Visual Basic.NET Programming

Object-Oriented Programming

Collections & Client/Server Application Architecture

(Part II of II)

(Lecture Notes 2B)

Prof. Abel Angel Rodriguez
1.0 SortedList

1.1 SortedList Overview

- The SortedList Class is a Collection based on the DICTIONARY Type Collections. These Collections managed objects based on a string key.
- It has the following description:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
</table>
| SortedList | - Implements a DICTIONARY collection whose size is dynamically increased, but is based on a collection of a string key-and-value pairs.  
|          | - This Collection is sorted by the keys and is accessible by key and by index.  
|          | - Note that you must manipulate this Collection via the string Key, but you can access the objects via the key or index.  
|          | - You can simply create object from these class and use the properties and methods.  
|          | - This Collection is of the type Dictionary that use Key-Value pair, which means that the DictionaryEntry object is used to stored the objects inside the collection  
|          | - In some cases, when using the properties and methods, type conversion is required using the CType() method. |

Using Sorted List

- Use a SortedList when the objects we need to store can be access, added, deleted based on a Key.
- Note that if the application we are writing requires that a key is provided or an Customer ID/key is known or obtained when searching for objects, then the SortedList is a good choice for implementing the application.
- These type of collections are best used when the Objects in which we are storing already contain the key as part of their data, such as a Social Security Number, Customer ID, Student ID, License ID etc. These id numbers or keys are available by the object.
1.2 SortedList Properties & Methods

- Since using the `SortedList` Collection really involves calling methods and properties of the Collection Object, we simply need to look at the list of properties and methods available to us.
- The following tables illustrate some of the basic properties and methods of the SortedList Collection:

### Public Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Syntax</th>
<th>Description &amp; Exception Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>colObject.Count</td>
<td>Gets the number of elements actually contained in the <code>SortedList</code>.</td>
</tr>
</tbody>
</table>
| Item     | Get: Object = colObject.Item(Key)  
          Set: colObject.Item(Key) = Object | Gets or Set or sets the element at the specified index.  
GET:  
- In GET mode the ITEM with specified KEY is returned  
- In addition if the KEY is not found, a NULL is returned  
SET:  
- In SET mode, the ITEM at the specified KEY is REPLACED, thus modified.  
- In addition if the KEY is NOT FOUND, a NEW ITEM is ADDED to the collection instead.  
- This is very important, when setting, a new item is added if key is not found. WE MAY NOT WANT A NEW ITEM ADDED IN MOST CASES, so we need to add Code to test or verify that the collection CONTAINS, the item, prior to attempting to SET. We can use the Contains() method to verify it exists prior to setting.  
- Exceptions raised:  
  - `ArgumentNullException` – Key is a NULL or NOTHING  
  - Data Type Conversion with CType() or DirectCast():  
    Option Strict ON – Must convert object being accessed. |
## Public Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Syntax</th>
<th>Description &amp; Exception Raised</th>
</tr>
</thead>
</table>
| **Add** | `colObject.Add(Key, Object)` | Adds an object to the `SortedList` based on `KEY` argument  
- If key already exist an `ArgumentException` is raised.  
- **Exceptions raised:**  
  - `ArgumentException` – KEY already exist or DUPLICATE KEY  
  - `ArgumentNullException` – Key is a NULL or NOTHING |
|        | Example 1: `colEmployees.Add(strKey, objEmp1)` |  |
|        | Example 2: `colEmployees.Add("111", objEmp1)` |  |
|        | Example 3(Best Practice): `colEmployees.Add(objEmp1.SSNum, objEmp1)` |  |
| **Clear** | `colObject.Clear` | Removes all elements from the `SortedList`. |
| **Contains** | `colObject.Contains` | Determines whether an object or element is in the `SortedList`.  
- **Return Value:** Boolean (True if exists, False otherwise).  
- **Exceptions raised:**  
  - `ArgumentNullException` – Key is a NULL or NOTHING |
| **Remove** | `colObject.Remove(Key)` | Removes the Object at the specified `Key` from the `SortedList`.  
- **Note that if KEY is NOT FOUND, then nothing is done and the collection stays unchanged. The collection DOES NOT raise any error or does not communicate with the calling program indication key is not found. It is the responsibility of the programmer to put logic to inform that nothing was done since the object was not found.**  
- **Exceptions raised:**  
  - `ArgumentNullException` – Key is a NULL or NOTHING |

### 1.3 Data Type Conversions: CType() or DirectCast()

- This Class is of Dictionary Type. The objects stored in the collection are of type `DICTIONARYENTRY` objects. We need to perform data type conversions or Casting when using objects retrieved from the collection.

#### Item Property
- You may need to convert the data type of the object stored in the `SortedList`.
- **Rule:**  
  - If `Option Strict` is set to **OFF**, no conversion needed.  
  - If `Option Strict` is set to **ON**, you must convert the data type of object in collection using `CType()` or `DirectCast()`

#### For Each..Next Loop
- Using `For..Each..Next` loop in the `SortedList` is supported.
- **Rule:**  
  - **IMPORTANT!** Nevertheless, unlike the `ArrayList`, even if `Option Strict` is set to **OFF**, we need to convert the data type using `CType()` for a `For Each Loop`.  
  - **IMPORTANT!** Therefore we must use `CTYPE()` conversion when using For-Each whether `Option Strict` is set to **OFF** or **ON**
In this section we will now modify the Customer List example to use a SortedList Collection that will search, store and manage the Customer Objects using a key. The unique key will be chosen internally to the Customer Object via it’s ID Number.

In the previous ArrayList Customer List example we showed that for a Customer Transaction Application using an index-based collection was not the best choice since we had to first search the collection for the customer based on their ID and when found, return the INDEX, then we were able to add, remove, insert, etc. We had to access the list twice for every transaction.

We will cover several examples using a Key-Based Collection, such as the SortedList.

In this example we will cover the following:

- Use SortedList to manage the customer Objects.
- Add exception handling to trap the various errors and exceptions raised by the Collection.
- Set Option Strict ON, to see how we must convert data types using CType() in order to use objects retrieved from collections.

Example 1.4 — SortedList & Customer Objects Example

Problem statement:
- Upgrade the Using a Customer Object/ArrayList Collection application by replacing the ArrayList with a SortedList Collection.
- Upgrade the application as follows:

  Object Management:
  - Replace the ArrayList with a SortedList Collection.
  - Set Option Strict ON and Convert Data Types using CType() function as necessary to demonstrate how this feature affects the use of the Collection
  - Same requirements as previous examples.

  Form Requirements:
  - Same requirements as previous example.
  - In order to comply better with our Application Development Architecture and keep the Form with User Interface code only, we will modify the Form code slightly differently than the previous example to show an alternate method.

  Other Requirements:
  - Our objectives is to implement this application with the three-tier application architecture described in class:
Therefore, all UI interaction code should reside in the Presentation layer of Forms. All processing should be done in the objects as best as possible. Unfortunately in this example, we will continue to keep most of the processing code in the module, which is actually part of the Presentation Layer. This is NOT what we want to do and DOES NOT follow our architecture, but this is done at this time for teaching purpose only, at this time we are not ready until we have the mechanism to do so. In later examples were we will move the processing code to a Collection Class or the Business Object Layer.
## HOW IT’S DONE:

### Part I – Create The Application:

#### Step 1: Start a new Windows Application project:

#### Step 2: Add a Form to the project and set its properties for an Employee:

<table>
<thead>
<tr>
<th>Object</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>Name</td>
<td>frmEmployeesForm</td>
</tr>
</tbody>
</table>

#### Step 3: Add a Standard Module set its properties as previous example:

<table>
<thead>
<tr>
<th>Object</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>Name</td>
<td>modMainModule</td>
</tr>
</tbody>
</table>

#### Step 4: Set the Project’s properties to behave as a Module-Driven Windows Application:

<table>
<thead>
<tr>
<th>Object</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>Name</td>
<td>EmployeeFormWinApp</td>
</tr>
<tr>
<td>Project</td>
<td>Startup Object</td>
<td>Sub Main()</td>
</tr>
</tbody>
</table>

---

![Visual representation of project settings and properties](image-url)
Step 5: Prepare to Reuse the Person Class from Previous ArrayList Application, by Copying the File from previous Application Folder to the Folder of this Windows Application Project

1. Using Windows Explorer, navigate to the previous ArrayList Application folder of the previous example.
2. Copy/Paste the file clsPerson.vb to this Project folder

Step 6 & 7: Add the Class to the Project

1. In the Project Menu, select Add Existing Item… and navigate to the project folder
2. Select the clsPerson.vb File and click OK
3. The class is now part of the project and ready to be reused!

Option Strict On

'Imported Libraries
Imports System.IO 'For file access code
Public Class clsPerson
  '*********************************************************************
  Private mName As String
  Private m_IDNumber As String
  Private m_BirthDate As Date
  Private m_strAddress As String
  Private m_Phone As String
  '*********************************************************************
Step 8: Leave the Property Procedure as is:

```vbnet
'Property Procedures
Public Property Name() As String
Get
    Return m_Name
End Get
Set(ByVal Value As String)
    m_Name = Value
End Set
End Property

Public Property IDNumber() As String
Get
    Return m_IDNumber
End Get
Set(ByVal Value As String)
    m_IDNumber = Value
End Set
End Property

Public Property BirthDate() As Date
Get
    Return m_BirthDate
End Get
Set(ByVal Value As Date)
    m_BirthDate = Value
End Set
End Property

Public Property Address() As String
Get
    Return m_strAddress
End Get
Set(ByVal Value As String)
    m_strAddress = Value
End Set
End Property

Public Property Phone() As String
Get
    Return m.Phone
End Get
Set(ByVal Value As String)
    m.Phone = Value
End Set
End Property
```
Step 9: Constructor Methods:

`'*********************************************************************
'Class Constructor Methods

'Default Constructor
Public Sub New()
'Note that private data members are being initialized
m_Name = ""
m_IDNumber = ""
m_BirthDate = #1/1/1900#
m_strAddress = ""
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 being set via the constructor
Me.Name = N
Me.IDNumber = SSNum
Me.BirthDate = BDate
Me.Address = Adr
Me.Phone = Ph
End Sub

Step 10: PrintPerson() Method stays the same:

`'*********************************************************************
'Regular Class Methods

'Author of base class allows sub classes to override 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_IDNumber & ", " & _
m_BirthDate & ", " & m_strAddress & ", " & m_Phone)

'Close StreamWriter Object
objPrinter.Close()
End Sub`
Part II – Module

Overview

- We will add PROCESSING METHODS in the module to handle the manipulation of the collection. Also we add Exception Handling using Try/Catch Blocks to trap for errors generated by the SortedList Collection.

**Step 1: In Module Add the Following Code:**

- Code any Global & Private Variable declarations and Sub Main()
  1. **Option Strict ON.** By Option Strict being on, we need to convert the object’s data type using \texttt{CType()} function when retrieving objects from collection.
  2. Import the \texttt{System.Collections} Library to support the File I/O features
  3. Use a SortedList Collection to store employees
  4. Declare Global Employee Form Object & Login Form object
  5. Add methods to support the processing required by the forms: Add, Edit, Search, Remove, PrintObject, Print All objects, & Authenticate method to search and authenticate an employee.

```vbscript
Option Explicit On
Option Strict On
Imports System.Collections

Module modMainModule

'Declare Public Array of Person Objects
Public CustomerList As New SortedList

'Declare Form Object
Public objCustomerForm As frmCustomerForm = New frmCustomerForm()
```

**Step 2: Sub Main:**

```vbscript
'*' <summary>
'*' Name: Main()
'*' Main routine that controls flow of program
'*' Step 1-Calls the InitializeList() method to perform any required Initialization
'*' Step 2-Calls Customer Form object's ShowDialog() method to display itself
'*' </summary>
'*' <remarks></remarks>
Public Sub Main()

    InitializeList()

    'Display Customer Form
    objCustomerForm.ShowDialog()

End Sub
```
Step 3: InitializeList Method – Create Objects, Initialized and add to list:

Public Sub InitializeList()
    'Step A-Begins Exception handling.
    Try
        'Step 1-Create default object
        Dim objDefaultCustomer As New clsPerson("Joe Smith", "111", #1/23/1971#, _
            "333 Jay Street", "718 260-5000")

        'Step 2-Add object to List
        CustomerList.Add(objDefaultCustomer.IDNumber, objDefaultCustomer)

        'Step B-Traps for ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step C-Throw Collection ArgumentNullException
            Throw New SystemArgumentNullException("Invalid Key Error: " & objX.Message)
        Catch objY As ArgumentException
            'Step E-Throw an ArgumentException to calling programs
            Throw New SystemArgumentException("Duplicate Key Error: " & objY.Message)
        Catch objE As Exception
            'Step G-Throw an General Exception
            Throw New SystemException("Initialize Error: " & objE.Message)
    End Try
End Sub
Step 4: Create GetItem Method:

```vbnet
Public Function GetItem(ByVal strKey As String) As clsPerson
    'Step A- Begin Error trapping
    Try
        'Step 1-Calls Collection.Item(Key) Method to get object from collection
        'Use CType() function to convert object retrieved from list to clsPerson
        Return CType(CustomerList.Item(strKey), clsPerson)
    '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("GetItem ID Error: " & objE.Message)
    End Try
End Function
```
Step 5: Implement the Sub Add method to manage the addition of objects into the list:

- **IMPORTANT!** In his implementation of the ADD we use the native Collection.Add(key, Object) to add an object to the collection.
- The main point here is that the object is created OUTSIDE the method and passed as argument.
- This method is a sub procedure and returns NO values. This was done since the native collection.add is also a Sub.

```vbnet
Public Sub Add(ByVal strKey As String, ByVal objItem As clsPerson)
    'Step A- Begin Error trapping
    Try
        'Step 1-Calls Collection.Add(Key, Object) Method to Add object
        CustomerList.Add(strKey, objItem)
    Catch objX As ArgumentNullException
        'Step B- Traps for ArgumentNullException when key is Nothing or null.
        'Step C-Throw NULL key exception ArgumentNullException
        Throw New System.ArgumentNullException(objX.Message)
    Catch objY As ArgumentException
        'Step E-Throw an Duplicate Key ArgumentException to calling programs
        Throw New System.ArgumentException(objY.Message)
    Catch objE As Exception
        'Step G-Throw an general exceptions
        Throw New System.Exception("Add Method Error: " & objE.Message)
    End Try
End Sub
```
Step 6: Implement the OVERLOADED Add method to manage the addition of objects into the list:

- **IMPORTANT!** This implementation of the ADD method is different than the previous. In this ADD we take as arguments the individual values or arguments that are used to create the object. The customer object is created inside this method and NOT on outside the method or a form.
- This is done to comply with our architectural requirement to keep the Form code with less processing. The values from the textboxes of the Form can be passed directly to this method.

```vbnet
Public Sub Add(ByVal strName As String, ByVal strIDNum 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 clsPerson
    With objItem
      .Name = strName
      .IDNumber = strIDNum
      .BirthDate = dBDate
      .Address = strAddress
      .Phone = strPhone
    End With

    'Step 3-Use Collection.Add(Key, Object)to add object. Object ID used as Key
    CustomerList.Add(objItem.IDNumber, objItem)
  Catch objX As ArgumentNullException
    'Step C-Throw Collection ArgumentNullException
    Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
  Catch objY As ArgumentException
    'Step E-Throw an ArgumentException to calling programs
    Throw New System.ArgumentException("Duplicate Key Error: " & objY.Message)
  Catch objE As Exception
    'Step G-Throw an general exceptions
    Throw New System.Exception("Add Method Error: " & objE.Message)
  End Try
End Sub
```
Step 7: Implement the Edit method to manage the process of modifying objects in the list:

- **IMPORTANT!** In his implementation of the EDIT we use the Item property of the Collection to REPLACE the object identified by the KEY. The main point here is that the object and key are created OUTSIDE the method and passed as argument.
- This method was created as a function, so we can return a confirmation that the object was modified.

```
Public Function Edit(ByVal strKey As String, ByVal objItem As clsPerson) As Boolean
    'Step A- Begin Error trapping
    Try
        'Step 1-Verify object exist
        If CustomerList.Contains(strKey) Then
            'Step 2- Sets CollectionObject.Item(Key) = object
            CustomerList.Item(strKey) = objItem
            'Step 3- Return confirmation
            Return True
        Else
            'Step 4- Return object not found
            Return False
        End If
    Catch objX As ArgumentNullException
        'Step C-Throw Collection ArgumentNullException
        Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
    Catch objE As Exception
        'Step E-Throw an general exceptions
        Throw New System.Exception("EditItem Error: " & objE.Message)
    End Try
End Function
```
Step 8: Implement the OVERLOADED Edit method to manage the process of modifying objects in the list:

- **IMPORTANT!** This implementation of the EDIT method is different than our previous examples. Here we take as arguments the individual values which make up the object to be edited which includes the KEY as well. THIS MEANS ALL PROCESSING IS DONE INSIDE THIS METHOD. The Form code can be kept simple with hardly no processing.
- THE ACTUAL OBJECT IN THE COLLECTION WAS MODIFIED NOT A REPLACEMENT, THUS PRESERVING THE ORIGINAL OBJECT.

```vbnet
'************************************************************************************
''' <summary>
''' Name: OVERLOADED EditItem(value1, value2, etc.) Method
''' Purpose: Sets the individual properties of the object residing in Collection
''' NO REPLACEMENT, ACTUAL OBJECT IN COLLECTION IS MODIFIED NOT REPLACED
''' </summary>
''' <param name="strName"></param>
''' <param name="strIDNum"></param>
''' <param name="dBDate"></param>
''' <param name="strAddress"></param>
''' <param name="strPhone"></param>
''' <returns></returns>
''' <remarks></remarks>

Public Function Edit(ByVal strName As String, ByVal strIDNum 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 clsPerson

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

'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.
    ' NOTE THAT SINCE THE ID NUMBER OF THE PERSON IS THE KEY, WE DO NOT
    ' WANT TO MODIFY IT OR TAMPER WITH IT IN ANY WAY
    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 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: Implement the Remove method to manage the removal of objects from the list:

- **IMPORTANT!** This implementation of the REMOVE method we verify if object exits prior to removal. This is done again because the `Collection.Remove` does not inform the caller if the object is NOT in the collection.

```vbnet
'************************************************************************************
' *** Name:Function Remove(Key)    
' *** Purpose:Remove object from collection based on key.  
' *** Search is done to verify if object exits prior to removal  
' *** </summary>  
' *** <returns></returns>  
' *** <remarks></remarks>  
Public Function Remove(ByVal strKey As String) As Boolean  
  Try  
    'Step A- Begin Error trapping  
    Try  
      'Step 1-Verify object exists  
      If CustomerList.Contains(strKey) Then  
        'Step 2-Calls CollectionObject.Remove(Key) Method  
        CustomerList.Remove(strKey)  
        'Step 3-Return True since found and removed  
        Return True  
      Else  
        'Step 4-Return False since not found  
        Return False  
      End If  
    Catch objX As ArgumentNullException  
      'Step C-Throw ArgumentNullException when key is Nothing or null.  
      Throw New SystemArgumentNullException("Invalid Key Error: " & objX.Message)  
    Catch objE As Exception  
      'Step E-Throw an general exceptions  
      Throw New System.Exception("Remove Error: " & objE.Message)  
    End Try  
  End Try  
End Function  
```
Step 10: Implement the PrintCustomer method to manage the process of printing an objects to File:

```vbnet
Public Function PrintCustomer(ByVal strKey As String) As Boolean
    'Step A- Begin Error trapping
    Try
        Dim objItem As clsPerson

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

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

        '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 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("PrintCustomer Error: " & objE.Message)
        End Try
    End Function
```
Step 11: Implement the PrintAllCustomers method to print all the Employees in the list to File:

```vbnet
Public Sub PrintAllCustomers()
    'Step A- Begin Error trapping
    Try
        'Step 1-Step 1-Create Temporary Person and Dictionary object POINTERS
        Dim objDictionaryEntry As DictionaryEntry
        Dim objItem As clsPerson

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

            '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("PrintAllCustomer Method Error: " & objE.Message)
        End Try
    End Sub
End Module
```

Brief Discussion of Module Code
- In each of the Method the manage the Collection, we added Try/Catch blocks to trap the errors generated by the Collection.
- Since Option Strict is On, and the SortedList Collection stores object as DictionaryEntry objects, we were forced to use the CType() method to convert to Employee Object when ever we are retrieving objects from the Collection.
Part III – User Interface Form

Overview
- We will add Try/Catch Block to the Form in order to trap the errors generated by the Method that manage the Collection in the Module.

Step 12: Add the required controls to the Form:

Step 13 In the Form
Level Object, Load(), Close() & Exit() handlers:

Option Explicit On
Option Strict On

Public Class frmCustomerForm
    Inherits System.Windows.Forms.Form

    'Declare Form Level Object POINTER
    Private objCustomer As clsPerson

    '************************************************************************************
    ''' <summary>
    ''' Name: Event-Handler Form_Load
    ''' Purpose:Nothing is done at this time
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub EditForm_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
        Handles MyBase.Load
    End Sub

    '************************************************************************************
    ''' <summary>
    ''' Name: Event-Handler Form_Close()
    ''' Purpose:Destroys Form-level object pointer when form closes
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    ''' <remarks></remarks>
    Private Sub frmEditForm_Closed(ByVal sender As Object, ByVal e As System.EventArgs)
        Handles MyBase.Closed
            'Destroy Form-Level Objects
            objCustomer = Nothing
    End Sub

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

            objCustomer = Nothing

            Me.Close()

    End Sub
Step 14 Form Exit Click Event:

```vbnet
'************************************************************************************
''<summary>
''Name: Event-Handler for for Exit button
''Purpose: 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
    Me.Close()
End Sub
```
Private Sub btnGetCustomer_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnGetCustomer.Click
    'Step A-Begins Exception handling.
    Try
        'Step 1-Call Overloaded GetItem(ID) to search for object that match ID
        objCustomer = GetItem(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 = .IDNumber
                txtBirthDate.Text = CStr(.BirthDate)
                txtAddress.Text = .Address
                txtPhone.Text = .Phone
            End With
        End If
    End Try
    '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 14: Enter Code for the Add_Click Event-handler:

```vbnet
Public 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 Add(Value1,Value2,Value3,...) method, properties are passed as argument
        Add(txtName.Text.Trim, txtIDNumber.Text.Trim, CDate(txtBirthDate.Text), _
            txtAddress.Text.Trim, txtPhone.Text.Trim)
    Catch objY As ArgumentException
        'Step C-Inform User
        MessageBox.Show(objY.Message)
    Catch objX As ArgumentException
        'Step E-Inform User
        MessageBox.Show(objX.Message)
    End Try
End Sub
```
Step 15: Enter code for EditCustomer Event:

```vbnet
'************************************************************************************
''' <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 = Edit(txtName.Text.Trim, txtIDNumber.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 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 16: Enter Code for the Delete_Click Event:

```vbnet
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 = 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 ArgumentException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step C- Inform User
            MessageBox.Show(objX.Message)
        End Catch

        'Step D- Traps for general exceptions.
        Catch objE As Exception
            'Step E- Inform User
            MessageBox.Show(objE.Message)
        End Try

    End Sub
```
Step 17: Add Code for Print Event:

```vbnet
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 = PrintCustomer(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 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 18: Add Code for PrintAll Event:

```csharp
'************************************************************************************
''' <summary>
''' Name: Event-Handler for btnPrintAllCustomers button
''' Purpose: Prints all Objects in the list
''' </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.
        PrintAllCustomers()
    Catch objE As Exception
        'Step C-Inform User
        MessageBox.Show(objE.Message)
    End Try
End Sub

End Class
```
Part IV – Output & Summary

Summary
- Run the program and you can then perform the necessary operations on the list.
- Also you can purchase and print customer information to file.
- As a final word, we notice that in the previous ArrayList examples, choosing the ArrayList for this type of application forced us to add additional code that normally we would want the Collection to handle for us.
- In short, ArrayList was NOT the best choice of Collection for this transactional based application.

Form Output:

File Output:

Joe Smith, 111, 1/23/1971, 333 Jay Street, 718 260-5000
Angel Rod, 222, 12/12/1973, 222 Flatbush, 718 260-3333
2.0 Strong-Typed Collection Classes & Application Architecture

2.1 Custom-Collection Class Overview

- As stated in the introductory lesson on Collections, a Collection will allow you to ADD any type of data. This is normally not desirable. For example, if we have a collection storing “Employee” objects, there is nothing stopping program code from accidentally ADDING objects of type say “Automobile”.
- A Strong-Typed Custom Collection Class is simply a Class created to encapsulate or protects the Collection Object.
- It is called a Strong-Typed Collection, because the public properties and methods of the custom class ensures that only the data type of the items we want to add are allowed.

Issue with Customer Management Example

- The Customer & Employee Management Application that we created is our examples, have the following drawbacks:
  - ISSUE # 1 – The Collection Object is accessible to the entire program and there is no protection from stopping unauthorized code from adding objects to the Collections that are NOT Customer Objects. This will cause the application to fail or work incorrectly.

  BEST PRACTICE! Solution to Issue #1
  - Creating a Strong-Type Collection Class by deriving an creating a Custom Collection Class from the DictionaryBase Collection.
  - This Class that we inherit will inherit all the feature and functionality of the Dictionary Class structure.
  - The properties and methods we created will ensure that only the correct data type is allowed.
  - Properties and Methods will wrap the functionality of the Collection Object. To the outside world using the Class is like using the Collection Object itself, but only the data type and policies dictated by the Collection Class can be passed to the Collection Object, thus protecting it from unauthorized code.

2.2 CollectionBase & DictionaryBase

- The .NET framework provides several Collection Base Classes you can inherit from. Examples are CollectionBase & DictionaryBase.
- A brief description:
  - CollectionBase – This Collection Base Class is of the Index or ILIST type. You can derive classes which provide an indexed based list or collection such as theArrayList.
  - DictionaryBase – This Collection Base Class is of the KEY or DICTIONARY type collection. You can derive classes which provide a KEY/VALUE pair such the SORTEDLIST.

CollectionBase versus DictionaryBase

- For your applications, you can use either one of these base classes to derive your Strong-Type Custom-Collection Classes.
- In my opinion, it really depends on what you are doing. For example, for an application such as the Customer Management or any transactional based application, I believe the DictionaryBase is best,
- DictionaryBase type collection provide a faster search mechanism to find objects based on the key. The key could be a Social Security Number, License Number, Unique Employee ID etc.
- On the other hand you could use CollectionBase to accomplish the same thing, but because it is INDEXED based. To search for an object you will need to know the INDEX, Employees, Customer etc, don’t walk around remembering indexes, so you will be forced to create additional SEARCH(ID) methods to search the collection for the object based on its ID.
- In most cases you may need a combination of both types to get the job done. Either way, we now have mechanism in place to manage & retrieve object from a list and provide a strong-typed method to do so.
2.2 Implementing the Client/Server Application Architecture

Issues with Implementing the Application Architecture

- Since our introduction to the Client/Server Application Architecture in CS608, our objectives were to create our application based on an architecture, which separates USER-INTERFACE (UI) code from PROCESSING OR IMPLEMENTATION code.
- PROCESSING code is to reside ONLY in the Business Object Layer or Classes. NO PROCESSING CODE IN THE Presentation layer or User-Interface FORMS.
- As you recall, the two versions of the architecture looks as follows:

- Up to now, we did not have the mechanism for properly implementing this architecture, primarily placing processing code in the Business Object Layer.
- In all our examples, we were forced to place our processing code, such as the ADD(), REMOVE(), EDIT(), PRINT(), PRINTALL() in the MODULE. As you recall, the MODULE is actually part of the Presentation layer or User-Interface (UI).
- This undesirable architecture looked as follows:

- Note that it is in the Presentation Layer, the business logic, processing, validation and data access is done from the Presentation Layer
Implementing the Application Architecture Using Custom-Collection Classes

- We now have the mechanism to place general processing code in the Business Object Layer, by using Custom-Collection Classes.
- We can derive/create a Custom-Collection Class either from `CollectionBase` or `DictionaryBase`.
- This Custom Class can contain properties and methods to manipulate the Collection storing our objects and in addition contain any other processing Code we need.
- So we will place our ADD(), REMOVE(), EDIT(), PRINT(), PRINTALL() and any other processing method required to manage the collection in our collection classes.
- From this point forward, we will be implementing the Client/Server Application Architecture in all the application we create:
## 2.2 DictionaryBase Overview

- **The DictionaryBase** Collection Class is a Must-Inherit Base Class. Based on the DICTIONARY Type Collections. These Collections managed objects based on a string **key**
- We can derive our own Classes or **Custom-Collection Classes** from this Base Class.
- It has the following description:

### Class | Description
--- | ---
**DictionaryBase** | Provides an abstract MustInherit base class for a strongly typed collection. Implements a DICTIONARY base on key as with the SortedList. Dictionary Base contains internally an IDictionary Collection Object. This is a MustInherit Base Class that we MUST derived our own Collection Class. We cannot create object from this class, we must first derive our own sub Collection Class and then create object of the derived class. This Collection is of the type Dictionary that use Key-Value pair, which means that the DictionaryEntry object is used to stored the objects inside the collection. In some cases, such as Option Strict ON, when using the properties and methods of the objects stored in Collection, type conversion is required using the CType() method. For..Each..Next Loop are supported. Type conversion is required within the loop structure using the CType() method to handle objects stored in collection.

- The DictionaryBase derived Collection Classes store, access, add, deleted based on an **Key**.
- Note that knowledge of the key must be known or obtained when searching for objects. If the key is readily available than the DictionaryBase derived Collection Class is a good choice for implementing the application.
- On the other hand, if the key is not available, then we need to write more code to obtain the key before finding the object we seek.
- These types of collections are best used when the Objects in which we are storing already contain the key as part of their data, such as a **Social Security Number**, **Customer ID**, **Student ID**, **License ID** etc. These id numbers or keys are available by the object we are trying to find, thus by querying the object we want to find, we can get its key and then quickly search and find it in the collection.

This diagram below is an illustration of the DictionaryBase Class and some of its PROPERTIES AND METHODS.

- Note the following in the diagram shown:
  - Base class contains a PROTECTED property named DICTIONARY, which is the actual DICTIONARY COLLECTION OBJECT that stores the data
  - Internal Dictionary object contains the PROPERTIES & METHODS to manage its collection (ADD, REMOVE, CONTAINS, CLEAR, ETC.)
  - Therefore you will need the **MyBase** keyword or MYBase.Dictionary.XXX, where XXX is a property or method, in order to manage the collection.
  - Note that the DictionaryBase class contains a few properties and methods that perform the same operation as the DICTIONARY object properties and methods. Examples are Count, & Clear().
  - This means you don’t need to make a call MyBase.Dictionary.Count or you can simply call MyBase.Count directly from the derived or subclass().

### Class MustInherit DictionaryBase

<table>
<thead>
<tr>
<th>Properties:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
</tr>
</tbody>
</table>

### Dictionary (Collection Object)

<table>
<thead>
<tr>
<th>Properties:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add()</td>
</tr>
<tr>
<td>Remove()</td>
</tr>
<tr>
<td>Clear()</td>
</tr>
<tr>
<td>Contains()</td>
</tr>
</tbody>
</table>

### Public Methods:

- Clear()
2.3 DictionaryBase Properties & Methods

- Since using the DictionaryBase Collection really involves calling methods and properties of the Custom-Collection Object, we simply need to look at the list of properties and methods available to us.
- This collection may be confusing because this Base Class provides an interface to another class IDICTIONARY. It is actually this IDICTIONARY INTERFACE Class that stores the Collection Object or List that stores the elements we need to store, retrieve, remove etc. We saw this in the diagram in previous section.
- With this in mind, realize that a property of the DictionaryBase Collection named Dictionary provides the access to the IDICTIONARY INTERFACE. More on this below.
- The following tables illustrate some of the basic properties and methods of the DictionaryBase Collection:

### Public Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Syntax</th>
<th>Description &amp; Exception Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Count</strong></td>
<td>intVariable = MyBase.Count()</td>
<td>Gets the number of elements actually contained in the DictionaryBase.</td>
</tr>
</tbody>
</table>
- This property returns the actual list or collection instance.  
- **This is the most important property**, because it provides access to the IDICTIONARY INTERFACE thus the Collection itself. This interface contains the actual properties and method we will use to manage the Collection or List object inherited.  
- Therefore, in the examples shown, the Properties and Methods called by the MyBase.Dictionary.XXXX are properties and methods of the IDICTIONARY INTERFACE.  
- This Collection or List object contains the objects we store. **More on this below** |

### Public Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Syntax</th>
<th>Description &amp; Exception Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clear</strong></td>
<td>MyBase.Clear()</td>
<td>Clears the contents of the DictionaryBase instance. Sets the count = 0</td>
</tr>
</tbody>
</table>

### Summary of Properties and Methods of DictionaryBase

- Note that I am not showing any Item property, and methods such as Add(), Remove(), Contains() etc. Reason is that these properties and methods are NOT in the main interface of the DictionaryBase as shown above, but within the DICTIONARY OBJECT is accessed through the property Dictionary shown in the table.
- It is this property Dictionary that will expose the properties and methods or the IDICTIONARY INTERFACE class.
- So, with that said, we need to show the properties and methods of the IDICTIONARY INTERFACE.
2.4 DICTIONARY OBJECT Properties & Methods

- It is the Dictionary Property of the DictionaryBase class that does all the work.
- It contains all the properties and methods to store, retrieve, remove etc., object from the collection.
- Let’s look at some of the most common methods and properties of this class.
- Note that the properties and methods are same as the SORTEDLIST collection class we reviewed prior. So this should be familiar to you at this point.

### Public Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Syntax</th>
<th>Description &amp; Exception Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
<td>Get: objMYObject = MyBase.Dictionary.Item(Key)</td>
<td>Gets or Set or sets the element at the specified index.</td>
</tr>
<tr>
<td></td>
<td>Set: MyBase.Dictionary.Item(Key) = objMYObject</td>
<td>• In GET mode, if the KEY is not found, a NULL is returned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• In SET mode, if the KEY is NOT FOUND, the item is ADDED to the collection instead of modifying. This is very important, when setting, a new item is added if key is not found. Code must manually test for this.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Exceptions raised:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArgumentException – Key is a NULL or NOTHING</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NotSupportedException – In Set Mode, and key does not exists.</td>
</tr>
</tbody>
</table>

### Public Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Syntax</th>
<th>Description &amp; Exception Raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>MyBase.Dictionary.Add(Key, Object)</td>
<td>Adds an object to the IDICTIONARY</td>
</tr>
<tr>
<td></td>
<td>Example 1: MyBase.Dictionary.Add(“111”, objEmpl)</td>
<td>• If key already exist an ArgumentException is raised.</td>
</tr>
<tr>
<td></td>
<td>Example 2: MyBase.Dictionary.Add(objEmpl.ID, objEmpl)</td>
<td>• Exceptions raised:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArgumentException – KEY already exist or DUPLICATE KEY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArgumentException – Key is a NULL or NOTHING</td>
</tr>
<tr>
<td>Clear</td>
<td>MyBase.Dictionary.Clear</td>
<td>Removes all elements from the IDICTIONARY.</td>
</tr>
<tr>
<td>Contains</td>
<td>MyBase.Dictionary.Contains</td>
<td>Determines whether an object or element is in the IDICTIONARY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Return Value: Boolean (True if exists, False otherwise).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Exceptions raised:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ArgumentException – Key is a NULL or NOTHING</td>
</tr>
</tbody>
</table>
| Remove | MyBase.Dictionary.Remove(Key) | Removes the Object at the specified Key from the SortedList.  
- Note that if KEY is NOT FOUND, then nothing is done and the collection stays unchanged. The collection DOES NOT raise an error or does not communicate with the calling program indication key is not found. It is the responsibility of the programmer to put logic to inform that nothing was done since the object was not found.  
- Exceptions raised:  
  ArgumentNullException – Key is a NULL or NOTHING |

### 2.5 Data Type Conversions: CType() or DirectCast()

- This Base Class is of Dictionary Type so it follows the same rule as the SortedList we covered before.
- The objects stored in the collection are of type DICTIONARYENTRY objects. We need to perform data type conversions or Casting when using objects retrieved from the collection.

**Item Property**
- You may need to convert the data type of the object stored in the DICTIONARY.
- **Rule:**
  - If Option Strict is set to OFF, no conversion needed.
  - If Option Strict is set to ON, you must convert the data type of object in collection using CType() or DirectCast()  

**For Each..Next Loop**
- Using For..Each..Next loop in the SortedList is supported.
- **Rule:**
  - **IMPORTANT!** Nevertheless, unlike the ArrayList, even if Option Strict is set to OFF, we need to convert the data type using CType() for a For Each Loop.
  - **IMPORTANT!** Therefore we must use TYPE() conversion when using For-Each whether Option Strict is set to OFF or ON
2.6 DictionaryBase Application Sample Program #1

- In this section we will upgrade the Customer Management Sample Program as follows:
  
  - BEST PRACTICE! Create a Collection Class to encapsulate the Collection Object. We will use the DictionaryBase Collection as the Base Class. We will derive our Collection Class from DictionaryBase and add all the functionality required.
  - We prefer to continue to make the KEY a string. Nevertheless to make our key more flexible, we will use the Object data type for our parameters to methods that represent our KEYS. We will pass a String KEY, but the parameter will be of type Object. Remember that the String Class is derived from Object. So, passing a string to an Object data type is OK.
  - Collection class will save and load itself from a comma-delimited data text file named CustomerData.txt
  - We will implement our Client/Server Architecture by moving all Processing Code from the Module to the Collection Class.
  - We will also continue best practice of setting Option Strict ON.
  - Add a Social Security Property in the Person Class.
  - Make the Person Class is now a MustInherit Base Class
  - Implement Inheritance by deriving a Customer Class from Person.
  - We will also begin to use the Region tag in our code to group similar code and make program more readable.
  - Project requires two user-interface Forms: Main Form & Customer Management Form.
  - To the Customer Management Form, we will add a List box to list the content of all objects stored in the collection.
  - The UI Form Customer Management will load the collection upon display and will save the collection upon close.

- We will continue to keep our application architecture in mind and perform all user interactions in the Form. That is all messages displayed to the user is in the Form.
- This means that we will be force to trap for the exceptions in the Module methods and again in the Form.

Example 2.6 – DictionaryBase Customer Manager with Inheritance Example

Problem statement:
- Upgrade the Using a Customer Object/ArrayList Collection application by replacing the ArrayList with a SortedList Collection.
- Upgrade the application as follows:

Object Model:
- Modify Person Class from previous examples as shown in diagram below
- Create Customer Class and Inherit from Person
- The object model/UML diagram looks as follows:

<table>
<thead>
<tr>
<th>Class MustInherit clsPerson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private data members:</td>
</tr>
<tr>
<td>strName, strSocialSecurity, dBirthDate</td>
</tr>
<tr>
<td>sAddress, sPhone</td>
</tr>
<tr>
<td>Properties</td>
</tr>
<tr>
<td>Get &amp; Set for each data</td>
</tr>
<tr>
<td>Public Constructors</td>
</tr>
<tr>
<td>New()</td>
</tr>
<tr>
<td>New(x,y,z...)</td>
</tr>
<tr>
<td>Public Methods</td>
</tr>
<tr>
<td>Print()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class clsCustomer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherits clsPerson</td>
</tr>
<tr>
<td>Private data members:</td>
</tr>
<tr>
<td>strCustomerID</td>
</tr>
<tr>
<td>Properties</td>
</tr>
<tr>
<td>Get &amp; Set for each data</td>
</tr>
<tr>
<td>Public Constructors</td>
</tr>
<tr>
<td>New()</td>
</tr>
<tr>
<td>New(x,y,z...)</td>
</tr>
<tr>
<td>Public Methods</td>
</tr>
<tr>
<td>Print()</td>
</tr>
</tbody>
</table>
Create Custom-Collection Class derived from `DictionaryBase`.

The object model/UML diagram looks as follows:

**Database/File Access Requirements:**
- We will simulate the database by saving to file. This will be handled by the `Load()` and `Save()` methods of the Collection Class and a text file name `CustomerData.txt`.

**Form Requirements:**
- Add listbox and list button to list the objects in collection.
- In order to comply better with our Application Development Architecture and keep the Form with User Interface code only, we will modify the Form code slightly differently than the previous example to show an alternate method.
Other Requirements:

- Fully implement the three-tiered Client/Server Application Architecture described in class:

  - Presentation Layer/UI
    - User Interface Code Only
  - Business Object Layer (BO)
    - Business Logic
    - Processing Code
    - Validation
    - Data Access Code
  - Database Services Layer
    (Database Management System DBMS)

- Therefore, all UI interaction code should reside in the Presentation layer of Forms. All processing should be done in the Business Objects (Person, Customer & CustomerList classes).
HOW IT’S DONE:

Part I – Create The Application:

Step 1:  Start a new Windows Application project:

Step 2:  Add a Form:

Step 3:  Add a Standard Module:

Step 4:  Set the Project’s properties to behave as a Module-Driven Windows Application:
Business Object Layer – Class Objects

**clsPerson Class:**

**Step 5:** Prepare to Reuse the Person Class from Previous Console Application, by Copying the File from previous Application Folder to the Folder of this Windows Application Project

1. Using Windows Explorer, navigate to the Employees Application folder of the previous example.
2. Copy/Paste the file clsPerson.vb, to this Project folder

**Step 6:** Add the Class to the Project

1. In the Project Menu, select Add Existing Item… and navigate to the project folder
2. Select the *clsPerson.vb* File and click OK
3. The class is now part of the project and ready to be reused!

**Step 7:** Add the following Regions to the Class

![Code Snippet](image-url)
Step 8: Modify the Class as follows:
- Remove ID Number and add Social Security property and data

```vbnet
Option Explicit On
Option Strict On

' Impoted Libraries
Imports System.IO  ' For file access code
Public MustInherit Class clsPerson

#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_strAddress As String
Private m_Phone As String

#End Region
```
Step 9: Remove IDNumber Property and add SocialSecurity Property. Leave the rest as previous example:

```vbnet
#Region "Property Procedures"
'******************************************************************************
'Property Procedures
Public Property Name() As String
    Get
        Return m_Name
    End Get
    Set(ByVal Value As String)
        m_Name = Value
    End Set
End Property

Public Property SocialSecurity() As String
    Get
        Return m_SSNumber
    End Get
    Set(ByVal Value As String)
        m_SSNumber = Value
    End Set
End Property

Public Property BirthDate() As Date
    Get
        Return m_BirthDate
    End Get
    Set(ByVal Value As Date)
        m_BirthDate = Value
    End Set
End Property

Public Property Address() As String
    Get
        Return m_strAddress
    End Get
    Set(ByVal Value As String)
        m_strAddress = Value
    End Set
End Property

Public Property Phone() As String
    Get
        Return m_Phone
    End Get
    Set(ByVal Value As String)
        m_Phone = Value
    End Set
End Property
#End Region
```
Step 10: Modify the Constructors Methods Accordingly:

```vbnet
#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_strAddress = ""
    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 11: Modify Print() Method to Save to an CustomerPrinter File:

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

'Author of base class allows sub classes to override 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_strAddress & ", " & m_Phone)

    'Close StreamWriter Object
    objPrinter.Close()
End Sub
#End Region
End Class
```
clsCustomer Class:

Step 12: Create the clsCustomer Class. It should have the following Regions:

```
Option Explicit On
Option Strict On
'Imported Libraries
Imports System.IO 'For Any File Access requirements

Public Class clsCustomer
    Inherits clsPerson

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

Step 13: Inherit from clsPerson and add the following data member:
Step 14: Add the following properties:

```vbnet
#Region "Property Procedures"
'*********************************************************************
Public Property CustomerID() As String
    Get
        Return m_CustomerID
    End Get
    Set(ByVal Value As String)
        m_CustomerID = Value
    End Set
End Property
#End Region
```

Step 15: Modify the Constructors Methods Accordingly:

```vbnet
#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 Parameterized Constructor
    MyBase.New(strNane, strSSNum, bBDate, strAddress, strPhone)

    'Property Member Initialize data
    Me.CustomerID = strCustomerID
End Sub
#End Region
```
Step 16: Add Print() Method to Save Customer Data to Printer File:

#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)
    'Close StreamWriter Object
    objPrinter.Close()
End Sub
#End Region

#End Region

End Class
clsCustomerList Collection Class:

Step 17: Add a Class to the Project named clsCustomerList. The class will have these Regions:

```vbnet
Option Explicit On
Option Strict On

'Import Libraries
Imports System.Collections
Imports System.IO

Public Class clsCustomerList
    Inherits DictionaryBase
End Class
```

Step 18: Add to Inherit from DictionaryBase Collection, Include the required Libraries and set Option Strict to ONN:

```vbnet
Option Explicit On
Option Strict On

'Import Libraries
Imports System.Collections
Imports System.IO

Public Class clsCustomerList
    Inherits DictionaryBase
```
Step 19: Create the Count Property and Item Property:

```vbnet
#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

#Region "Public Properties Declarations"

'*******************************************************************************
''<summary>
''' Name: Item(Key) Property
''' Purpose: GET or SET the object at the specified key in the Collection
''</summary>
'<'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
```
**Explanation**

- The GET portion of the property returns a `DictionaryEntry` POINTER to the object whose KEY is the parameter:

  - **clsCustomer Pointer**

  - **Dictionary**

    - Properties: Count, Item(key)
    - Methods: Add(key, object), Remove(key), Contains(key), Clear()

  - `DictionaryEntry` POINTERS

  - Copy of these pointers are returned by `Dictionary.Item(key)`

  - Use `CType()` to convert `DictionaryEntry` POINTERS to clsCustomer type

- SET portion replaces the object in the collection

  - **clsCustomer Pointer**

  - **Dictionary**

    - Properties: Count, Item(key)
    - Methods: Add(key, object), Remove(key), Contains(key), Clear()

    - Wrapper Methods: Add(key, clsCustomer), Add(value1, value2, value3), Remove(key), Contains(key), Clear()

    - Regular Methods: Edit(key, clsCustomer), Edit(value1, value2, value3), Print(key), PrintAll(), Load(key), Save()

  - **NEW OBJECT**

  - REPLACED OBJECT

  - No longer has a reference so it is destroyed
Step 21: Create Wrapper Add Method that takes the key and Object:

```csharp
#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)
    Catch objX As ArgumentNullException
        'Step B-Traps argumentNullException when key is Nothing or null
        Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
    Catch objY As ArgumentException
        'Step E-ReThrow an ArgumentException to calling programs
        Throw New System.ArgumentException("Duplicate Key Error: " & objY.Message)
    Catch objE As Exception
        'Step F-Traps for general exceptions.
        Throw New System.Exception("Add Method Error: " & objE.Message)
    End Try
End Sub
```
Step 22: Create an OVERLOADED Wrapper Add Method that takes the individual values:

```
'***********************************************
''<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="strName"></param>
''<param name="strSN"></param>
''<param name="dBDate"></param>
''<param name="strAddress"></param>
''<param name="strPhone"></param>
''<param name="strCustID"></param>
''<remarks></remarks>
Public Sub Add(ByVal strName As String, ByVal strSN As String, _
    ByVal dBDate As Date, ByVal strAddress As String, _
    ByVal strPhone As String, ByVal strCustID 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 = strSN
        .BirthDate = dBDate
        .Address = strAddress
        .Phone = strPhone
        .CustomerID = strCustID
    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 argumentNullException when key is Nothing or null
    Catch objX As ArgumentNullException
        'Step C-ReThrow ArgumentNullException
        Throw New System.ArgumentNullException("Invalid Key Error: " & objX.Message)
    End Catch

    'Step D-Traps for ArgumentException when KEY is duplicate.
    Catch objY As ArgumentException
        'Step E-ReThrow an ArgumentException to calling programs
        Throw New System.ArgumentException("Duplicate Key Error: " & objY.Message)
    End Catch

    '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 Catch

    End Try
End Sub
```
Step 23: Create the Remove Wrapper method:

```vbnet
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
    End Try
End Function
```

Step 24: Create the Clear Wrapper method:

```vbnet
Public Shadows Sub Clear()
    'Step A- Begin Error trapping
    Try
        'Step 1-Calls Collection.Clear() Method
        MyBase.Dictionary.Clear()
    Catch objx As Exception
        'Step B-Traps for General exceptions
    End Try
End Sub
```

#End Region
Step 25: Create the Edit method. This method takes key and object as arguments:

```vbnet
#Region "Public Regular Methods Declarations"

'**********************************************************************************
''<summary>
''<param name="key"></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 an ArgumentNullException
  Throw New SystemArgumentNullException("Invalid Key Error: " & objX.Message)
'Step D- Traps for general exceptions.
Catch objE As Exception
  'Step E- Throw an general exceptions
  Throw New SystemException("EditItem Error: " & objE.Message)
End Try
End Function
```

```
Dictionary
keys

```

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;444&quot;</td>
<td>Joe</td>
</tr>
<tr>
<td>&quot;222&quot;</td>
<td>Angel</td>
</tr>
<tr>
<td>&quot;111&quot;</td>
<td>Mary</td>
</tr>
</tbody>
</table>

```

Properties:
- Count
- Item(key)

Methods:
- Add(key, object)
- Remove(key)
- Contains(key)
- Clear()

```

Properties:
- Count
- Item(key)

Wrapper Methods:
- Add(key, clsCustomer)
- Add(value1, value2, value3)
- Remove(key)
- Contains(key)
- Clear()

Regular Methods:
- Edit(key, clsCustomer)
- Edit(value1, value2, value3)
- Print(key)
- PrintAll()
- Load(key)
- Save()
**Step 26: Create an OVERLOADED Edit method. This method takes individual values as arguments:**

```vbnet
'******************************************************************************************
'** Name: Function OVERLOADED Edit(value1, value2, etc.)
'** Purpose: Sets or MODIFIES object located at specified key in the Collection
'**</summary>
'**<param name="strName"></param>
'**<param name="strSSNum"></param>
'**<param name="dBDate"></param>
'**<param name="strAddress"></param>
'**<param name="strPhone"></param>
'**<param name="strCustomerID"></param>
'**<returns></returns>
'**<remarks></remarks>
Public Function Edit(ByVal strName As String, ByVal strSSNum As String, _
ByVal dBDate As Date, ByVal strAddress As String, _
ByVal strPhone As String, ByVal strCustomerID 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.
'NOTE THAT SINCE THE ID NUMBER OF THE CUSTOMER IS THE KEY, WE DO NOT
'WANT TO MODIFY IT OR TAMPER WITH IT IN ANY WAY
With objItem
    .Name = strName
    .SocialSecurity = strSSNum
    .BirthDate = dBDate
    .Address = strAddress
    .Phone = strPhone
End With

'Step 6-Return True since found and modified
Return True
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
```
**Explanation**

- Calling `CType(Dictionary.Item(key))` returns a CONVERTED DICTIONARY to CUSTOMER POINTER which points to the object in the collection. This POINTER gets assigned to `objItem`, so now `objItem` points to the object in the collection.
- What we do to `objItem` we do to the object inside the collection. Such as call properties (GET & SET) or call methods (PRINT(), PRINTALL(), LOAD(), SAVE() ETC.)

- `objItem` points to the object in the collection.
- At this point, you can SET, GET OR CALL METHODS of the object inside the collection.

The following three methods we will create: Print(key), PrintAll, and Save() will utilize this `objItem` POINTER to call methods inside the object residing in the collection.

Use this diagram to help you understand how these methods work.
Step 27: Create Print() Method as follows:

```vbc
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 ArgumentNullException when key is Nothing or null.
    Catch objX As ArgumentNullException
        'Step C-Throw Collection ArgumentNullException
        Throw New SystemArgumentNullException("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("PrintCustomer Error: " & objE.Message)
    End Try
End Function
```
Step 28: Create PrintAll() Method as follows:

```vbnet
Private Sub PrintAll()
    'Step A - Begin Error trapping
    Try
        '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
```
Step 29: Create Load() Method as follows:

Explanation
- Using the File Class we test to make sure the file exist before attempting to open it. If it does not exist we create it.
- We then open the file for reading using the StreamReader Class, loop and read a line, parse the line using Split() function and extract the data from the array.
- The Wrapper Add method is used to add the object to the collection.

```vbnet
'**********************************************************************************
''<summary>
''' Name: Load Sub Procedure
''' Purpose: Read Data.txt file, recreates objects and populates the collection
'''</summary>
''<remarks></remarks>
Public Sub Load()
    'Step A- Begin Error trapping
    Try
        'Step 1-Create necessary variables
        Dim strLine As String
        '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
            strLine = objDataFile.ReadLine
            'Step 6-Parse the line using VB Split() & assign to array
            Dim tempArray() As String = Split(strLine, ",")
            'Step 7-Call add to add object to Collection
            Add(tempArray(1), tempArray(2), CDate(tempArray(3)), tempArray(4), tempArray(5), tempArray(0))
        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 Sub
```
Step 30: Create Save() Method as follows:

Explanation
- The data file is open using StreamWriter class with options to overwrite the file.
- We iterate through the collection using a For Each loop and GET the property of each object and write it to file.

```vbnet
'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)
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 Sub
#End Region
End Class
```

- Note how all processing is done inside the classes, inside the Collection class is were all additions, editing and modifying of data is done. All processing, Business Logic & Validation should be in the class. Note that we are not implementing validation at this point.
- All processing is done in the Business Object layer!
Presentation Layer (UI) – Module & Forms

Module – User-Interface Code only. No Business processing!

Step 29: In the Module Add the Following Code:

- Code any Global & Private Variable declarations and Sub Main()
  1. **Option Strict** = ON.
  2. Import the System.Collections Library
  3. Create a Custom-Collection Class Object
  4. Declare Global Customer Form Object

```vbnet
Option Explicit On
Option Strict On

Module modMainModule

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

Dim objCustomerForm As frmCustomerForm = New frmCustomerForm()

End Sub
```

Step 30: Sub Main:

```vbnet
'********************************************************************************
''' <summary>
''' Name: Main()
''' Main routine that controls flow of program
''' Step 1-Calls the InitializeList() method to perform any required Initialization
''' Step 2-Calls Customer Form object's ShowDialog() method to display itself
''' </summary>
''' <remarks></remarks>
Public Sub Main()

InitializeList()

'Display Customer Form
objCustomerForm.ShowDialog()

End Sub
```
**Step 31: InitializeList Method:**

```vbnet
Public Sub InitializeList()
    'No objects are added to Customer Collection from initialize
    '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 an Exception.

End Sub
```

---

**Brief Discussion of Module Code**

- Note how the Module which is also part of the UI has no processing code.
- All processing is done in the Business Object layer
- So far we have achieved our Application architecture objectives, to separate User-Interface from Processing or implementation!
Main Form:

Overview
- The main form is the main portal for navigating to the Customer Management Form.
- The form provides the required controls & code for navigating to the other form.

Step 32: User-Interface and Controls:

Step 33: Main Form Code:

```vbnet
Option Explicit On
Public Class frmMain
    ''' <summary>
    ''' Displays Management Forms
    ''' </summary>
    Private Sub btnEMangement_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnEMangement.Click
        Dim objEM As New frmCustomerForm
        'Step 1-Display Modal Customers Form
        objEM.ShowDialog()

        'OPTIONAL METHOD TO DISPLAY CUSTOMER FORM WITH MAIN FORM HIDING:
        'Step 1-Hide yourself
        'Me.Hide()

        'Step 2-Display Modal Customers Form
        'objEM.ShowDialog()

        'Step 3-Show yourself after Customer Form closes
        'Me.Show()
    End Sub

    ''' <summary>
    ''' Exits the Main Form
    ''' </summary>
    Private Sub btnExit_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnExit.Click
        Me.Close()
    End Sub
End Class
```
Customer Management Form:

Overview
- The job of the Form or **User-Interface Layer** is to interact with the User. Received request from the user and pass it down to the **Business Objects Layer** for processing. On the other hand, it also receives results from the **Business Object Layer** and displays it to the user. User-Interface code only!!!
- We will add Try/Catch Block to the Form in order to trap the errors generated by the Method that manage the Collection in the Module.

Step 32: User-Interface and Controls:

Step 33: In the Form frmCustomerForm Create Temp object POINTER:

```vbnet
Option Explicit On
Option Strict On

Public Class frmCustomerForm
    Inherits System.Windows.Forms.Form

    'Declare Form Level POINTER
    Private objCustomer As clsCustomer
```
Step 34: In Form_Load Event-Handler, Populate the Collection from file:

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

 'Step A-Begins Exception handling.
 Try

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

 'Step B-Traps for general exceptions.
 Catch objE As Exception
 'Step C-Inform User
 MessageBox.Show(objE.Message)
 End Try
 End Sub
```

Step 35: Form_Close Event-Handler:

```vbnet
Private Sub frmCustomerForm_Closed(ByVal sender As Object, ByVal e As System.EventArgs) Handles MyBase.Closed

 'Step A-Begins Exception 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 general exceptions.
 Catch objE As Exception
 'Step C-Inform User
 MessageBox.Show(objE.Message)
 End Try
 End Sub
```
Step 36: EXIT Button Click Event:

```vbnet
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 37: Add Code and Exception Handling to GetCustomer Click Event:

```vbnet
'************************************************************************************
''' <summary>
''' Name: Event-Handler for btnGetCustomer button
''' Purpose: To retrieve an POINTER TO object from the collection base on ID or Key
''' </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 Exception handling.
    Try
        'Step 1-Call Calls Collection.Item() Property to return pointer to object
        'in Collection
        objCustomer = objCustomerList.Item(txtIDNumber.Text)
        '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 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 38: Add Code and Exception handling to the Add_Click Event-handler:

```vbnet
'************************************************************************************
''' <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(txtName.Text.Trim, txtSSNum.Text.Trim, _
        CDate(txtBirthDate.Text), txtAddress.Text.Trim, txtPhone.Text.Trim, _
        txtIDNumber.Text.Trim)
        '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 ArgumentException when KEY is duplicate.
        Catch objY As ArgumentException
            'Step E-Inform User
            MessageBox.Show(objY.Message)
            'Step F-Traps for general exceptions.
        Catch objE As Exception
            'Step G-Inform User
            MessageBox.Show(objE.Message)
    End Try
End Sub
```
Step 39: Add Code and Exception handling to EditCustomer Event:

```vbnet
'************************************************************************************
'''' <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
        '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 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 40: Add Code and Exception Handling to Delete_Click Event:

```vbnet
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
        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 ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step C- Inform User
            MessageBox.Show(objX.Message)
        End Catch

        'Step D- Traps for general exceptions.
        Catch objE As Exception
            'Step E- Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```
Step 41: Add Code and Exception Handling to Print Click Event:

```vbnet
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 ArgumentNullException when key is Nothing or null.
        Catch objX As ArgumentNullException
            'Step C-Inform User
            MessageBox.Show(objX.Message)
        End Catch

        'Step D-Traps for general exceptions.
        Catch objE As Exception
            'Step E-Inform User
            MessageBox.Show(objE.Message)
        End Try
    End Sub
```
Step 40: Add Code and Exception Handling to the PrintAll Click Event:

```vbnet
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()
    Catch objE As Exception
        'Step C- Inform User
        MessageBox.Show(objE.Message)
    End Try
End Sub
```
Step 41: Add Code and Exception Handling to the List Click Event:

```vbnet
'*******************************************************************************
''<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 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, 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 general exceptions.
    Catch objE As Exception
        'Step C-Inform User
        MessageBox.Show(objE.Message)
    End Try
  End Sub
End Class
```
Part IV – Output & Summary

Summary

- Run the program and you can then perform the necessary operations on the list.
- Also you can purchase and print customer information to file.
- The main point to this example is that we moved all procession into the Business Object Layer and we kept all user interaction code in the User Interface Layer.

Form Output:
File Output:

Data File:

```
333, Sam Franks, 333-22-3333, 3/12/1967, 333 Jay Street, 718 260-5333
111, Joe Smith, 111-11-1111, 1/23/1971, 333 Jay Street, 718 260-5000
222, Angel Rod, 222-22-2222, 3/12/1967, 222 Jay Street, 718 260-5000
```

Printer File:

```
Joe Smith, 111-11-1111, 1/23/1971, 333 Jay Street, 718 260-5000, 111
Joe Smith, 111-11-1111, 1/23/1971, 333 Jay Street, 718 260-5000, 111
Angel Rod, 222-22-2222, 3/12/1967, 222 Jay Street, 718 260-5000, 222
```