methods that we need to implement in our classes and are contained in the ***BusinessClass*** & ***BusinessCollection*** Class templates. Since we are NOT starting from scratch we don't want to use these *Business Class* Templates as a starting point. So what we are going to do is simply copy what we need from them into our existing classes to turn them into Business Classes and save us some typing.
- I provided business class & business collection class templates for your use.
- Open these templates using Visual Studio and keep them handy so you can copy what you need from them as you modify your project.

Step 2: Add Business Base & Business Collection Base Classes to Project.

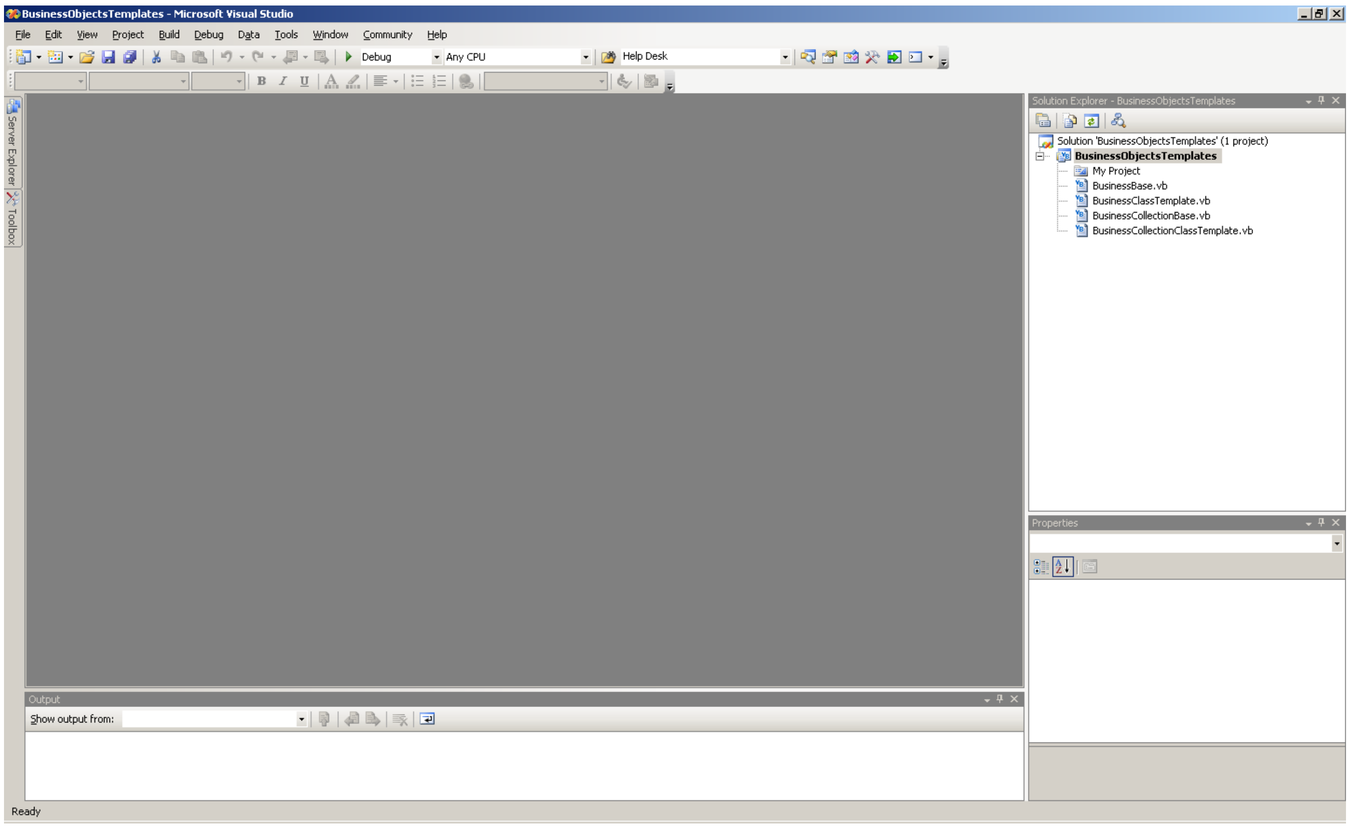
- Steps are as follows:

Step 1: Open CUSTOMER RETAIL MANAGEMENT SOLUTION (SHOULD ALREADY BE OPEN):

- At this point, you should have the ***Customer Retail Management*** solution DLL project from STEP 1 above running.

Step 2: Open THE BUSINESS OBJECTS TEMPLATES available on the WEB SITE

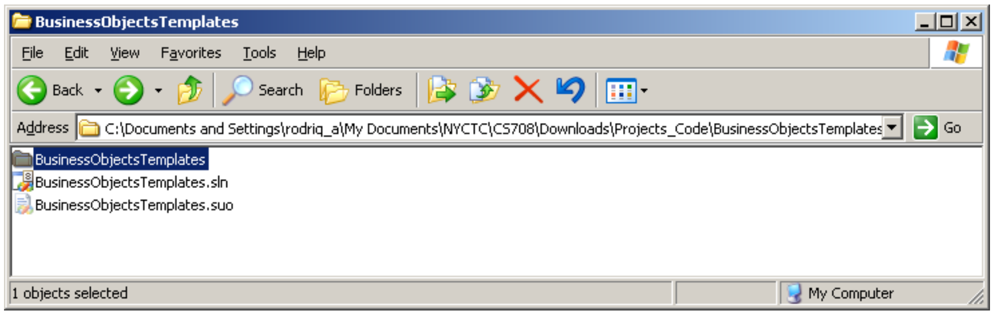
- At this point, ALSO OPEN THE BUSINESS OBJECTS TEMPLATE DLL Project I available on the COURSE WEB SITE.
- This DLL project contains all the Business Class TEMPLATES, AS WELL AS BASE CLASSES FOR ALL OUR BUSINESS CLASSES AND BUSINESS COLLECTION CLASSES


BusinessObjectsTemplates - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Debug
Any CPU
Help Desk
Server Explorer
Toolbox
Solution Explorer - BusinessObjectsTemplates
Solution 'BusinessObjectsTemplates' (1 project)
BusinessObjectsTemplates
My Project
BusinessBase.vb
BusinessClassTemplate.vb
BusinessCollectionBase.vb
BusinessCollectionClassTemplate.vb
Properties
Output
Show output from:
Ready

Step 3: COPY BASE CLASSES FILES FROM TEMPLATE DLL PROJECT TO CUSTOMER MANAGEMENT PROJECT:

- Now we need to navigate to the folder containing the ***BusinessBase*** & ***BusinessCollectionBase*** classes and copy/paste into our project.
- Steps are as follows:

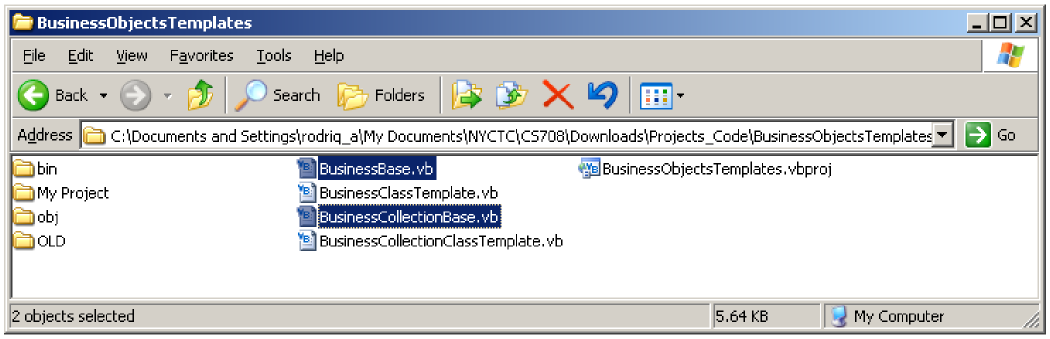
1. Using *Widows Explore* or *My Computer*, navigate to the BUSINESS OBJECTS DLL TEMPLATE PROJECT FOLDER where the ***BusinessBase*** & ***BusinessCollectionBase*** classes are located:



C:\Documents and Settings\rodriq_a\My Documents\NYCTC\CS708\Downloads\Projects_Code\BusinessObjectsTemplates

BusinessObjectsTemplates
BusinessObjectsTemplates.sln
BusinessObjectsTemplates.suo

1 objects selected | My Computer



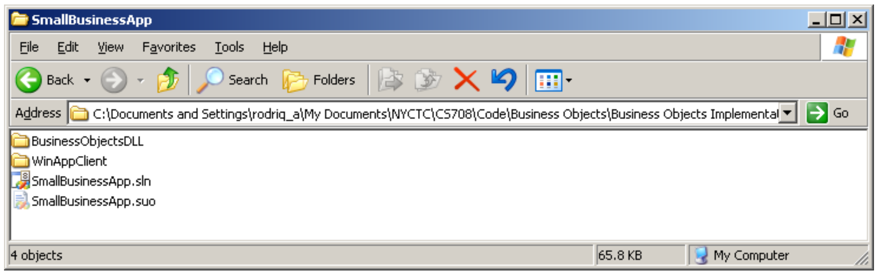
C:\Documents and Settings\rodriq_a\My Documents\NYCTC\CS708\Downloads\Projects_Code\BusinessObjectsTemplates

bin
My Project
obj
OLD
BusinessBase.vb
BusinessClassTemplate.vb
BusinessCollectionBase.vb
BusinessCollectionClassTemplate.vb
BusinessObjectsTemplates.vbproj

2 objects selected | 5.64 KB | My Computer

2. Right-Click & COPY the two base classes: ***BusinessBase*** & ***BusinessCollectionBase***

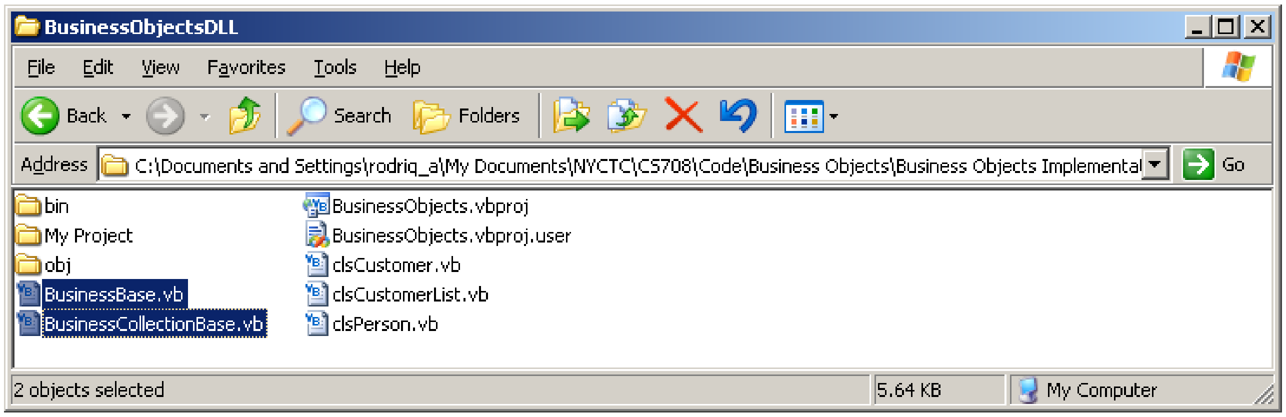
3. Now navigate to the TARGET LOCATION IN YOU'RE THE CUSTOMER MANAGEMENT SOLUTION **BUSINESSOBJECTS** DLL COMPONENT & <u>**PASTE**</u> the two base classes: ***BusinessBase*** & ***BusinessCollectionBase***:



C:\Documents and Settings\rodriq_a\My Documents\NYCTC\CS708\Code\Business Objects\Business Objects Implementat

BusinessObjectsDLL
WinAppClient
SmallBusinessApp.sln
SmallBusinessApp.suo

4 objects | 65.8 KB | My Computer



C:\Documents and Settings\rodriq_a\My Documents\NYCTC\CS708\Code\Business Objects\Business Objects Implementat

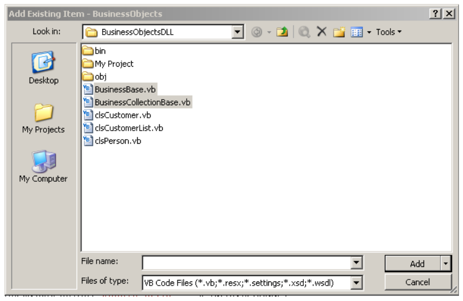
bin
My Project
obj
BusinessBase.vb
BusinessCollectionBase.vb
BusinessObjects.vbproj
BusinessObjects.vbproj.user
clsCustomer.vb
clsCustomerList.vb
clsPerson.vb

2 objects selected | 5.64 KB | My Computer

Step 4: ADD *BusinessBase & BusinessCollectionBase* to the CUSTOMER MANAGEMENTSOLUTION:

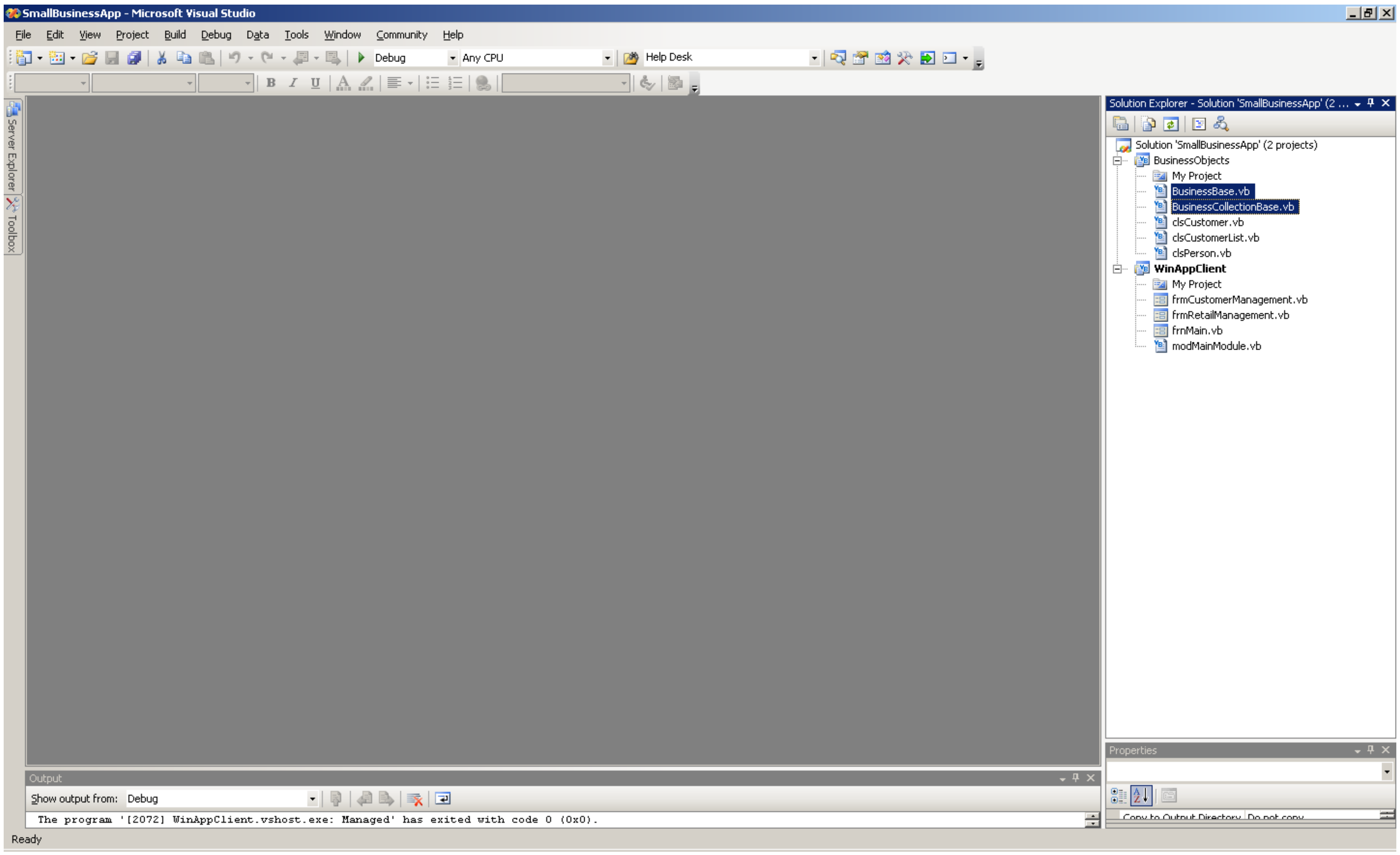
- ❑ Now we ADD the TWO BASE CLASSES (***BusinessBase*** & ***BusinessCollectionBase***) to the Solution.
- ❑ Steps are as follows:

1. Go or open the Customer Management Solution, if you have not done so
2. In the *Solution Explore window*, RIGHT-CLICK the BusinessObjects DLL COMPONENT PROJECT
3. In the drop-down menu, select ***ADD|EXISTING ITEM…***
4. Navigate to the Client or WinAppClient project and select and add the two base classes (***BusinessBase*** & ***BusinessCollectionBase***) just copied to that folder:



Step 5: CUSTOMER MANAGEMENT APPLICATION NOW HAS THE TWO BASE CLASSES AS PART OF THE DLL COMPONENT:

- ❑ The BusinessObjects DLL COMPONENT now contains the ***BusinessBase*** & ***BusinessCollectionBase*** to serve as the base classes for ALL OUR CLASSES & COLLECTION CLASSES:



```
The program '[2072] WinAppClient.vshost.exe: Managed' has exited with code 0 (0x0).
```

Step 3: Modify the clsPerson TO USE THE BUSINESSBASE CLASS

- We need to modify the ***clsPerson*** Class to contain the required Business rules based on the ***BusinessClassTemplate*** as follows:

  1. The ***clsPerson*** Class **HAS TO BE** a **MustInherit** Class, otherwise, we will be **FORCED** to implement the **MUSTOVERRIDE** Data Access Methods of the ***BusinessBase*** Class (Load(), Save(), DeleteObject() etc.) here in ***clsPerson***.
  - We **DON'T** want to implement the **MustOverride** Business methods of the ***BusinessBase*** here because Person is not a complete Customer. Only the Customer class contains all the data necessary for the application, therefore it is in CUSTOMER that we will implement all the **MustOverrided** Business methods.
  - Since ***clsPerson*** is a *MustInherit* Class, we cascade all the FORCED MustOverride Business methods from ***BusinessBase*** to the inherited ***Customer*** class.

  2. Nevertheless, ***clsPerson*** contains private data and properties which need to adhere to our business rules, such as **MarkDirty()** and validation rules, etc.
  3. We also need to copy all the Inports and ***Serialization*** tag to make this class an **UnAnchored Class**.
  4. THE CLSPERSON CLASS DOES NOT REQUIR DATA ACCESS METHOD. IT IS A MUST INHERIT BASE CLASS FOR IT'S CHILDREN clsCustomer & clsEmployee.
  5. At the end of this section, the structure of the ***clsPerson*** class should look like our BusinessClass Template.

- Perform the following steps:

Step 1: AT THIS POINT, MAKE SURE BOTH YOUR CUSTOMER RETAIL SOLUTION AND BUSINESS OBJECTS DLL TEMPLATE SOLUTION ARE BOTH OPEN:

- Verify both Solutions are running:

Step 2: Copy from BusinessClass Template the Imports and Serializable Tag & other IMPORT statements:

1. IN THE TEMPLATE DLL PROJECT, Open the **BusinessClassTemplate** class
2. In the header section of this class, **SELECT/COPY** all the *Imports*, declarations & the SERIALIZABLE TAG information
3. IN THE CUSTOMER RETAIL SOLUTION, OPEN THE CLSPERSON CLASS, in the top declaration section click **PASTE**
4. Make sure THERE ARE NO SPACES BETWEEN THE <Serializable()> _ TAG AND THE *clPerson* CLASS DECLARATION
5. The DECLARATION portion of the clsPerson class now looks as follows:

```
Option Explicit On
Option Strict On
Imports System.IO                                        'File/IO
Imports System.Data                                      'Data Access (DataSet)
Imports System.Data.OleDb                                'OLEDB Provider
Imports System.Configuration                             'Configuration File for DB
Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                          'Remoting
'Imports System.Runtime.Remoting.Channels                 'Remoting
'Imports System.Runtime.Remoting.Channels.Http            'Remoting

<Serializable()> _
Public MustInherit Class clsPerson
```

Step 3: INHERIT FROM BUSINESS BASE:

1. NOW we need to Inherit from BusinessBase Class
2. IN THE TEMPLATE DLL PROJECT, Open the **BusinessClassTemplate** class
3. **SELECT/COPY** THE STATEMENT TO INHERIT FROM BUSINESSBASE
4. IN THE CUSTOMER RETAIL SOLUTION, BELOW THE DECLARATION OF THE CLSPERSON CLASS, click **PASTE**
5. THE INHERIT FROM BUSINESS BASE STATEMENT IS NOW LOCATED BELOW THE CLASS DECLARATION AS EXPECTED:

```
Option Explicit On
Option Strict On
Imports System.IO                                        'File/IO
Imports System.Data                                      'Data Access (DataSet)
Imports System.Data.OleDb                                'OLEDB Provider
Imports System.Configuration                             'Configuration File for DB
Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                         'Remoting
'Imports System.Runtime.Remoting.Channels                'Remoting
'Imports System.Runtime.Remoting.Channels.Http           'Remoting

<Serializable()> _
Public MustInherit Class clsPerson
    Inherits BusinessBase  'Inherits from BusinessBase.
```

Step 4: General Class Private data:

- No changes required for Private Data.

```
#Region "Private Data"

    '*********************************************************************
    'Class Data or Variable declarations
    Private m_Name As String
    Private m_SSNumber As String
    Private m_BirthDate As Date
    Private m_Address As String
    Private m_Phone As String

#End Region
```

Step 5: Add DIRTY OBJECT Mechanism (MANDATORY!) & [OPTIONAL] add any FIELD-LEVEL VALIDATION rules To Properties:

- **IMPORTANT & MANDATORY!** Add code for implementing DIRTY OBJECTS. EVERY PROPERTY SET MUST INCLUDE THE MARKDIRTY() CALL AFTER THE DATA IS SET.
- [OPTIONAL] if required add any Field-Level Validation Business Rules:

```
#Region "Property Procedures"
    '*********************************************************************
    'Enforcing NO-BLANK, MAX-LENGTH & MARK DIRTY for Name
    Public Property Name() As String
        Get
            Return m_Name
        End Get
        Set(ByVal Value As String)

            'NO-BLANK validation
            If Len(Trim(Value)) = 0 Then
              Throw New NotSupportedException("Business Rule: Name cannot be blank")
            End If

            'MAX-LENTHG VALIDATION
            If Len(Value) > 25 Then
                Throw New NotSupportedException("Business Rule: Name is too long")
            End If

            m_Name = Value

            MyBase.MarkDirty() 'Mark Ojbect as dirty it has been modified
        End Set
    End Property

    '*********************************************************************
    'Enforcing NO-BLANK, WRITE-ONCE, EXACT-LENGTH & MARK DIRTY for Address
    Public Property SocialSecurity() As String
        Get
            Return m_SSNumber
        End Get
        Set(ByVal Value As String)

            'NO-BLANK validation
            If Len(Trim(Value)) = 0 Then
             Throw New NotSupportedException("Business Rule: SSNum cannot be blank")
            End If

            'WRITE-ONCE validation
            If Not Me.IsNew Then
               Throw New NotSupportedException("Business Rule: SSNum is Write-once")
            End If

            'EXACT-LENTH validation
            If (Len(Trim(Value)) <> 11) Then
                Throw New NotSupportedException("Value not exact Lenght")
            End If

            m_SSNumber = Value

            MyBase.MarkDirty() 'Mark Ojbect as dirty it has been modified
        End Set
    End Property
    '*********************************************************************
    'Enforcing MARK DIRTY for Birthday
    Public Property BirthDate() As Date
        Get
            Return m_BirthDate
        End Get
        Set(ByVal Value As Date)

            m_BirthDate = Value

            MyBase.MarkDirty() 'Mark Ojbect as dirty it has been modified
        End Set
    End Property

    '*********************************************************************
    'Enforcing NO-BLANK & MARK DIRTY for Address
    Public Property Address() As String
        Get
            Return m_Address
        End Get
        Set(ByVal Value As String)

            'NO-BLANK validation
           If Len(Trim(Value)) = 0 Then
            Throw New NotSupportedException("Business Rule: Address cannot be blank")
           End If

            m_Address = Value

            MyBase.MarkDirty() 'Mark Ojbect as dirty it has been modified
        End Set
    End Property

    '*********************************************************************
    'Enforcing NO-BLANK & MARK DIRTY for Phone
    Public Property Phone() As String
        Get
            Return m_Phone
        End Get
        Set(ByVal Value As String)

            'NO-BLANK validation
           If Len(Trim(Value)) = 0 Then
            Throw New NotSupportedException("Business Rule: Address cannot be blank")
           End If

            m_Phone = Value

            MyBase.MarkDirty() 'Mark Ojbect as dirty it has been modified
        End Set
    End Property

#End Region
```

Step 6: MAKE sure the PAREMETERIZED Constructors are using the class PROPERTIES AND NOT PRIVATE DATA:

- No changes required for the constructors.

```
#Region "Constructor Methods"
    '*********************************************************************
    'Class Constructor Methods

    'Default Constructor
    Public Sub New()
        'Note that private data members are being initialized
        m_Name = ""
        m_SSNumber = ""
        m_BirthDate = #1/1/1900#
        m_Address = ""
        m_Phone = "(000)-000-0000"
    End Sub

    'Parameterized Constructor
    Public Sub New(ByVal N As String, ByVal SSNum As String, ByVal BDate As Date, _
    ByVal Adr As String, ByVal Ph As String)
        'Note that Property Procedures are used when setting the data

        Me.Name = N
        Me.SocialSecurity = SSNum
        Me.BirthDate = BDate
        Me.Address = Adr
        Me.Phone = Ph

    End Sub

#End Region
```

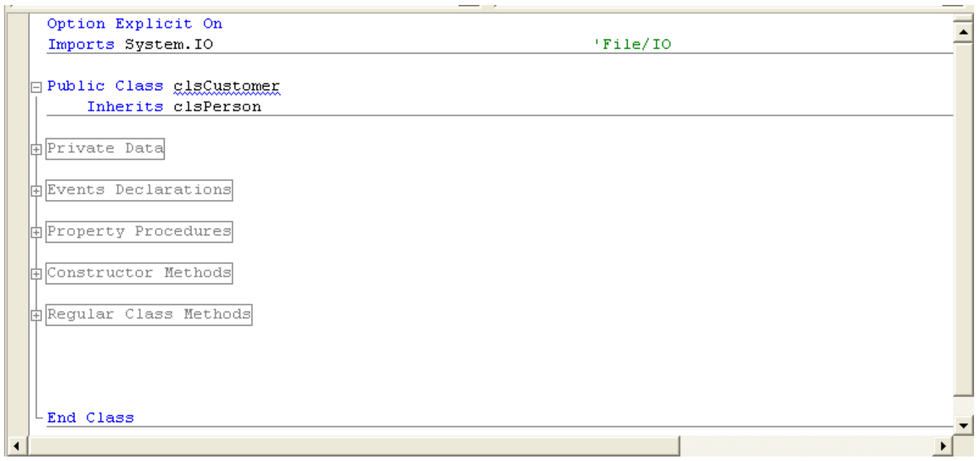
Step 7: Print Class requires NO change since it DOES NOT MODIFY DATA. NO MARKDIRTY() REQUIRED:

- No changes required to this method since no modification to data is made.

```
#Region "Regular Class Methods"

    '*********************************************************************
    '*********************************************************************
    'Class Methods
    '*********************************************************************

    'Author of base class allows sub classes to overide Print()
    'If they want to, it is not mandatory
    Public Overridable Sub Print()
        'Create StreamWriter Object for append to file listed
        Dim objPrinter As New StreamWriter("PersonPrinter.txt", True)

        'Call StreamWriter Object WriteLine method to write the string to file
        objPrinter.WriteLine(m_Name & ", " & m_SSNumber & ", " & _
        m_BirthDate & ", " & m_Address & ", " & m_Phone)

        'Close StreamWriter Object
        objPrinter.Close()

    End Sub
#End Region

End Class
```

Step 4: Modify the clsCustomer class

- Now we focus on *clsCustomer*. This is another Business Class where we implement all the Business Rules passed down from ***clsPerson*** from **BusinessBase**
- All the *MustOverride* methods enforced by the **BusinessBase** will be implemented here since we passed them down from ***clsPerson***. Once again, realize that this is ONLY possible because we made clsPerson a **<u>MustInherit</u>** Class.
- NOTE THAT YOU MAY SEE A SYNTAX ERROR INDICATION DURING THE STEPS BELOW, IGNORE THEM UNTIL ALL STEPS HAVE BEEN COMPLETED
- Currently the ***clsCustomer*** class has the following structure:



```
Option Explicit On
Imports System.IO                                        'File/IO

Public Class clsCustomer
    Inherits clsPerson

Private Data

Events Declarations

Property Procedures

Constructor Methods

Regular Class Methods



End Class
```

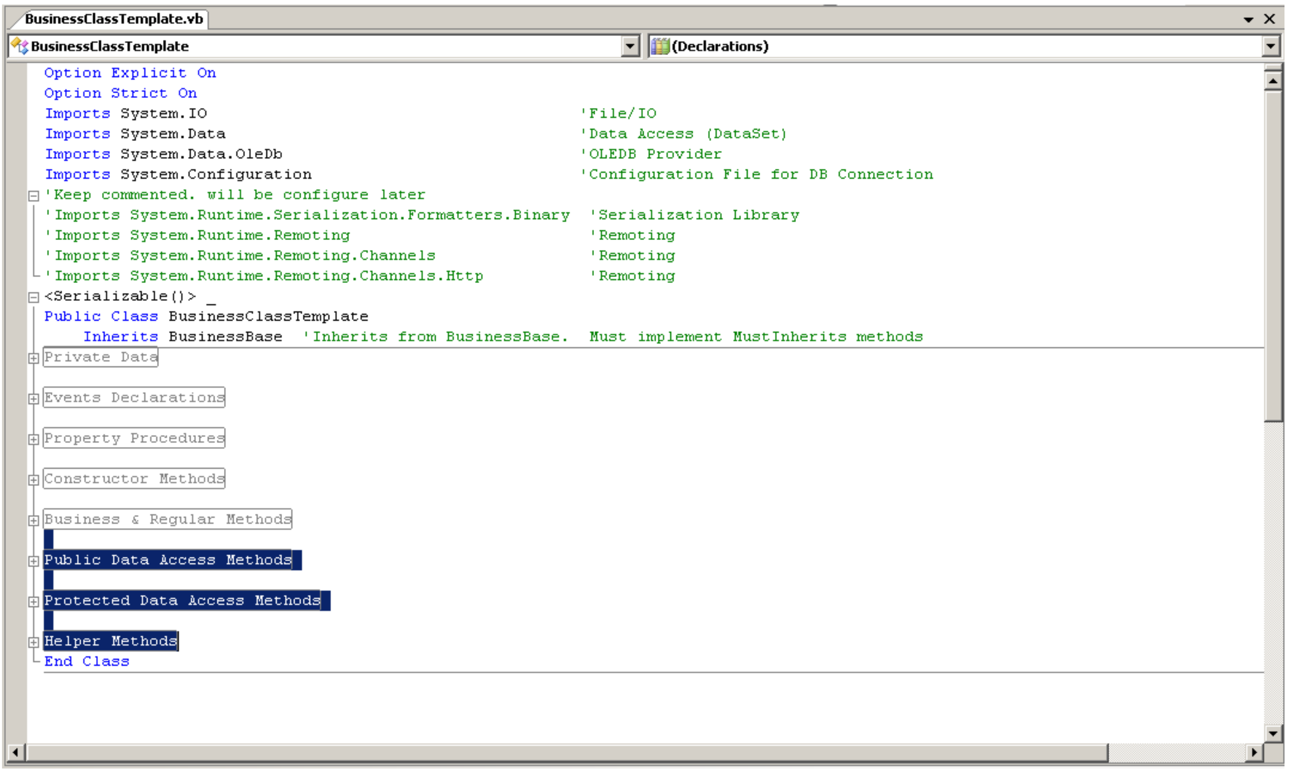
Step 1: Copy from BusinessClass Template the Imports and Serializable Tag & other IMPORT statements & PASTE TO clsCustomer:

1. Open the **BusinessClass** Template and copy/paste all the *Imports* & Serialization declarations.
2. PASTE the imports into the ***clsCustomer*** class
3. CONTINUE TO INHERIT FROM CLSPERSON
4. Make sure THERE ARE NO SPACES BETWEEN THE <Serializable()> _ TAG AND THE *clCustomer* CLASS DECLARATION
5. NOTE THAT YOU MAY SEE A SYNTAX ERROR INDICATOR IN THE CLASS, IGNORE THIS FOR NOW!
6. At the end of this section, the structure of the ***clsCustomer*** class should look as follows:

```
Option Explicit On
Option Strict On

Imports System.IO                                        'File/IO
Imports System.Data                                      'Data Access (DataSet)
Imports System.Data.OleDb                                'OLEDB Provider
Imports System.Configuration                             'Configuration File for DB
Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                         'Remoting
'Imports System.Runtime.Remoting.Channels                'Remoting
'Imports System.Runtime.Remoting.Channels.Http           'Remoting

<Serializable()> _
Public Class clsCustomer
    Inherits clsPerson
```

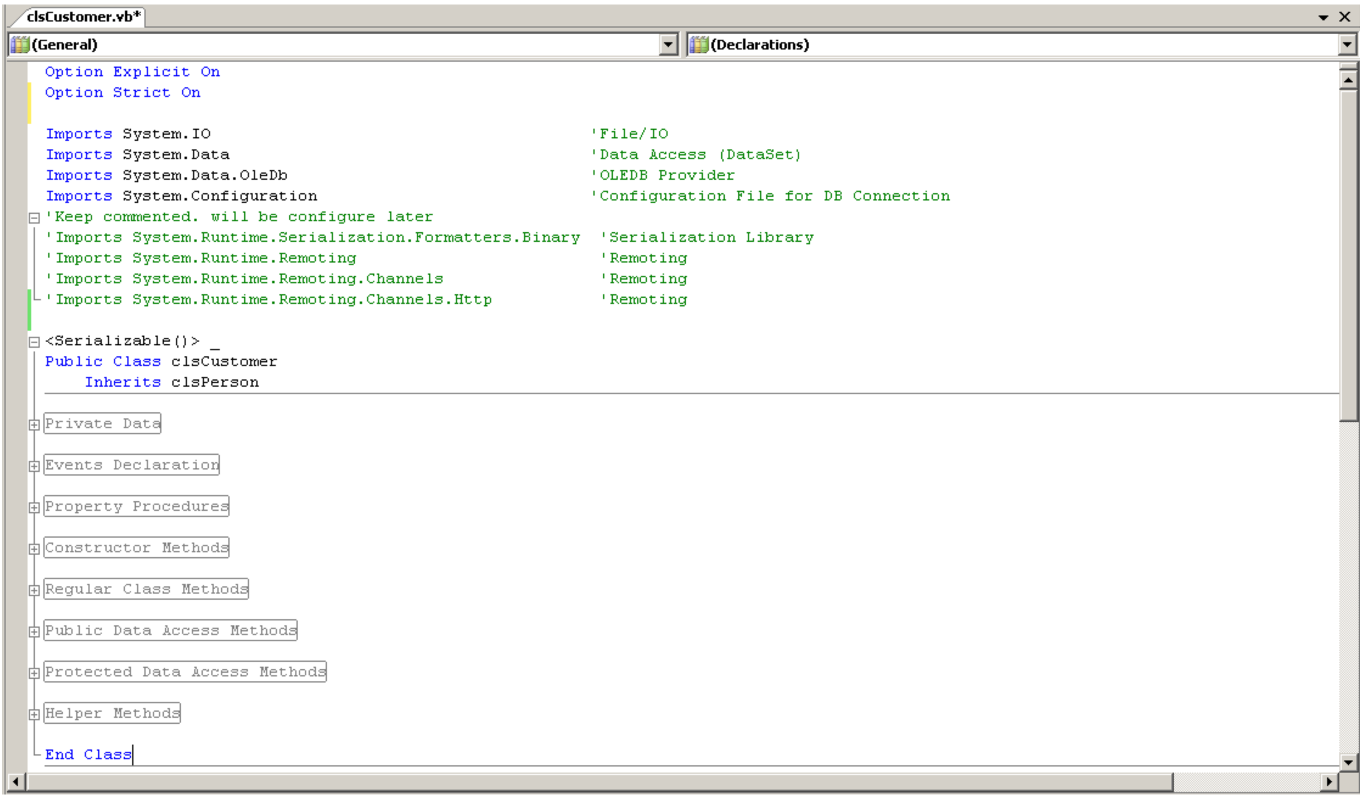
Step 2: COPY/PASTE from BusinessClass Template the DATA ACCESS METHODS to the clsCustomer Class:

1. Open the **BusinessClass** Template and COPY the **Public & Protected Data Access** Methods REGION:



```
Option Explicit On
Option Strict On
Imports System.IO                                   'File/IO
Imports System.Data                                 'Data Access (DataSet)
Imports System.Data.OleDb                           'OLEDB Provider
Imports System.Configuration                        'Configuration File for DB Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                         'Remoting
'Imports System.Runtime.Remoting.Channels                'Remoting
'Imports System.Runtime.Remoting.Channels.Http           'Remoting
<Serializable()> _
Public Class BusinessClassTemplate
    Inherits BusinessBase  'Inherits from BusinessBase.  Must implement MustInherits methods
Private Data

Events Declarations

Property Procedures

Constructor Methods

Business & Regular Methods

Public Data Access Methods

Protected Data Access Methods

Helper Methods
End Class
```

2. PASTE into the ***clsCustomer*** class the DATA ACCESS CODE REGIONS, the class should look as follows when completed:



```
Option Explicit On
Option Strict On

Imports System.IO                                   'File/IO
Imports System.Data                                 'Data Access (DataSet)
Imports System.Data.OleDb                           'OLEDB Provider
Imports System.Configuration                        'Configuration File for DB Connection
'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                         'Remoting
'Imports System.Runtime.Remoting.Channels                'Remoting
'Imports System.Runtime.Remoting.Channels.Http           'Remoting

<Serializable()> _
Public Class clsCustomer
    Inherits clsPerson

Private Data

Events Declaration

Property Procedures

Constructor Methods

Regular Class Methods

Public Data Access Methods

Protected Data Access Methods

Helper Methods

End Class
```

Step 3: NO CHANGES REQUIRED IN Private data & Event Regions:

- The private data & Event declarations stay the same as before

```
#Region "Private Data"
    '*********************************************************************
    'Class Data or Variable declarations
    Private m_CustomerID As String
    Private m_TotalItemsPurchased As Integer

#End Region

#Region "Events Declaration"
    '*********************************************************************
    'Event Declarations
    Public Event OnShopping(ByVal intTotalItems As Integer)

#End Region
```

Step 4: Add MANDATORY DIRTY Object and [OPTIONAL] Validation Rules to the Properties:

- ADD THE MANDATORY DIRTY OBJECT STATEMENT
- ADD ANY REQUIRED VALIDATION CODE

```
#Region "Property Procedures"
    '*********************************************************************
    'Enforcing NO-BLANK, WRITE-ONCE, EXACT-LENGTH & MARK DIRTY for Address
    Public Property CustomerID() As String
        Get
            Return m_CustomerID
        End Get
        Set(ByVal Value As String)

            'NO-BLANK validation
            If Len(Trim(Value)) = 0 Then
                Throw New NotSupportedException("Business Rule: ID cannot be blank")
            End If

            'WRITE-ONCE validation
            If Not Me.IsNew Then
             Throw New NotSupportedException("Business Rule: ID is Write-once only")
            End If

            'EXACT-LENTH validation
            If (Len(Trim(Value)) <> 3) Then
                Throw New NotSupportedException("ID Value not exact Lenght")
            End If

            m_CustomerID = Value

            MyBase.MarkDirty() 'Mark Ojbect as dirty it has been modified
        End Set
    End Property
```

- Continue with PROPERTIES modifications.

```
  '*********************************************************************
   Public Property TotalItemsPurchased() As Integer
       Get
           Return m_TotalItemsPurchased
       End Get
       Set(ByVal Value As Integer)
           m_TotalItemsPurchased = Value

           MyBase.MarkDirty() 'Mark Ojbect as dirty it has been modified
       End Set
   End Property

#End Region
```

Step 5: NO CHANGES IN CONSTRUCTORS:

- No change required to constructor methods.
-

```
#Region "Constructor Methods"
    '*********************************************************************
    'Default Constructor
    Public Sub New()
        'Call Base Class Constructor
        MyBase.New()

        'data member is initialized
        m_CustomerID = ""
    End Sub

    'Parameterized Constructor
    Public Sub New(ByVal strNane As String, ByVal strSSNum As String, _
    ByVal bBDate As Date, ByVal strAddress As String, _
    ByVal strPhone As String, ByVal strCustomerID As String)

        'Call Base Class Paremeterized Constructor
        MyBase.New(strNane, strSSNum, bBDate, strAddress, strPhone)

        'Property Member Initialize data
        Me.CustomerID = strCustomerID
    End Sub

#End Region
```

Step 6: Regular Methods: Print() method stay the same, SHOP() method needs to be MARKED DIRTY

- Regular methods require Business Rules only when you are modifying or making the object dirty, in this case the SHOP() METHOD REQUIRES.

```
#Region "Regular Class Methods"
    '*********************************************************************
    'Regular Class Methods

    'This implementation does not call the base class Print to do the work
    'but instead calls each property individually. This is done because if
    'we call the base class Print() first, then we require two output in the
    'file which contain the record for each object. We only want one print
    'file with all the customer data in one line.
    Public Overrides Sub Print()
        'Create StreamWriter Object for append to file listed
        Dim objPrinter As New StreamWriter("CustomerPrinter.txt", True)

        'Call StreamWriter Object WriteLine method to write the string to file
        objPrinter.WriteLine(MyBase.Name & "," & MyBase.SocialSecurity & "," & _
        MyBase.BirthDate & "," & MyBase.Address & "," & _
        MyBase.Phone & "," & Me.CustomerID & "," & Me.TotalItemsPurchased)

        'Close StreamWriter Object
        objPrinter.Close()

    End Sub

    '*********************************************************************
    ''' <summary>
    ''' Shops by addign items to be purchased to running total items.
    ''' Triggers On Shopping Event & MARK DIRY since we are modifying
    ''' </summary>
    ''' <param name="intItems"></param>
    ''' <remarks></remarks>
    Public Sub Shop(ByVal intItems As Integer)
        m_TotalItemsPurchased = m_TotalItemsPurchased + intItems

        MyBase.MarkDirty() 'Mark Ojbect as dirty it has been modified

        'Raise or trigger event & send information with the event
        RaiseEvent OnShopping(m_TotalItemsPurchased)

    End Sub

#End Region
```

Step 7: VIEW Public Data Access Method from BusinessClass Template

- NO MODIFICATION NEEDED to the Public Shared Data Access Methods we copied from the ***BusinessClass*** template and FORCED upon us by ***BusinessBase***.

```
#Region "Public Data Access Methods"
    'Public interface to Create objects from database
    '*********************************************************************
    ''' <summary>
    ''' [OPTIONAL] Method to create object if default values
    ''' from database are required
    ''' </summary>
    ''' <remarks></remarks>
    Public Overrides Sub Create()
        DataPortal_Create()
    End Sub
    '*********************************************************************
    ''' <summary>
    ''' Method to LOAD() OBJECT from DATABASE
    ''' </summary>
    ''' <param name="Key"></param>
    ''' <remarks></remarks>
    Public Overrides Sub Load(ByVal Key As Object)
        'Calls Local DatPortal_Fetch(Key) To do the work
        DataPortal_Fetch(Key)

    End Sub
    '*********************************************************************
    ''' <summary>
    ''' Method to SAVE() OBJECT to DATABASE. Decision to insert or update
    ''' is done via DIRTY and NEW Mechanism
    ''' </summary>
    ''' <remarks></remarks>
    Public Overrides Sub Save()
        'Only save if dirty, otherwise do nothing in this method
        If Me.IsDirty Then
            If Me.IsNew Then
                'We are new and being inserted
                'Calls Local DataPortal_Insert()
                DataPortal_Insert()
            Else
                'We are OLD so we are being updated
                'Calls Local DataPortal_Update()
                DataPortal_Update()
            End If
        End If

    End Sub
    '*********************************************************************
    ''' <summary>
    ''' Method to delete an object record's from database via ID or key
    ''' </summary>
    ''' <param name="Key"></param>
    ''' <remarks></remarks>
    Public Overrides Sub DeleteObject(ByVal Key As Object)
        'Calls Local DatPortal_DeleteObject() To do the work
        DataPortal_DeleteObject(Key)
    End Sub
#End Region
```

Step 8: VIEW Protected Data Access Methods from Business Class Template

- **No Modification** is needed in the protected from the ***BusinessClass*** template
- Implementation of these methods will take place when we learn ADO.NET

```
#Region "Protected Data Access Methods"
    '*********************************************************************
    'Protected Data Access Methods declarations

    'Data Access Code for Creating a New Business Object
    Protected Overrides Sub DataPortal_Create()
        'Create object and assign default values from database etc.

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'At the end, set New flag to True a new object is created
        MyBase.MarkNew()
    End Sub

    'Data Access Code to fetch an object from Database
    Protected Overrides Sub DataPortal_Fetch(ByVal Key As Object)
        'ADO.NET Queries for Fetching (Select/From/Where) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'At the end, set New flag to False.  NOT Dirty since found in database
        MyBase.MarkOld()
    End Sub

    'Data Access Code to Update an Objects data to database
    Protected Overrides Sub DataPortal_Update()
        'ADO.NET Queries for updating (Update/Set/Where) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'Set New flag to False since exist in database/and is Not dirty any longer
        MyBase.MarkOld()
    End Sub

    'Data Access Code to insert a new object to database
    Protected Overrides Sub DataPortal_Insert()
        'ADO.NET Queries for Inserting (Insert/Into) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'Set New flag to False since exist in database/and is Not dirty any longer
        MyBase.MarkOld()
    End Sub

    'Data Access Code to immediatly delete an object from database.
    Protected Overrides Sub DataPortal_DeleteObject(ByVal Key As Object)
        'ADO.NET Queries for deleting (Delete/From/Where) or Stored Procedures

        'ADD DATA ACCESS CODE HERE USING ADO.NET

        'Object no longer in database, therefore reset our status to be a new object
        MyBase.MarkNew()
    End Sub

#End Region
```

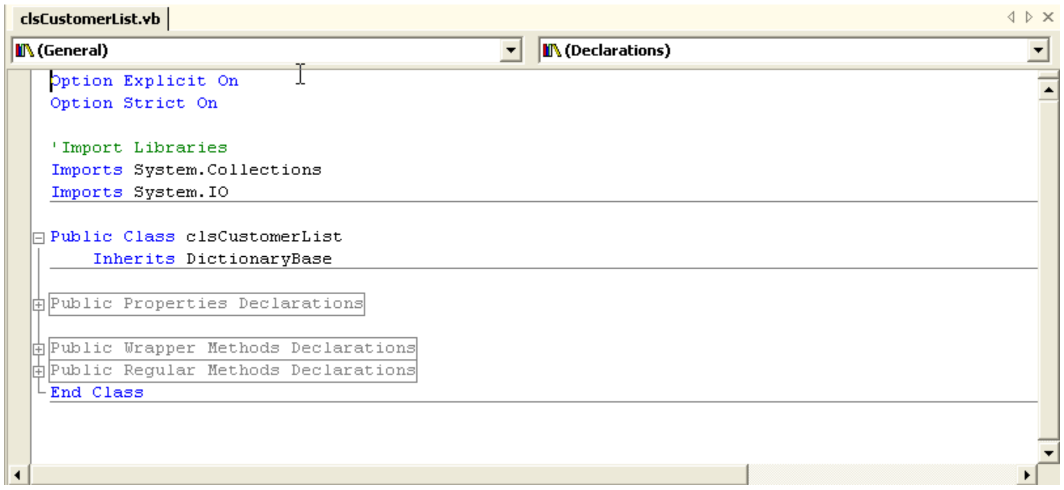
Step 9: VIEW Helper Methods:

- Currently there are no non-business related methods in this class.

```
#Region "Helper Methods"
    '*******************************************************************
    'Methods used to assist other methods or maintenance


#End Region
```

Step 5: The clsCustomerList Collection Class

- Now we turn our attention to the Collection Classes. We need to implement the rules and logic from the ***BusinessCollectionBase*** and the ***BusinessCollectionClass*** template.
- In addition, we need to add *File Access Code* to **load** and **save** the Business Objects in the collection temporarily to a file.
- We will implement these File Access code in the Protected Data Access **Methods DataPortal_Fetch** & **DataPortal_Save()**.
- The current structure of the ***clsCustomeListManager*** class currently looks as follows:



```
Option Explicit On
Option Strict On

'Import Libraries
Imports System.Collections
Imports System.IO

Public Class clsCustomerList
    Inherits DictionaryBase

Public Properties Declarations

Public Wrapper Methods Declarations
Public Regular Methods Declarations
End Class
```

Step 1: COPY/PASTE import & Serializable statements and Data Access Methods from BusinessCollectionClass Template AND PASTE TO clsCustomerList Class:

```
Option Explicit On
Option Strict On

Imports System.IO                                            'File/IO
Imports System.Data                                          'Data Access (DataSet)
Imports System.Data.OleDb                                    'OLEDB Provider
Imports System.Configuration                                 'Configuration File for DB
Connection

'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary      'Serialization Library
'Imports System.Runtime.Remoting                             'Remoting
'Imports System.Runtime.Remoting.Channels                    'Remoting
'Imports System.Runtime.Remoting.Channels.Http               'Remoting
<Serializable()> _
Public Class clsCustomerList
```

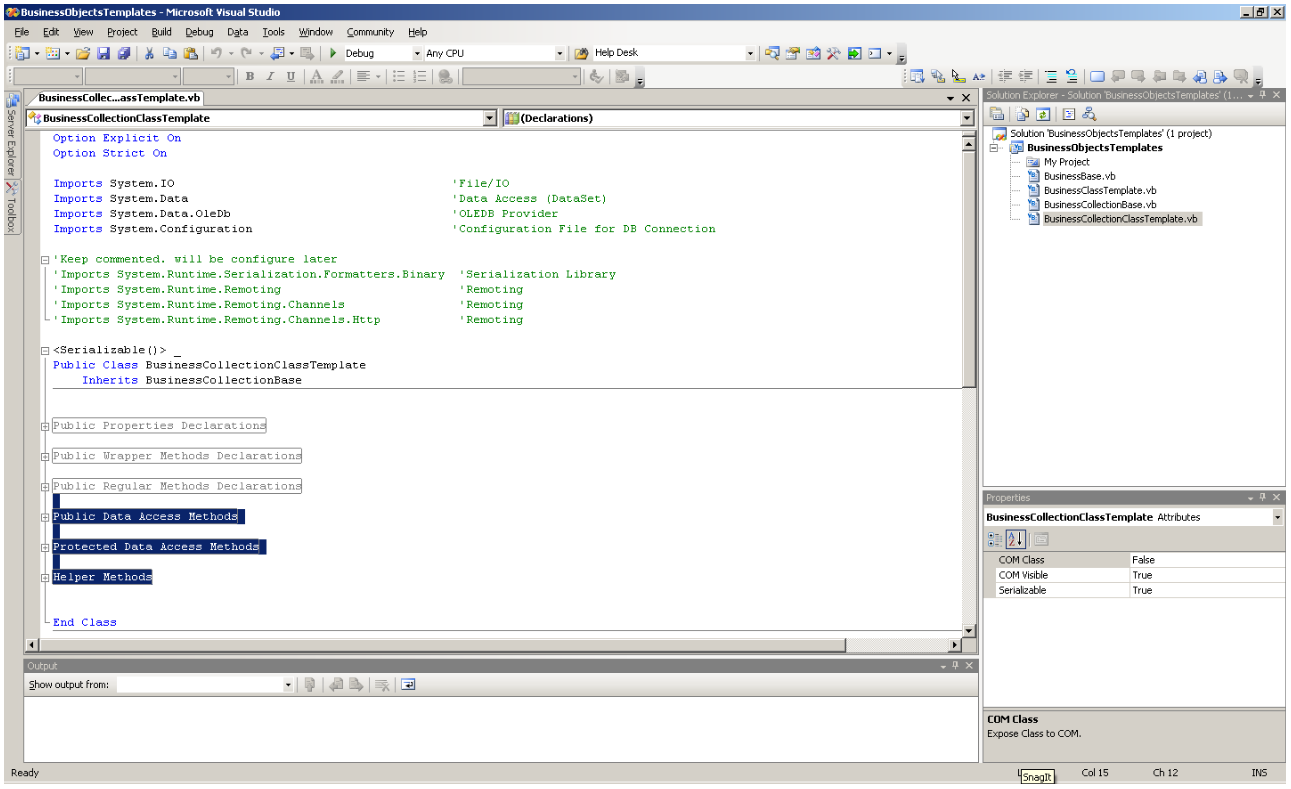
Step 2: COPY/PASTE the INHERIT from BusinessCollectionClass Template:

1. Open the **BusinessCollectionClass** Template and copy the INHERIT BUSINESSCOLLECTIONBASE statement
2. PASTE into the clsCustomer Class UNDER THE CLASS DECLARATION
3. The declaration looks as follows:

```
Option Explicit On
Option Strict On

Imports System.IO                                         'File/IO
Imports System.Data                                       'Data Access (DataSet)
Imports System.Data.OleDb                                 'OLEDB Provider
Imports System.Configuration                              'Configuration File for DB
Connection

'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                         'Remoting
'Imports System.Runtime.Remoting.Channels                'Remoting
'Imports System.Runtime.Remoting.Channels.Http           'Remoting
<Serializable()> _
Public Class clsCustomerList
    Inherits BusinessCollectionBase
```

Step 3: COPY/PASTE Data Access Methods from BusinessCollectionClass Template:

1. Open the **BusinessCollectionClass** Template from TEMPLATE DLL PROJECT and COPY all **Public** & **Protected Data Access Methods REGIONS**:

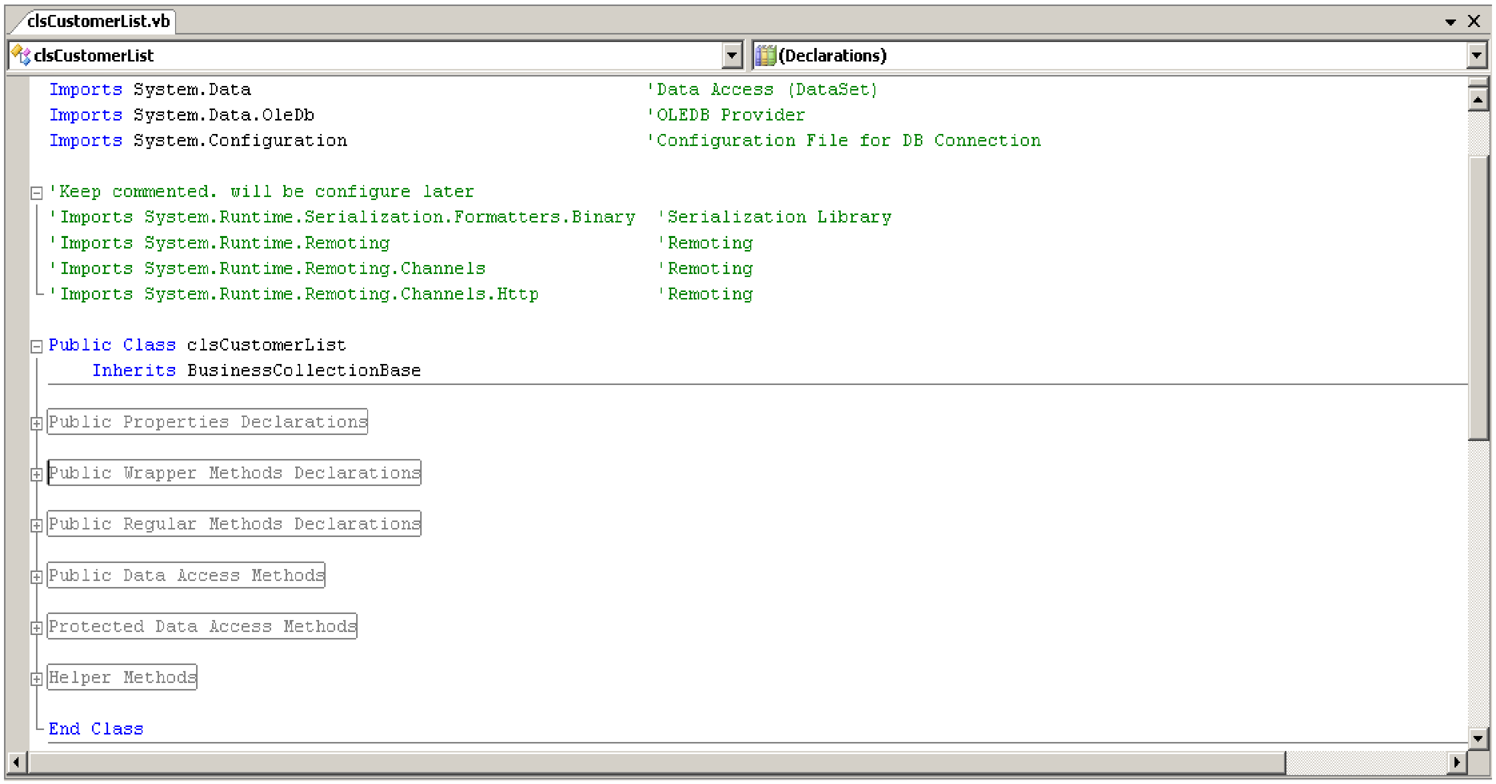

BusinessObjectsTemplates - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Debug
Any CPU
Help Desk
BusinessCollec...assTemplate.vb
BusinessCollectionClassTemplate
(Declarations)
Server Explorer
Toolbox
Solution Explorer - Solution 'BusinessObjectsTemplates' (1...
Solution 'BusinessObjectsTemplates' (1 project)
BusinessObjectsTemplates
My Project
BusinessBase.vb
BusinessClassTemplate.vb
BusinessCollectionBase.vb
BusinessCollectionClassTemplate.vb
Properties
BusinessCollectionClassTemplate Attributes
Output
Show output from:
Ready
SnagIt
Col 15
Ch 12
INS

```
Option Explicit On
Option Strict On

Imports System.IO                                         'File/IO
Imports System.Data                                       'Data Access (DataSet)
Imports System.Data.OleDb                                 'OLEDB Provider
Imports System.Configuration                              'Configuration File for DB Connection

'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary  'Serialization Library
'Imports System.Runtime.Remoting                         'Remoting
'Imports System.Runtime.Remoting.Channels                'Remoting
'Imports System.Runtime.Remoting.Channels.Http           'Remoting

<Serializable()> _
Public Class BusinessCollectionClassTemplate
    Inherits BusinessCollectionBase


Public Properties Declarations

Public Wrapper Methods Declarations

Public Regular Methods Declarations

Public Data Access Methods

Protected Data Access Methods

Helper Methods


End Class
```

COM Class	False
COM Visible	True
Serializable	True

COM Class
Expose Class to COM.

2. At the end of this step, the structure of the ***clsCustomerList*** class should look as follows when completed



```
Imports System.Data                                        'Data Access (DataSet)
Imports System.Data.OleDb                                  'OLEDB Provider
Imports System.Configuration                               'Configuration File for DB Connection

'Keep commented. will be configure later
'Imports System.Runtime.Serialization.Formatters.Binary   'Serialization Library
'Imports System.Runtime.Remoting                           'Remoting
'Imports System.Runtime.Remoting.Channels                  'Remoting
'Imports System.Runtime.Remoting.Channels.Http             'Remoting

Public Class clsCustomerList
    Inherits BusinessCollectionBase

Public Properties Declarations

Public Wrapper Methods Declarations

Public Regular Methods Declarations

Public Data Access Methods

Protected Data Access Methods

Helper Methods

End Class
```

Properties

- NO CHANGES REQUIRED.

Step 4: Property Declaration stays the same:

```
#Region "Public Properties Declarations"
  '*********************************************************************************
    ''' <summary>
    ''' Name: Count() Property
    ''' Purpose: Return number of objects in collection
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shadows ReadOnly Property Count() As Integer
        Get
            Return MyBase.Dictionary.Count
        End Get
    End Property

'*********************************************************************************
    ''' <summary>
    ''' Name: Item(Key) Property
    ''' Purpose: GET or SET the object at the specified key in the Collection
    ''' </summary>
    ''' <param name="key"></param>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property Item(ByVal key As Object) As clsCustomer
        Get
            'Step 1- Return POINTER of Object of associated key
            'Convert returned POINTER
            Return CType(MyBase.Dictionary.Item(key), clsCustomer)

        End Get
        Set(ByVal value As clsCustomer)

            'Step 1-Verify if key exists
            If MyBase.Dictionary.Contains(key) Then
                'Step 2-Set or overwrite object in collection
                MyBase.Dictionary.Item(key) = value
            Else
                'Step 3-Else throws an Argument Exeption to indicate not found.
                Throw New System.ArgumentException("ID Not found")
            End If
        End Set
    End Property

#End Region
```

Wrapper Methods

- ❑ Only wrapper methods that create and modify **Business Object** need to trap for ***NotSupportedException***.

Step 5: ADD Wrapper Method

- In this case the ADD WRAPPER METHOD needs NO MODIFICATION SINCE NO BUSINESS OBJECTS ARE CREATED OR MANIPULATED

```
#Region "Public Wrapper Methods Declarations"

'*********************************************************************************
    ''' <summary>
    ''' Name: Add(Key, Object)Method
    ''' Purpose: Adds new object to the Collection.
    ''' Includes support for duplicate key
    ''' </summary>
    ''' <param name="key"></param>
    ''' <param name="objCustomer"></param>
    ''' <remarks></remarks>
    Public Sub Add(ByVal key As Object, ByVal objCustomer As clsCustomer)
        'Step A- Begin Error trapping
        Try
            'Step 1-Calls Collection.Add(Key,Object) Method to Add object
            MyBase.Dictionary.Add(key, objCustomer)

            'Step B-Traps argumentNullException when key is Nothing or null
        Catch objX As ArgumentNullException
            'Step C-ReThrow ArgumentNullException
        Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
            'Step D-Traps for ArgumentExecption when KEY is duplicate.
        Catch objY As ArgumentException
            'Step E-ReThrow an ArgumentExecption to calling programs
         Throw New System.ArgumentException("Duplicate Key Error: " & objY.Message)
            'Step F-Traps for general exceptions.
        Catch objE As Exception
            'Step G-ReThrow an general exceptions
            Throw New System.Exception("Add Method Error: " & objE.Message)
        End Try
    End Sub
```

Step 6: OVERLOADED ADD Wrapper Method

- The OVERLOADED ADD WRAPPER METHOD CREATES & MANIPULATES a BUSINESS OBJECT, therefore it requires the ***NotSupportedException*** EXCEPTION to be added to the TRY/CATCH

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Overloaded Add(value1, value2..)Method
    ''' Purpose: Add object to collection by passing individual values
    ''' instead of an object. Object is created and populated with parameter values
    ''' Ideal for passing values directly from a user interface textbox control.
    ''' </summary>
    ''' <param name="strCustomerID"></param>
    ''' <param name="strName"></param>
    ''' <param name="strSSNum"></param>
    ''' <param name="dBDate"></param>
    ''' <param name="strAddress"></param>
    ''' <param name="strPhone"></param>
    ''' <remarks></remarks>
    Public Sub Add(ByVal strCustomerID As String, ByVal strName As String, _
    ByVal strSSNum As String, ByVal dBDate As Date, ByVal strAddress As String, _
    ByVal strPhone As String)
        'Step A- Begin Error trapping
        Try
            'Step 1-Creates NEW Temp Object
            Dim objItem As New clsCustomer

            'Step 2-Populates object it with data passed as argument
            With objItem
                .Name = strName
                .SocialSecurity = strSSNum
                .BirthDate = dBDate
                .Address = strAddress
                .Phone = strPhone
                .CustomerID = strCustomerID
            End With

        'Step 3-Use Collection.Add(Key, Object)to add object. Object ID used as Key
            MyBase.Dictionary.Add(objItem.CustomerID, objItem)

            'Step B-Traps for Business Rule violations since object is modified
        Catch objNSE As NotSupportedException
            Throw New System.NotSupportedException(objNSE.Message)
            'Step C-Traps argumentNullException when key is Nothing or null
        Catch objX As ArgumentNullException
            'Step D-ReThrow ArgumentNullException
        Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
            'Step E-Traps for ArgumentExecption when KEY is duplicate.
        Catch objY As ArgumentException
            'Step F-ReThrow an ArgumentExecption to calling programs
         Throw New System.ArgumentException("Duplicate Key Error: " & objY.Message)
            'Step G-Traps for general exceptions.
        Catch objE As Exception
            'Step H-ReThrow an general exceptions
            Throw New System.Exception("Add Method Error: " & objE.Message)
        End Try
    End Sub
```

Step 7: REMOVE Wrapper Method

- The Remove Wrapper method requires NO modification since there are no Business Objects Created or Modified.

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Function Remove(Key)Sub Method
    ''' Purpose: Remove object from collection based on key.
    ''' </summary>
    ''' <param name="key"></param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Function Remove(ByVal key As Object) As Boolean
        'Step A- Begin Error trapping
        Try

            'Step 1-Verify object exists
            If MyBase.Dictionary.Contains(key) Then
                'Step 2-Calls CollectionObject.Remove(Key) Method
                MyBase.Dictionary.Remove(key)
                'Step 3-Return True since found and removed
                Return True
            Else
                'Step 4-Return False since not found
                Return False
            End If

            'Step B-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step C-Throw Collection ArgumentNullException
        Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
            'Step D-Traps for general exceptions.
        Catch objE As Exception
            'Step E-Throw an general exceptions
            Throw New System.Exception("Remove Error: " & objE.Message)
        End Try
    End Function
```

Regular Methods

- Again, only regular methods that create and modify **Business Object** need to trap for ***NotSupportedException***.

Step 6: EDIT Methods

- The regular *EditItem* method performs on manipulation of Business Objects therefore work is needed here.

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Function Edit(Key, object)Method
    ''' Purpose: Replaces object located at specified key in the Collection
    ''' </summary>
    ''' <param name="key"></param>
    ''' <param name="objItem"></param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Function Edit(ByVal key As Object, ByVal objItem As clsCustomer) As
Boolean
        'Step A- Begin Error trapping
        Try
            'Step 1-Verify object exist
            If MyBase.Dictionary.Contains(key) Then
                'Step 2-Sets CollectionObject.Item(Key) = object
                MyBase.Dictionary.Item(key) = objItem
                'Step 3-Return confirmation
                Return True
            Else
                'Step 4-Return object not found
                Return False
            End If

            'Step B-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step C-Throw Collection ArgumentNullException
        Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
            'Step D-Traps for general exceptions.
        Catch objE As Exception
            'Step E-Throw an general exceptions
            Throw New System.Exception("EditItem Error: " & objE.Message)
        End Try
    End Function
```

Step 9: OVERLOADED EDIT Methods

- The Overloaded EditItems(x, y, z…) CREATES & MODIFIES a Business Object, trapping for ***NotSupportedException*** required.
- **IMPORTANT!** NOTE THAT ID NUMBER & SOCIAL SECURITY ARE NOT BEING EDITED! THEY ARE WRITE-ONCE PROPERTY AND CANNOT BE MODIFIED DURING AND UPDATE OR WHEN OBJECT IS OLD!

```
'*********************************************************************************
   ''' <summary>
   ''' Name: Function OVERLOADED Edit(value1, value2,etc.)
   ''' Purpose: Sets or MODIFIES object located at specified key in the Collection
   ''' </summary>
   Public Function Edit(ByVal strCustomerID As String, ByVal strName As String, _
   ByVal strSSNum As String, ByVal dBDate As Date, ByVal strAddress As String, _
   ByVal strPhone As String) As Boolean
       'Step A- Begin Error trapping
       Try
           'Step 1-Create temporary POINTER
           Dim objItem As clsCustomer

     'Step 2-Get a Reference of pointer to the actual object inside the collection.
     'Use CType() function to convert object retrieved from list to clsCustomer
           objItem = CType(MyBase.Dictionary.Item(strCustomerID), clsCustomer)

           'Step 3-Verify object exists
           If objItem Is Nothing Then
               'Step 4-Return False since not found
               Return False
           Else
      'Step 5-Sets individual properties of actual object inside the collection.
      'ANY PROPERTY THAT IS WRITE-ONCE CANNOT BE MODIFIED.
      'NOTE THAT THE ID NUMBER & SOCIAL SECURITY ARE NOT PART OF THE PROPERTY SET
      'CODE BECAUSE THEY ARE BOTH WRITE-ONCE PROPERTY AND CANNOT BE MODIFIED
      'WHEN AN OBJECT IS NOT NEW (OLD)/LOADED FROM DATABASE AND MARKED FOR UPDATE!
               With objItem
                   .Name = strName
                   .BirthDate = dBDate
                   .Address = strAddress
                   .Phone = strPhone
               End With

               'Step 6-Return True since found and modified
               Return True
           End If

           'Step B-Traps for Business Rule violations since object is modified
       Catch objNSE As NotSupportedException
           Throw New System.NotSupportedException(objNSE.Message)
           'Step C-Traps for ArgumentNullException when key is Nothing or null.
       Catch objX As ArgumentNullException
           'Step D-Throw Collection ArgumentNullException
       Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
           'Step E-Traps for general exceptions.
       Catch objE As Exception
           'Step F-Throw an general exceptions
           Throw New System.Exception("EditItem Error: " & objE.Message)
       End Try
   End Function
```

Step 10: PRINT Methods

- No modification is required for the Print and PrintAll methods since no Business Objects are being modified

```
'*****************************************************************************
    ''' <summary>
    ''' Name: Print(Key)Sub Method
    ''' Purpose: Prints object from collection to Printer File
    ''' </summary>
    ''' <param name="key"></param>
    ''' <returns></returns>
    ''' <remarks></remarks>
Public Function Print(ByVal key As Object) As Boolean
      'Step A- Begin Error trapping
      Try

            'Step 1-Step 1-Create Temporary object POINTER
            Dim objItem As clsCustomer

      'Step 2-Get a Reference of pointer to the actual object inside the collection
      'Use CType() function to convert object retrieved from list to clsCustomer
            objItem = CType(MyBase.Dictionary.Item(key), clsCustomer)

            'Step 3-Verify object exists
            If objItem Is Nothing Then
                'Step 4-Return False since not found
                Return False
            Else
                'Step 5-Calls Temp Object.Print Method to print the object to file
                objItem.Print()

                'Step 6-Return True since found
                Return True
            End If

            'Step B-Traps for Business Rule violations since object is modified
        Catch objNSE As NotSupportedException
            Throw New System.NotSupportedException(objNSE.Message)
            'Step C-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step D-Throw Collection ArgumentNullException
        Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
            'Step E-Traps for general exceptions.
        Catch objE As Exception
            'Step F-Throw an general exceptions
            Throw New System.Exception("PrintCustomer Error: " & objE.Message)
      End Try

End Function
```

Step 11: PRINTALL Methods

- No modification is required for the Print and PrintAll methods since no Business Objects are being modified

```
'*************************************************************************************
    ''' <summary>
    ''' Name: PrintAllCustomers()Sub Method
    ''' Purpose: Use For..Each loop to Prints all objects in collection to File
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub PrintAll()
        'Step A- Begin Error trapping
        Try

            'Step 1-Step 1-Create Temporary customer and Dictionary object POINTERS
            Dim objDictionaryEntry As DictionaryEntry
            Dim objItem As clsCustomer

            'Step 2-Use For..Each loop to iterate through Dictionary
            'Pointer points to each object during every iteration.
            For Each objDictionaryEntry In MyBase.Dictionary
                'Step 3-Convert DictionaryEntry pointer returned to Type Person
                objItem = CType(objDictionaryEntry.Value, clsCustomer)

                'Step 4-Calls Temp Object.Print Method to print the object to file
                objItem.Print()
            Next

            'Step B-Traps for general exceptions.
        Catch objE As Exception
            'Step C-Throw an general exceptions
            Throw New System.Exception("PrintAll Method Error: " & objE.Message)
        End Try
    End Sub

#End Region
```

Public Data Access Methods

- Now we need to look at the Public Data Access Methods we copied from the ***BusinessCollectionClass*** template.
- NO MODIFICATION IS REQUIRED, SINCE THESE METHODS SIMPLY CALL THE PROTECTED DATA ACCESS METHOD TO DO THE WORK.

Step 12: Public Shared Data Access Method

- NO MODIFICATION REQUIRED.

```
#Region "Public Data Access Methods"
    '*********************************************************************
    ''' <summary>
    ''' [Optional] Calls Data Portal_Create to create a Collection Object. This
    ''' Method is not used in this class, but can be used in the
    ''' future to create objects that need data from database upon Creation
    ''' </summary>
    ''' <remarks></remarks>
    Public Overrides Sub Create()
        'Calls Local DatPortal_Create() To do the work
        DataPortal_Create()

    End Sub
    '**************************************************************************
    ''' <summary>
    ''' Calls Data_Portal_Fetch to load all objects from database
    ''' </summary>
    ''' <remarks></remarks>
    Public Overrides Sub Load()
        'Calls Local DatPortal_Fetch() To do the work
        DataPortal_Fetch()

    End Sub
    '**************************************************************************
    ''' <summary>
    ''' Calls DataPortal_Save() to save all objects in collection to Database
    ''' </summary>
    ''' <remarks></remarks>
    Public Overrides Sub Save()
        'Verify there are dirty objects in Collection
        'Only modify if dirty, otherwise do nothing in this method
        If IsDirty Then
            'Dirty Collection, go ahead and update
            DataPortal_Save()
        End If

    End Sub
    '**************************************************************************
    ''' <summary>
    ''' Calls DataPortal_DeleteObject to delete an object residing
    '''  In the collection from the database
    ''' </summary>
    ''' <param name="Key"></param>
    ''' <remarks></remarks>
    Public Overrides Sub DeleteObject(ByVal Key As Object)
        'Calls Local DatPortal_DeleteObject() To do the work
        DataPortal_DeleteObject(Key)
    End Sub

#End Region
```

Protected Data Access Methods

- Now we need to modify the PROTECTED SHARED Data Access Methods we copied from the ***BusinessCollectionClass*** template.
- THESE ARE THE METHODS THAT PERFORM THE ACTUAL DATA ACCESS, THERE ARE TWO TYPES OF MODIFICATIONS REQUIRED FOR THE PROTECTED DATA ACCESS METHOD:

  1. The modification is simply to replace the ***BusinessCollectionClass*** statements in the code with ***clsCustomerList***
  2. ADD the DATA ACCESS CODE USING ADO.NET. WE WILL NOT DO THIS STEP IN THIS EXAMPLE.
  3. **TEMPORARY!!!!** CUT/PASTE THE FILE ACCESS CODE from the PREVIOUS LOAD() & SAVE() Method TO MAKE THIS PROJECT WORK USING THE FILE ACCESS. THIS IS ONLY TEMPORARY SINCE THE NEXT STEP IS TO PUT REAL ADO.NET DATA ACCESS CODE

Step 13: CREATE PROTECTED DataPortal_Create Data Access Method

- THIS IS AN OPTIONAL METHOD. Only required when we need CREATE A COLLECTION that requires DEFAULT DATA FROM THE DATABASE.
- NO MODIFICATION REQUIRED AT THIS TIME.

```
#Region "Protected Data Access Methods"
    '*********************************************************************
    'Protected Data Access Methods declarations
    '**************************************************************************
    ''' <summary>
    ''' Data Access or other Code for Creating a New Business COLLECTION Object
    ''' Used when object requires data from db upon creation
    ''' </summary>
    ''' <remarks></remarks>
    Protected Overrides Sub DataPortal_Create()
        'Create object and assign default values from database etc.
    End Sub
```

Step 14: *****SPECIAL TEMPORARY FILE ACCESS CODE DataPortal_Fetch and File

- In the **DataPortal_Fetch()** method is where we will place our temporary FILE ACCES CODE to the Fetch data from the ***Customer.txt*** file.
- Future implementation will use ADO.NET, but for now we will use a file.

```
'***************************************************************************
Protected Overrides Sub DataPortal_Fetch()
    '********TEMPORARY FILE ACCESS CODE FOR LODADING DATA*****************
    'Step A- Begin Error trapping
    Try
        'Step 1-Declare Customer POINTER
        Dim objCustomer As clsCustomer

        'Step 2-Use File class Shared method to test if File exists
        If Not File.Exists("CustomerData.txt") Then
            'Create the file since it does not exist
            Dim objFile As New StreamWriter("CustomerData.txt")
            'Close the file for writing
            objFile.Close()
        End If

        'Step 3-Open file for reading
        Dim objDataFile As New StreamReader("CustomerData.txt")

        'Step 4-Loop through file
        Do While objDataFile.Peek <> -1

            'Step 5-Read a line from file & assign to variable
            Dim strLine As String = objDataFile.ReadLine

            'Step 6-Parse the line using VB Split() & assign to array
            Dim tempArray() As String = Split(strLine, ",")

            'Step 7-Create NEW Object
            objCustomer = New clsCustomer()

            'Step 7-Populates object it with data from file
            With objCustomer
                .CustomerID = tempArray(0)
                .Name = tempArray(1)
                .SocialSecurity = tempArray(2)
                .BirthDate = CDate(tempArray(3))
                .Address = tempArray(4)
                .Phone = tempArray(5)
                .TotalItemsPurchased = CInt(tempArray(6))
            End With

            'Step 7-Call add to add object to Collection
            Add(objCustomer.CustomerID, objCustomer)
        Loop

        'Step 8-Close File
        objDataFile.Close()

        'Step B-Traps for general exceptions.
    Catch objE As Exception
        'Step C-Throw an general exceptions
        Throw New System.Exception("Load Error: " & objE.Message)
    End Try
    '******** END OF TEMPORARY FILE ACCESS CODE
    'THE CORRECT CODE WILL BE IMPLEMENTED WHEN DURING THE ADO.NET LECTURES
End Sub
```

Step 15: *****SPECIAL TEMPORARY CHANGES to DataPortal_Save()

- Now we need to add File Access code to ***DataPortal_Save()***. Nevertheless, the job of this method is to iterate through the collection and call each CHILD Object's Save() method to do the work. The code to do this is ALREADY IN THE TEMPLATE.
- We need to modify this code AS FOLLOWS:

  1. Replace the CHILD OBJECT ***BusinessClass*** declarations in the code with ***clsCustomer***
  2. LEAVE THE CODE PROVIDED BY THE TEMPLATE ALONE.
  3. ADD THE FILE ACCESS CODE.
  4. IN THE FUTURE - IMPORTANT!!! WHEN WE USE **ADO.NET**, YOU NEED TO UNCOMMENT THE CODE AND REMOVE THE FILE ACCESS CODE.

- Comment existing Business Object code and add File Access code

```
'***************************************************************************
''' <summary>
''' SAVES all objects from database by Iterating through Collection, and
''' calling Each ITEM object SAVE() method so each Item saves itself
''' </summary>
''' <remarks></remarks>
Protected Overrides Sub DataPortal_Save()

    'Iterates through Collection, Calling Each CHILD object.Save() method
    'CHILD Objects save themselves
    'Step A- Begin Error trapping
    Try
        'Step 1-Step 1-Create Temporary Person and Dictionary object POINTERS
        Dim objDictionaryEntry As DictionaryEntry
        Dim objChild As clsCustomer

        'Step 2-Use For..Each loop to iterate through Collection
        For Each objDictionaryEntry In MyBase.Dictionary
            'Step 3-Convert DictionaryEntry pointer returned to Type Person
            objChild = CType(objDictionaryEntry.Value, clsCustomer)

            'Step 4-Call Child to Save itself
            objChild.Save()

        Next
        'Step B-Traps for general exceptions.
    Catch objE As Exception
        'Step C-Throw an general exceptions
        Throw New System.Exception("Save Error! " & objE.Message)
    End Try
```

- Continue DataPortal_Save().
- Add File Access Code

```
        '********TEMPORARY FILE ACCESS CODE FOR LODADING DATA*****************
        'Step A- Begin Error trapping
        Try

         'Step 1-Open file for writing with options to Overwrites the existing file
            Dim objWrite As New StreamWriter("CustomerData.txt", False)

            'Step 2-Create Temporary DictionaryEntry and Customer POINTERS
            Dim objDictionaryEntry As DictionaryEntry
            Dim objItem As clsCustomer

            'Step 3-Use For..Each loop to iterate through SortedList
            'Pointer points to each object during every iteration.
            For Each objDictionaryEntry In MyBase.Dictionary
                'Step 4-Convert DictionaryEntry pointer returned to Type Person
                objItem = CType(objDictionaryEntry.Value, clsCustomer)

                'Step 5-Write Object's content as a COMMA-DELIMITED line to the file
                objWrite.WriteLine(objItem.CustomerID & "," & _
                objItem.Name & "," & _
                objItem.SocialSecurity & "," & _
                objItem.BirthDate & "," & _
                objItem.Address & "," & _
                objItem.Phone & "," & _
                objItem.TotalItemsPurchased)
            Next

            'Step 6-Close file
            objWrite.Close()

            'Step B-Traps for general exceptions.
        Catch objE As Exception
            'Step C-Throw an general exceptions
            Throw New System.Exception("Save Error: " & objE.Message)
        End Try

   '********END OF TEMPORARY FILE ACCESS CODE FOR LODADING DATA*****************

    End Sub
```

Step 16: CHANGES to DataPortal_DeleteObject(Key)

- Now we need to make some small changes to DataPortal_DeleteObject(). We need to the following:

1. Replace the CHILD OBJECT ***BusinessClass*** declarations in the code with ***clsCustomer***
2. Select the Correct Property for the CHILD OBJECT which represents the KEY

```
    '**************************************************************************
    ''' <summary>
    ''' DELETES AN OBJECT BY ID from database by Iterating through Collection
    ''' and calling Each ITEM object DELETE(ID) method so each Item delete itself
    ''' </summary>
    ''' <param name="Key"></param>
    ''' <remarks></remarks>
    Protected Overrides Sub DataPortal_DeleteObject(ByVal Key As Object)
        'Iterates through Collection, Calling Each CHILD object.Delete() method
        'CHILD Objects Delete themselves

        'Step A- Begin Error trapping
        Try
            'Step 1-Step 1-Create Temporary Person and Dictionary object POINTERS
            Dim objDictionaryEntry As DictionaryEntry
            Dim objItem As clsCustomer

            'Step 2-Use For..Each loop to iterate through Collection
            For Each objDictionaryEntry In MyBase.Dictionary
                'Step 3-Convert DictionaryEntry pointer returned to Type Person
                objItem = CType(objDictionaryEntry.Value, clsCustomer)

                'Step 4-Find target object based on key
                'YOU WILL NEED TO SELECT THE CORRECT PROPERTY
                'FOR objItem.Property, ALSO YOU NEED TO CONVERT THE
                'KEY PARAMETER USING CSTR OR CINT ETC. DEPENDING
                'ON THE DATATYPE OF THE objItem.Property
                If objItem.CustomerID = CStr(Key) Then
                    'Step 5-Object deletes itself
                    objItem.DeleteObject(Key)

                    ''Step 6-[OPTIONAL] Remove Object From Collection
                    ''since no longer in DB
                    'MyBase.Dictionary.Remove(Key)
                End If
            Next
            'Step B-Traps for general exceptions.
        Catch objE As Exception
            'Step C-Throw an general exceptions
            Throw New System.Exception("Delete Error! " & objE.Message)
        End Try

    End Sub

#End Region
```

Step 17: Helper Methods:

- Now we need to modify or add any Helper Methods. Currently there are no non-business related methods in this class.

```
#Region "Helper Methods"
    '*********************************************************************
    'Methods used to assist other methods or maintenance


#End Region

End Class
```

Presentation/User-Interface Layer

Module code:

- Now we need to make required changes to the Module. We need to the following:

  1. ADD REFERENCE TO DLL where required
  2. ADD any required Error handling when CREATING AND MODIFYING BUSINESS OBJECTS, by trapping for ***NotSupportedException*** Exception

Step 3: Modify the Code in the Module

- In the Module, we are force to make some changes.

  - REFERENCE DLL WHEN CREATING COLLECTION CLASS OBJECT

Step 1: Sub Main() Stays the same

- No changes needed in Sub Main()

```
Option Explicit On
Option Strict On

Module modMainModule

    'Declare Public Array of Person Objects
    Public objCustomerList As New BusinessObjectsDLL.clsCustomerList


    Dim objMainForm As frnMain = New frnMain

    Public Sub Main()

        'Perfom initialization
        InitializeList()

        'Display Customer Form
        objMainForm.ShowDialog()


    End Sub
```

Step 2: IntializeList Method:

- In this implementation, I will NOT create default OBJECTS HERE OR ANY INITIALIZATION

```
'*************************************************************************************
    ''' <summary>
    ''' Name: InitializeList() Method
    ''' Purpose: Nothing is required for this example
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub InitializeList()
        'No objects are added to Customer Collection from intialize
        'Since we are storing our Customers in a File, we don't really
        'want to add Customer object from here! If we do
        'these objects will be stored in the file via Save() and then
        'we will have duplicate objects during the load(), and since we cannot have
        'two objects with the same key we will raise and Exception.

    End Sub


End Module
```

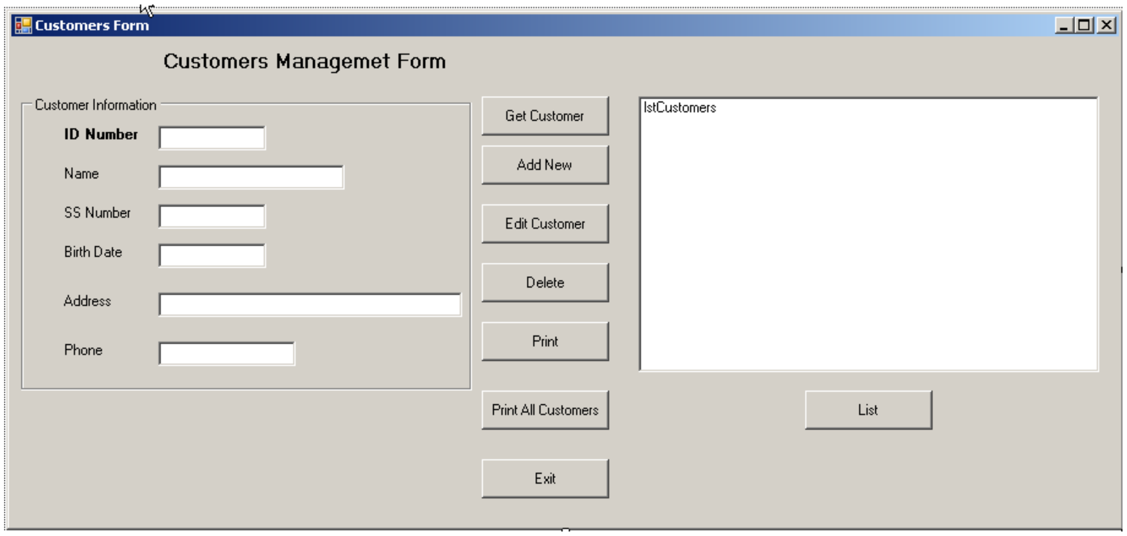
Step 1: Modify the Presentation/User Interface Layer

Customer Management Form

- Now we need to make some small changes to the Customer Management Form. We need to the following:

  1. ADD REFERENCE TO DLL where required
  2. ADD any required Error handling when CREATING AND MODIFYING BUSINESS OBJECTS, by trapping for ***NotSupportedException*** Exception

- The ***Customer Management Form*** looks as follows:



- In addition we will automatically LOAD all customer data from file during *Form_Load* event and SAVE all customer data to file when the *Exit* button is clicked.

Step 1: Modity the Form Level Object to Use the DLL. Also we show the Form Load() event

- Modify OBJECT DECLARATION TO USE CLASS in DLL:

```
Option Explicit On
Option Strict On

Public Class frmCustomerManagement

    'Declare Form Level POINTER
    Private objCustomer As BusinessObjectsDLL.clsCustomer
```

Step 2: The FORM_LOAD() event-handler

- WE NEED TO TRAP FOR ***NotSupportedException*** to support our BUSINESS VALIDATION RULES:

```
'*********************************************************************************
   ''' <summary>
   '''Name: Event-Handler Form_Load
   '''Purpose: Calls Collection.Load() to populate collection with objects from file
Private Sub frmCustomerForm_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        'Step A-Begins Exeception handling.
        Try

            'Step 1-Load objects from file to collection
            objCustomerList.Load()

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step C-Traps for general exceptions.
        Catch objE As Exception
            'Step D-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 3: The FORM_CLOSE() event-handler

- WE NEED TO TRAP FOR ***NotSupportedException*** to support our BUSINESS VALIDATION RULES:

```
'*********************************************************************************
    ''' <summary>
    '''Name: Event-Handler Form_Close()
    '''Purpose:Destroys Form-level object pointer when form closes
    '''Saves Collection objects to file and clears the collection
Private Sub frmCustomerManagement_FormClosed(ByVal sender As Object, ByVal e As
System.Windows.Forms.FormClosedEventArgs) Handles Me.FormClosed
        'Step A-Begins Exeception handling.
        Try
            'Step 1-Destroy Form-Level Objects
            objCustomer = Nothing

            'Step 2-Save objects from Collection to file
            objCustomerList.Save()

            'Step 3-Clear the Collection
            objCustomerList.Clear()

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step C-Traps for general exceptions.
        Catch objE As Exception
            'Step D-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 4: The btnExit_Click() event-handler

- NO MODIFICATION REQUIRED SINCE NO BUSINESS OBJECTS ARE CREATED OR MODIFIED:

```
'*******************************************************************************
    ''' <summary>
    '''Name: Event-Handler for for Exit button
    '''Purpose:Closes the Form
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub btnExit_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnExit.Click

        'Step 1-Close the file
        Me.Close()

    End Sub
```

Step 5: GetCustomer_Click() event-handler – We Catch a NotSupportedException for Our Business Object Validation Rules

- TRAP FOR ***NotSupportedException*** to support our BUSINESS VALIDATION RULES.

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Event-Handler for btnGetCustomer button
    ''' Purpose: To retrieve an POINTER TO object from the collection base on ID
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
Private Sub btnGetCustomer_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnGetCustomer.Click
        'Step A-Begins Exeception handling.
        Try
     'Step 1-Call Calls Item() Property to return pointer to objecT in Collection
            objCustomer = objCustomerList.Item(txtIDNumber.Text.Trim)

     'Step 2-If result of search is Nothing, then display customer is not found
            If objCustomer Is Nothing Then
                MessageBox.Show("Customer Not Found")

                'Step 3-Clear all controls
                txtName.Text = ""
                txtIDNumber.Text = ""
                txtBirthDate.Text = ""
                txtAddress.Text = ""
                txtPhone.Text = ""
            Else

      'Step 4-Then Data is extracted from customer object & placed on textboxes
                With objCustomer
                    txtIDNumber.Text = .CustomerID
                    txtName.Text = .Name
                    txtSSNum.Text = .SocialSecurity
                    txtBirthDate.Text = CStr(.BirthDate)
                    txtAddress.Text = .Address
                    txtPhone.Text = .Phone
                End With
            End If

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step C-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step D-Inform User
            MessageBox.Show(objX.Message)
            'Step E-Traps for general exceptions.
        Catch objE As Exception
            'Step F-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 6: Add_Click() event-handler – Trap for NotSupportedException

- TRAP FOR ***NotSupportedException*** to support our BUSINESS VALIDATION RULES.

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Event-Handler for btnAdd button
    ''' Purpose:To add new object to the collection
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
Private Sub btnAdd_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles btnAdd.Click
        'Step A- Begin Error trapping
        Try

           'Step 1-Calls Collection Add(Value1,Value2,.) pass text control arguments
            objCustomerList.Add(txtIDNumber.Text.Trim, txtName.Text.Trim, _
            txtSSNum.Text.Trim, CDate(txtBirthDate.Text), txtAddress.Text.Trim, _
            txtPhone.Text.Trim)

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step C-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step D-Inform User
            MessageBox.Show(objX.Message)
            'Step E-Traps for ArgumentExecption when KEY is duplicate.
        Catch objY As ArgumentException
            'Step F-Inform User
            MessageBox.Show(objY.Message)
            'Step G-Traps for general exceptions.
        Catch objE As Exception
            'Step H-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 7: EditCustomer_Click() event-handler – Trap for NotSupportedException

- TRAP FOR ***NotSupportedException*** to support our BUSINESS VALIDATION RULES.

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Event-Handler for btnEditCustomer button
    ''' Purpose: Initiate the Edit process to modify an object in the collection
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
Private Sub btnEditCustomer_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnEditCustomer.Click
        'Step A- Begin Error trapping
        Try
            Dim bolResults As Boolean

            'Step 1-Call Module EditItem(index,x,y,z,...) method with textbox data
            bolResults = objCustomerList.Edit(txtIDNumber.Text.Trim, _
            txtName.Text.Trim, txtSSNum.Text.Trim, CDate(txtBirthDate.Text), _
            txtAddress.Text.Trim, txtPhone.Text.Trim)

            'Step 2-If not found display Message & clear all controls
            If bolResults <> True Then
                MessageBox.Show("Customer Not Found")
            End If

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation!  " & objNSE.Message)
            'Step B-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step C-Inform User
            MessageBox.Show(objX.Message)
            'Step D-Traps for general exceptions.
        Catch objE As Exception
            'Step E-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 8: Delete_Click() event-handler – Trap for NotSupportedException

- TRAP FOR ***NotSupportedException*** to support our BUSINESS VALIDATION RULES.

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Event-Handler for btnDelete button
    ''' Purpose: To delete an object from the collection base on ID or Key
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
Private Sub btnDelete_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnDelete.Click
        'Step A- Begin Error trapping
        Try
            Dim bolResults As Boolean

            'Step 1-Calls Remove() method of module. Key is passed as argument
            bolResults = objCustomerList.Remove(txtIDNumber.Text.Trim)

            'Step 2-If not found display Message & clear all controls
            If bolResults <> True Then
                MessageBox.Show("Customer Not Found")
            End If

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation!  " & objNSE.Message)
            'Step C-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step D-Inform User
            MessageBox.Show(objX.Message)
            'Step E-Traps for general exceptions.
        Catch objE As Exception
            'Step F-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 9: Print_Click() event-handler – Trap for NotSupportedException

- Trap for ***NotSupportedException*** exception in case the call to the PrintCustomer method may return business object exceptions.

```
'*************************************************************************************
    ''' <summary>
    ''' Name: Event-Handler for btnPrint button
    ''' Purpose: Prints Object in the list to printer file
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
Private Sub btnPrint_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnPrint.Click
        'Step A- Begin Error trapping
        Try
            Dim bolResults As Boolean

            'Step 1-Calls Remove(Key) method of module
            bolResults = objCustomerList.Print(txtIDNumber.Text.Trim)

            'Step 2-If not found display Message & clear all controls
            If bolResults <> True Then
                MessageBox.Show("Customer Not Found")
            End If

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step C-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step D-Inform User
            MessageBox.Show(objX.Message)
            'Step E-Traps for general exceptions.
        Catch objE As Exception
            'Step F-Inform User
            MessageBox.Show(objE.Message)
        End Try
End Sub
```

Step 10: PrintAll_Click() event-handler– Trap for NotSupportedException

- Trap for ***NotSupportedException*** exception in case the call to the PrintaLLCustomer method may return business object exceptions.

```
'*********************************************************************************
    ''' <summary>
    ''' Name: Event-Handler for btnPrintAllCustomers button
    ''' Purpose: Prints all Objects in the list to file
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub btnPrintAll_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnPrintAll.Click
        'Step A- Begin Error trapping
        Try
            'Step 1-Calls PrintAllCustomers() method of module.
            objCustomerList.PrintAll()

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step D-Traps for general exceptions.
        Catch objE As Exception
            'Step E-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 11: Add code to the btnList_Click() event-handler

- Trap for ***NotSupportedException*** exception due to the BUSINESS OBJECT POINTER CREATED FOR THE FOR-EACH LOOP.
- IN ADDITION, WE NEED TO MODIFY THE OBJECT CREATION CODE TO REFERENCE THE DLL.

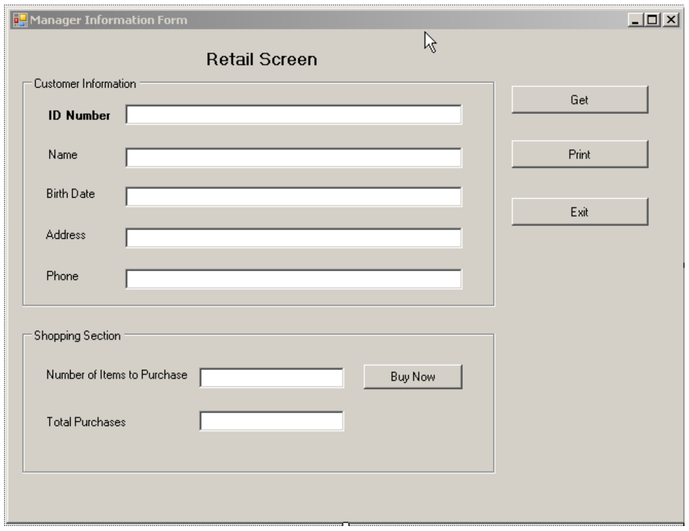
```
'************************************************************************************
    ''' <summary>
    ''' Name: Event-Handler for btnList button
    ''' Purpose: List properties of object to the listBox as comma-delimited line
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
Private Sub btnList_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles btnList.Click
        'Step A- Begin Error trapping
        Try

            'Step 1-Clear the list
            lstCustomers.Items.Clear()


            'Step 2-Create Temporary Person and Dictionary object POINTERS
            Dim objDictionaryEntry As DictionaryEntry
            Dim objItem As BusinessObjectsDLL.clsCustomer


            'Step 3-Use For..Each loop to iterate through Collection Class Object
            'GET properties of object pointed by objItem and write to listbox
            For Each objDictionaryEntry In objCustomerList

                'Step 4-Convert DictionaryEntry pointer returned to Type Person
           objItem = CType(objDictionaryEntry.Value, BusinessObjectsDLL.clsCustomer)

                'Step 5-Create the string to list
                Dim strLine As String = objItem.CustomerID & "," & _
                objItem.Name & "," & _
                objItem.SocialSecurity & "," & _
                objItem.BirthDate & "," & _
                objItem.Address & "," & _
                objItem.Phone

                'Step 6-Add string to ListBox
                lstCustomers.Items.Add(strLine)
            Next

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation!  " & objNSE.Message)
            'Step C-Traps for general exceptions.
        Catch objE As Exception
            'Step D-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub

End Class
```

Retail Management Form

- Now we need to make some small changes to the Customer Management Form. We need to the following:

  1. ADD REFERENCE TO DLL where required
  2. ADD any required Error handling when CREATING AND MODIFYING BUSINESS OBJECTS, by trapping for ***NotSupportedException*** Exception

- The ***Retail Management Form*** looks as follows:



- In addition we will automatically LOAD all customer data from file during *Form_Load* event and SAVE all customer data to file when the *Exit* button is clicked.

Step 1: Modity the Form Level Object to Use the DLL. Also we show the Form Load() event

- Modify OBJECT DECLARATION TO USE CLASS in DLL:

```
    '*********************************************************************************
    ' FORM-LEVEL VARIABLES & OBJECT DECLARATIONS SECTION
    '*********************************************************************************
    'Module-level Object POINTER Declaration
    Private WithEvents objCustomer As BusinessObjectsDLL.clsCustomer
```

Step 2: The FORM_LOAD() event-handler

- WE NEED TO TRAP FOR ***NotSupportedException*** to support our BUSINESS VALIDATION RULES:

```
    '*********************************************************************
    ' EVENT-HANDLER DECLARATIONS SECTION
    '*********************************************************************

    '*********************************************************************
    ''' <summary>
    ''' Form_Load event. Create object and popoulate Form controls
    ''' With object's default values. Also Sets text box to Read-only
    ''' in MODULE
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub frmRetailManagement_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        'Step A-Begins Exeception handling.
        Try
            'Step 1-Create EMPTY Form-Level Object
            objCustomer = New BusinessObjectsDLL.clsCustomer

            'Step 2-Populate Form Controls with Object's data
            With objCustomer
                txtName.Text = .Name
                txtIDNumber.Text = .CustomerID
                txtBirthDate.Text = CStr(.BirthDate)
                txtAddress.Text = .Address
                txtPhone.Text = .Phone
            End With

            'Step 3-Disable txtTotalPurchases Text Box to make it Read-only
            txtTotalPurchases.Enabled = False


            'Step 1-Load objects from file to collection
            objCustomerList.Load()

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation!  " & objNSE.Message)
            'Step C-Traps for general exceptions.
        Catch objE As Exception
            'Step D-Inform User
            MessageBox.Show(objE.Message)
        End Try

    End Sub
```

Step 3: The FORM_CLOSE() event-handler

- WE NEED TO TRAP FOR ***NotSupportedException*** to support our BUSINESS VALIDATION RULES:

```
'*********************************************************************************
    ''' <summary>
    '''Name: Event-Handler Form_Close()
    '''Purpose:Destroys Form-level object pointer when form closes
    '''Saves Collection objects to file and clears the collection
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub frmRetailManagement_FormClosed(ByVal sender As Object, ByVal e As
System.Windows.Forms.FormClosedEventArgs) Handles Me.FormClosed
        'Step A-Begins Exeception handling.
        Try
            'Step 1-Destroy Form-Level Objects
            objCustomer = Nothing

            'Step 2-Save objects from Collection to file
            objCustomerList.Save()

            'Step 3-Clear the Collection
            objCustomerList.Clear()

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step C-Traps for general exceptions.
        Catch objE As Exception
            'Step D-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 4: The btnExit_Click() event-handler

- NO MODIFICATION REQUIRED SINCE NO BUSINESS OBJECTS ARE CREATED OR MODIFIED:

```
    '*****************************************************************************
    ''' <summary>
    ''' Event-handler calls Form Close() method to close the Form.
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub btnExit_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnExit.Click
        'Step 1-Close yourself (Form)
        Me.Close()
    End Sub
```

Step 5: Get_Click() event-handler – We Catch a NotSupportedException for Our Business Object Validation Rules

- TRAP FOR ***NotSupportedException*** to support our BUSINESS VALIDATION RULES.

```
    '*******************************************************************************
    ''' <summary>
    ''' Calls Search method of module to search database for object
    ''' whose ID is passed as argument. Returns a pointer to the object
    ''' found, else returns a Nothing.
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub btnGet_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnGet.Click
        'Step A-Begins Exeception handling.
        Try
        'Step 1-Call Calls Collection.Item() Property to return pointer to object
            objCustomer = objCustomerList.Item(txtIDNumber.Text.Trim)

         'Step 2-If result of search is Nothing, then display customer is not found
            If objCustomer Is Nothing Then
                MessageBox.Show("Customer Not Found")

                'Step 3-Clear all controls
                txtName.Text = ""
                txtIDNumber.Text = ""
                txtBirthDate.Text = ""
                txtAddress.Text = ""
                txtPhone.Text = ""
            Else
          'Step 4-Then Data is extracted from customer object & placed on textboxes
                With objCustomer
                    txtName.Text = .Name
                    txtIDNumber.Text = .CustomerID
                    txtBirthDate.Text = CStr(.BirthDate)
                    txtAddress.Text = .Address
                    txtPhone.Text = .Phone

                    'Set total purchases
                    txtTotalPurchases.Text = CStr(.TotalItemsPurchased)
                End With
            End If

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step C-Traps for general exceptions.
        Catch objE As Exception
            'Step D-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 6: Print_Click() event-handler – Trap for NotSupportedException

- Trap for ***NotSupportedException*** exception in case the call to the Print method may return business object exceptions.

```
    '*****************************************************************************
    ''' <summary>
    ''' Event-handler call PRINT() METHOD of Form-Level object.
    ''' </summary>
Private Sub btnPrint_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnPrint.Click
        'Step A- Begin Error trapping
        Try
            'Step 1-Tell object to print itself
            objCustomer.Print()

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            'Step C-Inform User
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step D-Traps for general exceptions.
        Catch objE As Exception
            'Step E-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 7: SHOP_Click() event-handler – Trap for NotSupportedException

- Trap for ***NotSupportedException*** exception in.

```
    '*****************************************************************************
    ''' <summary>
    ''' Calls customer object Shop() method to purchase items and cleas text box.
    ''' Also displays total purchases of customer
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub btnShop_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnShop.Click
        'Step A-Begins Exeception handling.
        Try
            'Step 1-Call the Shop Method of the Object to shop and trigger event
            objCustomer.Shop(CInt(txtItems.Text.Trim))

            'Step 2-Clear Items textbox
            txtItems.Text = ""

            'Step 3-Set total purchases
            txtTotalPurchases.Text = CStr(objCustomer.TotalItemsPurchased)

            'Step B-Traps for Business Rule violations
        Catch objNSE As NotSupportedException
            MessageBox.Show("Business Rule violation! " & objNSE.Message)
            'Step C-Traps for general exceptions.
        Catch objE As Exception
            'Step D-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```

Step 8: ONSHOPPING_Click() event-handler

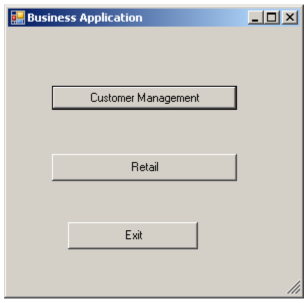
- ❑ NO MODIFICATIONS REQUIRED!

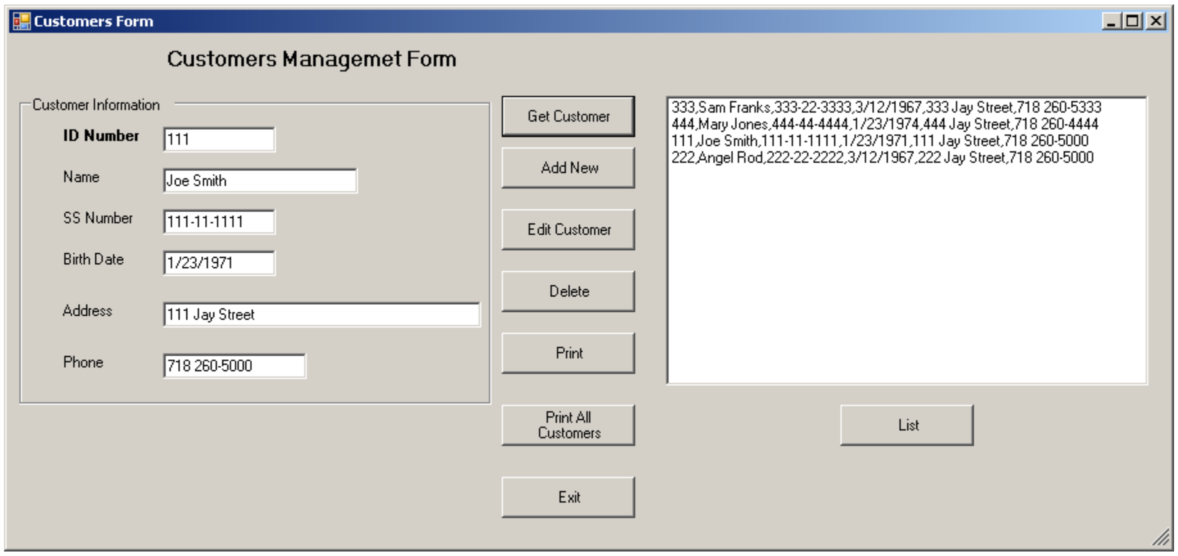
```
    '***************************************************************************
    ''' <summary>
    ''' Event-handler of Customer Objects. Triggered when Shop() method is called.
    ''' Displays a message every time customer shops.
    ''' </summary>
    ''' <param name="intTotalItems"></param>
    ''' <remarks></remarks>
Private Sub objCustomer_OnShopping(ByVal intTotalItems As Integer) Handles
objCustomer.OnShopping

        MessageBox.Show("The Total items purchased by the Customer is " &
intTotalItems)

End Sub

End Class
```

Step 4: Build & Execute Project

Step 1: Compile and Build the project.

Step 2: Execute the application.





Customers Managemet Form

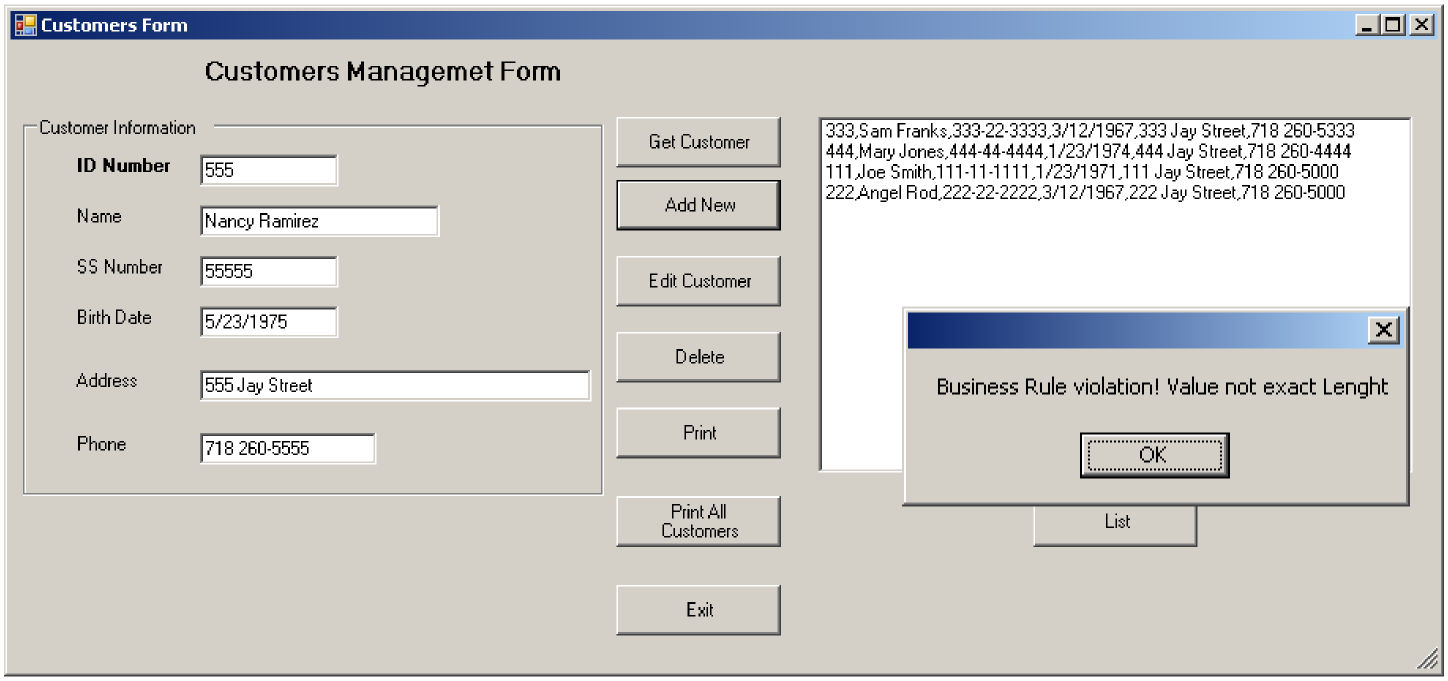
Customer Information

ID Number	111
Name	Joe Smith
SS Number	111-11-1111
Birth Date	1/23/1971
Address	111 Jay Street
Phone	718 260-5000

```
333,Sam Franks,333-22-3333,3/12/1967,333 Jay Street,718 260-5333
444,Mary Jones,444-44-4444,1/23/1974,444 Jay Street,718 260-4444
111,Joe Smith,111-11-1111,1/23/1971,111 Jay Street,718 260-5000
222,Angel Rod,222-22-2222,3/12/1967,222 Jay Street,718 260-5000
```

Step 3: Test the Business Rules

- Attempting to ADD Customer and violating the exact length Business Rule of the SS Number:



Customers Managemet Form

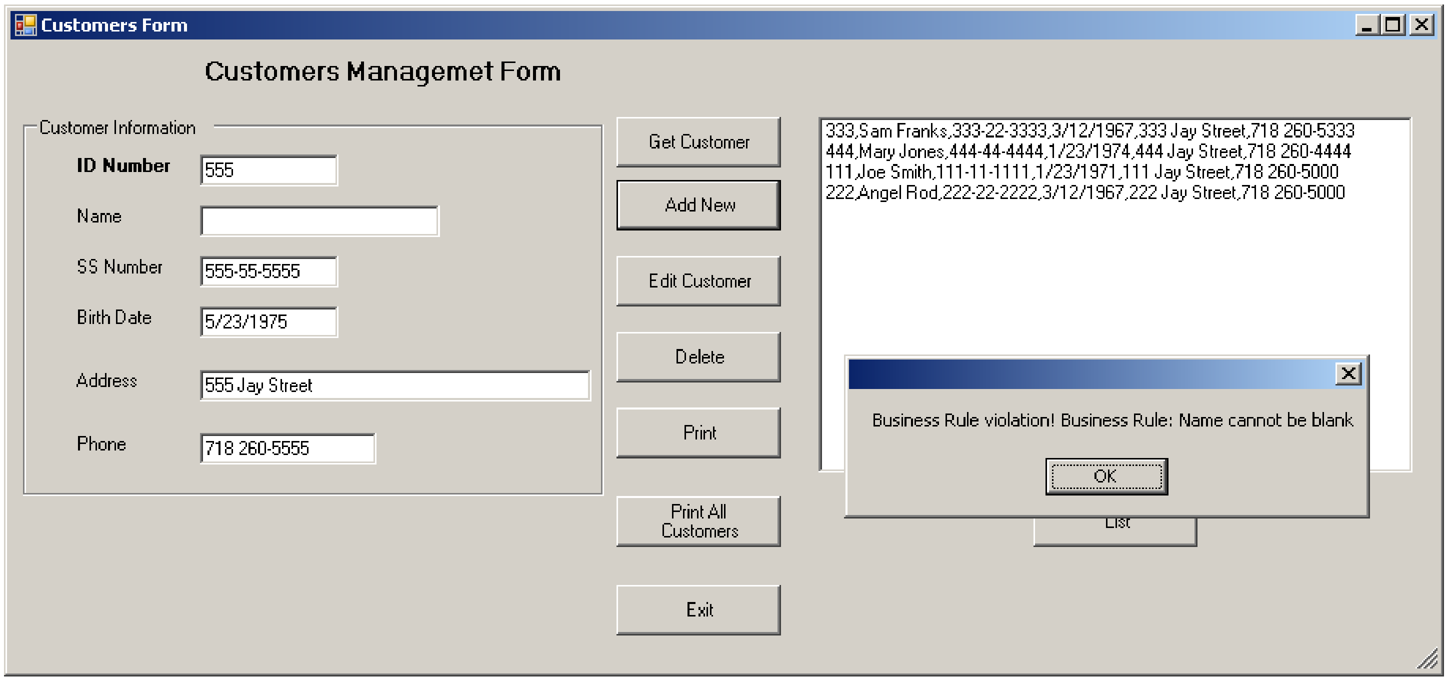
Customer Information

ID Number	555
Name	Nancy Ramirez
SS Number	55555
Birth Date	5/23/1975
Address	555 Jay Street
Phone	718 260-5555

```
333,Sam Franks,333-22-3333,3/12/1967,333 Jay Street,718 260-5333
444,Mary Jones,444-44-4444,1/23/1974,444 Jay Street,718 260-4444
111,Joe Smith,111-11-1111,1/23/1971,111 Jay Street,718 260-5000
222,Angel Rod,222-22-2222,3/12/1967,222 Jay Street,718 260-5000
```

Business Rule violation! Value not exact Lenght

- Attempting to ADD Customer and violating the NON-EMPTY Business Rule for the Name Property:

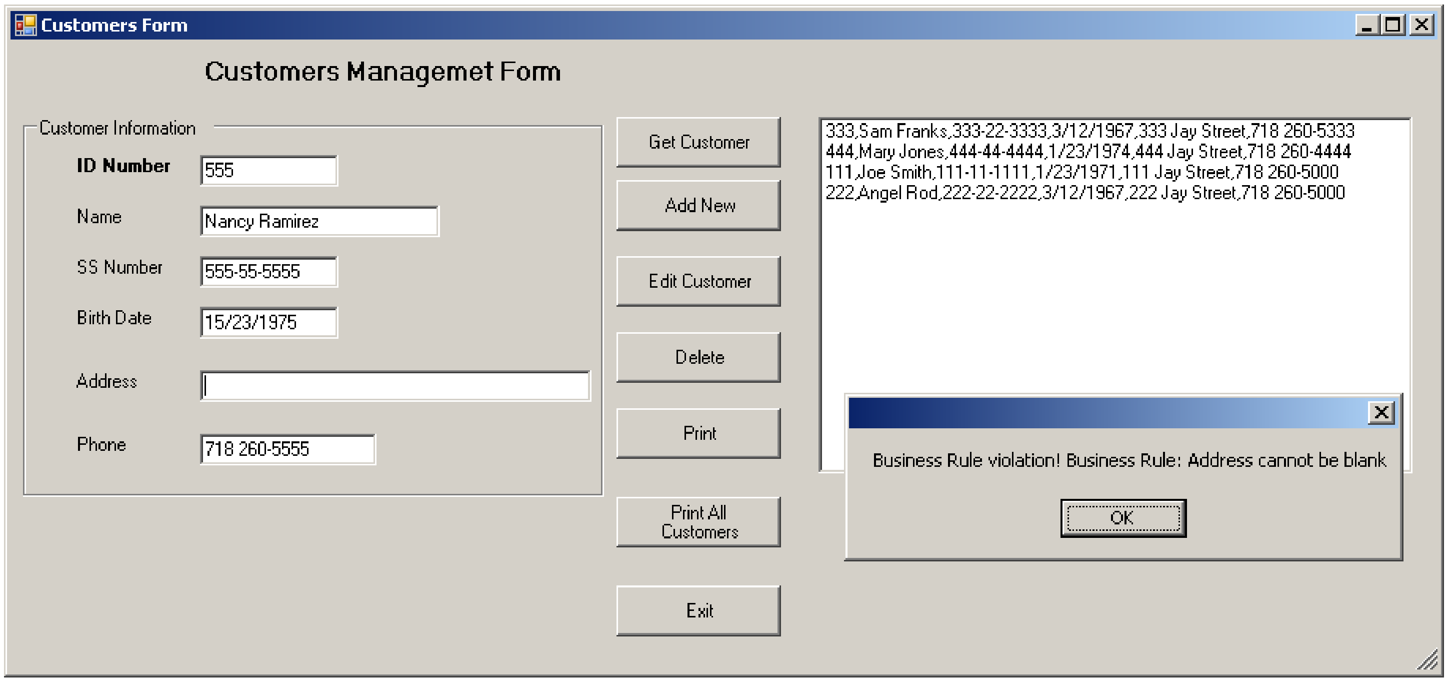


ID Number: 555
Name:
SS Number: 555-55-5555
Birth Date: 5/23/1975
Address: 555 Jay Street
Phone: 718 260-5555

```
333,Sam Franks,333-22-3333,3/12/1967,333 Jay Street,718 260-5333
444,Mary Jones,444-44-4444,1/23/1974,444 Jay Street,718 260-4444
111,Joe Smith,111-11-1111,1/23/1971,111 Jay Street,718 260-5000
222,Angel Rod,222-22-2222,3/12/1967,222 Jay Street,718 260-5000
```

Business Rule violation! Business Rule: Name cannot be blank

- Attempting to ADD Customer and violating the NON-EMPTY Business Rule for the ADDRESS Property:

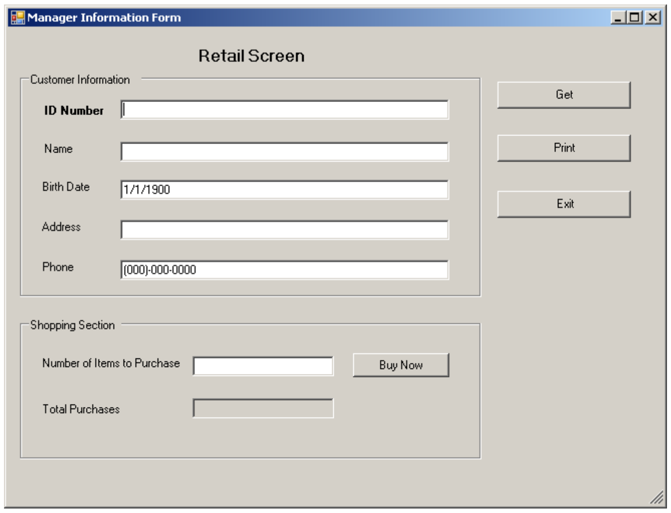


ID Number: 555
Name: Nancy Ramirez
SS Number: 555-55-5555
Birth Date: 15/23/1975
Address:
Phone: 718 260-5555

```
333,Sam Franks,333-22-3333,3/12/1967,333 Jay Street,718 260-5333
444,Mary Jones,444-44-4444,1/23/1974,444 Jay Street,718 260-4444
111,Joe Smith,111-11-1111,1/23/1971,111 Jay Street,718 260-5000
222,Angel Rod,222-22-2222,3/12/1967,222 Jay Street,718 260-5000
```

Business Rule violation! Business Rule: Address cannot be blank

- **ATTENTION!** SOME BUSINESS RULES CANNOT BE TESTED AT THIS TIME. BUSINESS RULES INVOLVE THE "DIRTY & NEW MECHANISM, WHICH ALSO WORK HAND-IN-HAND WITH THE DATABASE ACCESS METHOD! SINCE WE ARE NOT USING THE *CUSTOMER.LOAD()* METHOD AT THIS TIME, WHICH DETERMINES IF AN OBJECT IS **NEW** OR **OLD**, WE CANNOT TEST BUSINESS RULES SUCH AS "WRITE-ONCE" ETC., WHICH IS A RULE BASED ON THE OBJECT BEING NEW OR OLD.
- IN THE NEXT LECTURE, WE WILL IMPLEMENT THE DATA ACCESS CODE AND WILL BE ABLE TO TEST ALL OUR BUSINESS RULES.



Manager Information Form

Retail Screen

Customer Information

ID Number

Name

Birth Date 1/1/1900

Address

Phone (000)-000-0000

Get

Print

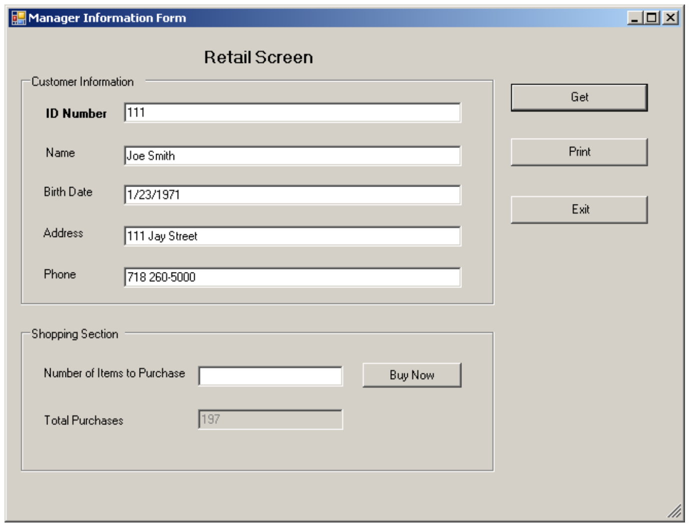
Exit

Shopping Section

Number of Items to Purchase

Buy Now

Total Purchases



Manager Information Form

Retail Screen

Customer Information

ID Number 111

Name Joe Smith

Birth Date 1/23/1971

Address 111 Jay Street

Phone 718 260-5000

Get

Print

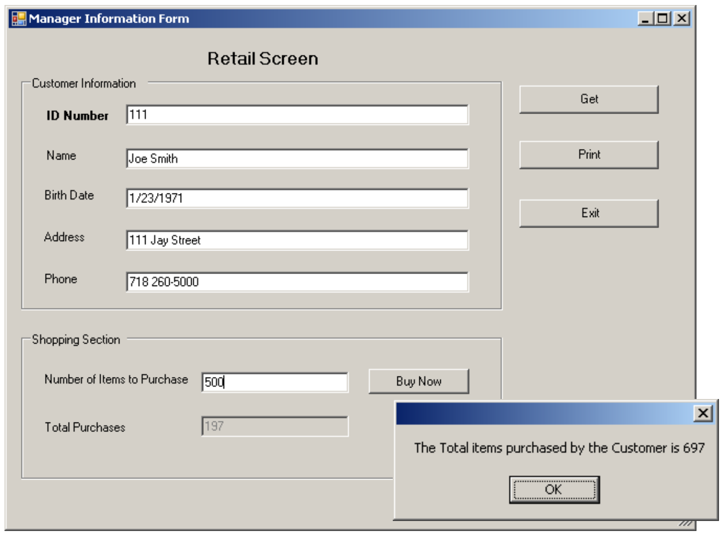
Exit

Shopping Section

Number of Items to Purchase

Buy Now

Total Purchases 197



Manager Information Form

Retail Screen

Customer Information

ID Number 111

Name Joe Smith

Birth Date 1/23/1971

Address 111 Jay Street

Phone 718 260-5000

Get

Print

Exit

Shopping Section

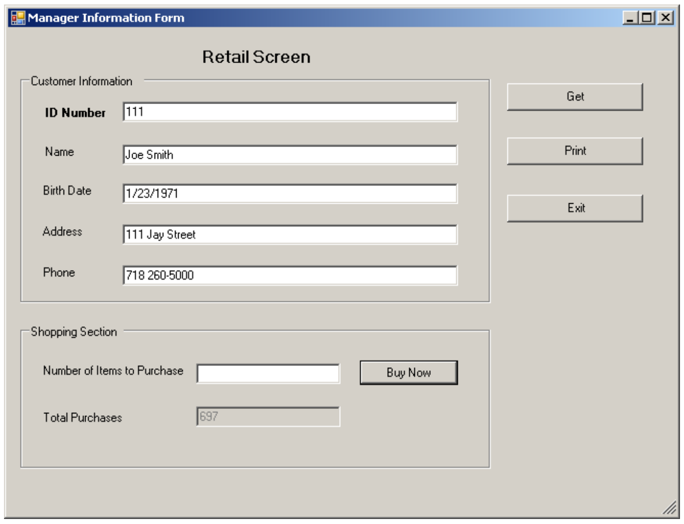
Number of Items to Purchase 500

Buy Now

Total Purchases 197

The Total items purchased by the Customer is 697

OK



Manager Information Form

Retail Screen

Customer Information

ID Number 111

Name Joe Smith

Birth Date 1/23/1971

Address 111 Jay Street

Phone 718 260-5000

Get

Print

Exit

Shopping Section

Number of Items to Purchase

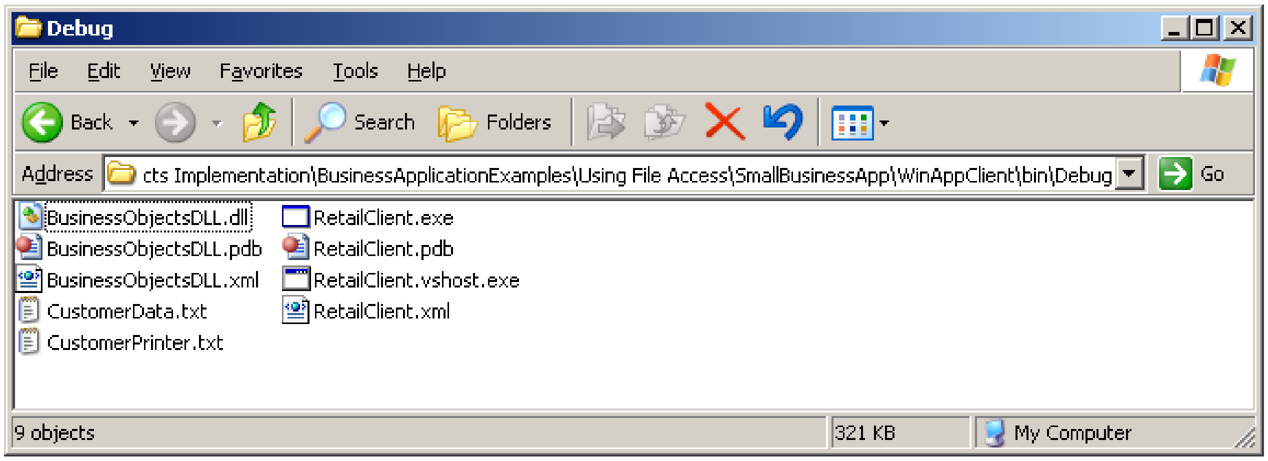
Buy Now

Total Purchases 697

Database Layer

Temporarily implemented using Files

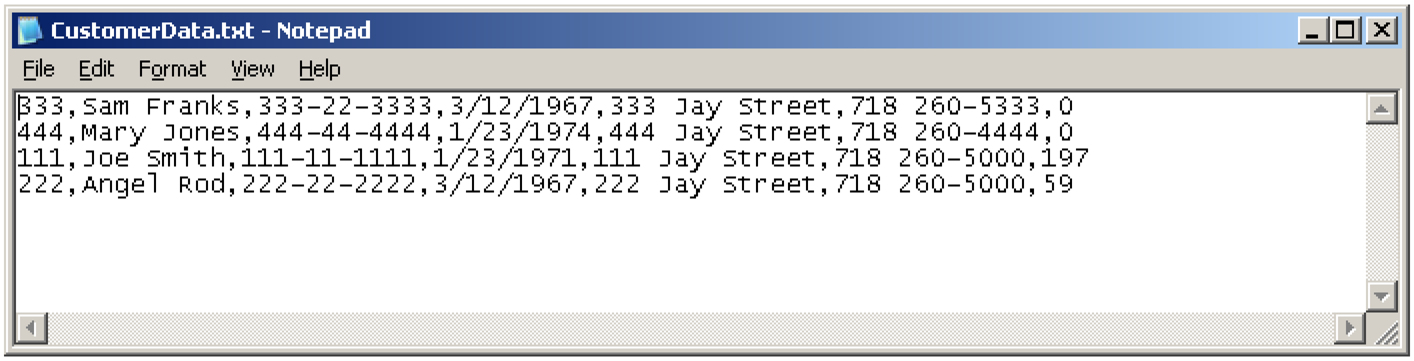
- The File which are simulating our database are located in the Bin folder of the Client Application as shown in the illustration below:


Debug
File Edit View Favorites Tools Help
Back Search Folders
Address cts Implementation\BusinessApplicationExamples\Using File Access\SmallBusinessApp\WinAppClient\bin\Debug Go

BusinessObjectsDLL.dll
BusinessObjectsDLL.pdb
BusinessObjectsDLL.xml
CustomerData.txt
CustomerPrinter.txt
RetailClient.exe
RetailClient.pdb
RetailClient.vshost.exe
RetailClient.xml

9 objects | 321 KB | My Computer

- The content of the file is formatted as comma delimited strings as shown below:



```
333,Sam Franks,333-22-3333,3/12/1967,333 Jay Street,718 260-5333,0
444,Mary Jones,444-44-4444,1/23/1974,444 Jay Street,718 260-4444,0
111,Joe Smith,111-11-1111,1/23/1971,111 Jay Street,718 260-5000,197
222,Angel Rod,222-22-2222,3/12/1967,222 Jay Street,718 260-5000,59
```

Real Data Access will be implemented in Next Lectures

- We will implement this Layer using MS Access. Although Access is not a True DBMS, nevertheless, it is commonly used for many application projects.
- We will finally implement using a true DBMS via SQL 2005 SERVER EXPRESS.