

# **CS708 Lecture Notes**

## **Visual Basic.NET Programming**

### **Object-Oriented Programming**

#### **Collections Review**

**(Part I of II)**

**(Lecture Notes 2A)**

Prof. Abel Angel Rodriguez

<b>CHAPTER 1</b>	<b>SYSTEM.COLLECTIONS NAMESPACE</b>	<b>3</b>
<b>1.0</b>	<b>Introduction to the System.Collections Namespace</b>	<b>3</b>
1.1	System.Collections Overview	3
1.2	Using Collections in a Nut Shell	3
1.3	The Concept of Key and Index	5
1.4	Data Type Conversions using Collections	6
1.5	The For..Each..Next Loop	7
1.6	Collections & Exception Handling	8
1.6	The System.Collection NameSpace	9
<b>2.0</b>	<b>ArrayList</b>	<b>11</b>
2.1	ArrayList Overview	11
2.2	ArrayList Properties & Methods	11
2.3	Using For..Each..Next Loop in ArrayList	12
2.4	ArrayList Application Sample Program #1	13
	Example 2.1 - Module-Driven Window Application – ArrayList & Customer Objects Example	13

# Chapter 1 System.Collections Namespace

## 1.0 Introduction to the System.Collections Namespace

### 1.1 System.Collections Overview

- ❑ Collections are a list of classes available in the System Namespace to handle a list of Objects without having to use arrays.
- ❑ Arrays are used to store a list of items of a specific VB data type, such as integer, string, date, etc.
- ❑ Arrays can also be used to store a list of user-defined Objects or Objects we create. This is known as an **array of Objects**. Using arrays is a powerful and efficient way to store Objects. We have used arrays of object in our examples and projects, but such implementation can result in messy, less maintainable code when coding real-world applications that usually require that the array size varied and not have a fixed size limitation. Even if we are using dynamic arrays, we would need to use and keep track of the size of the array in order to make the array dynamic. In short array programming can be complex.
- ❑ Visual Basic.NET provides a better way to store a list of Objects without the use of arrays. This mechanism is called a **Collection**.
- ❑ The .NET framework namespace offers the **System.Collections** library which contains a number of Collection Classes we can use.
- ❑ The **System.Collection Classes** are build-in classes that are ready for you to use and each Collection Class provides built-in *Properties & Methods* (procedures/functions), which allows you **store, delete, search** or organize the Objects on the list.
- ❑ The items or members of a collection don't have to be of the same data type unlike arrays.
- ❑ The Collection Classes are a useful alternative to storing Objects in arrays since they provide a mechanism for writing clean and maintainable code. But keep in mind that there is a price paid to use collections, and that is the overhead required.
- ❑ Before I list the some of the various Collection Classes available in the library, I would like to first review and discuss few concepts.

### 1.2 Using Collections in a Nut Shell

#### Collection Object

- ❑ Since the Collections in the Visual Basic.NET Library are classes, as with any class we follow are three step process to creating OOP programs, and that is:

- I. **Create the Class specification**
  - Private Data/ Properties/Methods
- II. **Create Object of the Class**
- III. **Use the Object of the Class**
  - Write the program to manipulate, access or modify the properties & Call Methods

- ❑ But since the Classes are already created by Microsoft or within VB.NET, we don't have to perform the first step, but only the last two as follows:

- I. **Create Object of the Collection Class from the library (Collection Object)**
- II. **Use the Object of the Class**
  - Write the program to manipulate, access or modify the properties & Call Methods

- ❑ Note that Creating an Object of the *System.Collection* Classes is known as a **Collection Object** or Object of the Collection.

## Custom Collection Class and Inheritance (**BEST PRACTICE**)

### Strong-Typed Collection

- ❑ Creating an Object of the *System.Collection* Classes or **Collection Object** is simple and common, but there are some disadvantages to this. In an application the Collection Object contains the data, but this data is now accessible to the entire program, which can cause a security risk or can be corrupted/deleted by unauthorized code.
- ❑ **Another and most important issue is that a Collection Object will accept any type of data. That is if we create a Collection Object to store say Employee Objects, there is nothing to stop unauthorized code from adding say a VideoTape Object into the collection or a date or integer etc.**
- ❑ A better implementation is to encapsulate the *Collection Object* inside a custom Class that we create. In other words, make the Collection Object a private member to a custom class that we create to protect the Collection Object. This is known as a **Strong-Typed Collection**.
- ❑ These custom classes that protect the collection are called **Custom Collection Classes**
- ❑ Don't get the terminology confused, the **Collection Object** is an object created from the *System.Collections* namespace, while the **Custom Collection Class** is a class we create to encapsulate the *Collection Object*.

### Implementing Custom Collection Classes using Inheritance

- ❑ Enclosing the **Collection Object** inside a class or **Collection Class** is a good idea. VB.NET has the capabilities of inheritance, you can simply derive a Collection Class or Sub-Class directly from the *System.Collections* Namespace.
- ❑ This class that we inherit, will inherit all the power and capability (properties/methods) of the Base class in the library. In addition, we can Overload and Override any of the properties or methods.
- ❑ Therefore we can also have the following steps added to the Collection Class:

- I. **Derive a Sub-Class from the System.Collection Namespace (Custom Collection Class)**
  - Properties/Methods
    1. **Call the inherited Properties/Methods. Overload/Override properties/methods as necessary.**
- II. **Create Object of the Class (Collection Class Object)**
- III. **Use the Object of the Class**
  - Write the program to manipulate, access or modify the properties & Call Methods

### Summary

- ❑ We can use the *System.Collections* class library in our OOP programs using any of the following three methods:
  1. Create **Collection Objects** from the *System.Collections* and use the Object by calling properties/methods of the Collection Object
  2. Create **Collection Classes**, create and use a Collection Object from the *System.Collections* as a private member. Create Object of the Class (Collection Class Object), use the object by calling properties/methods of the objects
  3. (**Best Practice!**) **Inherit Collection Class** from the *System.Collections*, Create Object of the Class (Collection Class Object), use the object by calling properties/methods of the objects

### 1.3 The Concept of Key and Index

- ❑ Before learning how to use the Collections, we need to understand the concept of **Keys** and **Index**. These mechanisms are needed in order to be able to *add*, *delete* or *retrieve* data from the Collection.

#### Keys

- ❑ Some collection store **Key/Item** pairs. That is the object you add will have an associated key.
- ❑ The **KEY** is a **UNIQUE** value that you define and is used to identify and associate each item on the list when the item is added to the list
- ❑ What that means is that when you add an item to the list with a given *key*, the *key* is the unique value used to identify the object in the list
- ❑ In .NET, the *key* is of type **Object**, as you recall, the Object data type is an internal data type of VB.NET. All classes created are decedent of Object, including the **String Class**.
- ❑ So a Key can be of type Object, but most popular is of type **string**.
- ❑ The *key* value must be a **unique string**. It can be a name, a driver's license number, a social security number
- ❑ Remember, that the key should be string; if you decide to use an **Integer** value or some other value, you must convert it to a **String** or type **Object** first, and then use it as a key.
- ❑ In **OOP**, a common use of *keys* is to use a unique data value of the Object you plan to add to the list, as the key that is used to store and identify that object. For example if we write a **Customer** class, and we create a **Customer Object**, the *key* can be the **Object's CustomerID** or **AccountNumber** data member, which are usually unique for each Customer.

#### Index

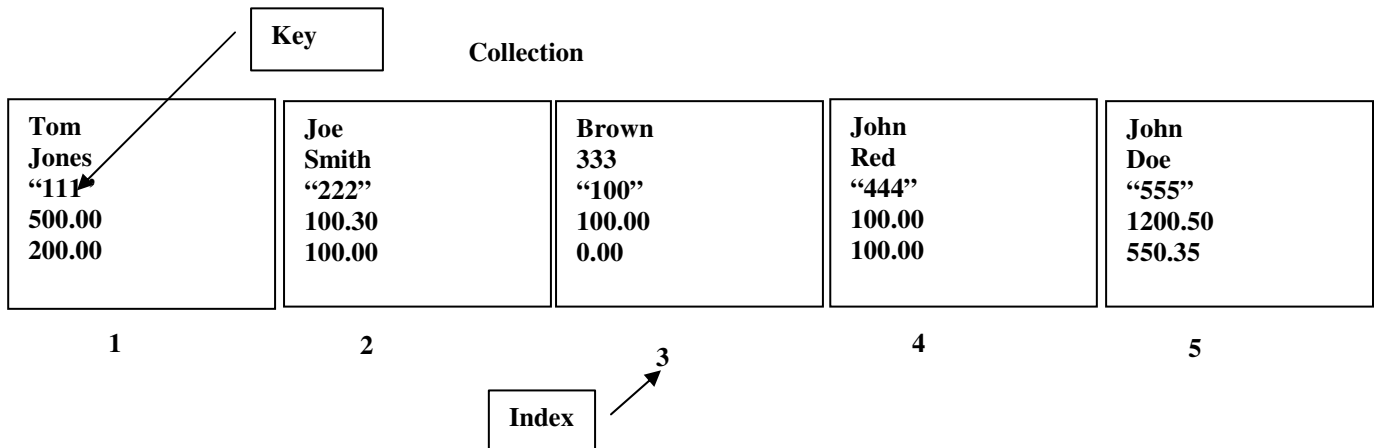
- ❑ An **Index** is an **integer** value used which identifies a cell on the list. This is similar to the index in an array.
- ❑ **Index** are usually used to iterate or search through the Collection without having to worry about the object or key.

#### Zero-Based and One-Based Collections

- ❑ If the Collection starts with an *index* = 0, than the collection is a **Zero-Based** Collection.
- ❑ If the Collection starts with an *index* = 1, than the collection is a **One-Based** Collection

#### Summary

- ❑ Some Collections in the namespace are based on Keys or Index or a combination of both.
- ❑ A key must be string and an index an integer.
- ❑ In a collection that stores keys, you can use a unique string as the key, or use a unique string within the object as a key as well.
- ❑ The figure below illustrates the concept of keys & Index. It assumes a list that stores Bank Customer Objects. Each Object has a name, Customer ID (Used as the key when objects were added), savings & checking accounts balances:



## 1.4 Data Type Conversions using Collections

- ❑ Objects stored in collections are converted and stored using a different class type than that of the original object being stored.
- ❑ That is if you store Customer object in a collections, it is converted and stored as a different type inside the collection.
- ❑ For example, collections that stored items via *index*, are known as **ILIST** collections and stored the objects as type *Object*, which is an internal storage class in VB.NET
- ❑ For example there are collections that store items via *key*, these collections are know as **DICTIONARY** type collections and store objects as type of the class *DictionaryEntry*.
- ❑ What this means is that in some cases you cannot use the objects returned from the collection as is, you need to convert them back to the data type in which they were originally.
- ❑ You can use the **CType()** or **DirectCast** Functions to perform the conversion:

```
MyNewDatatype = CType(ObjectInCollection, ClassType)
```

```
MyNewDatatype = DirectCast(ObjectInCollection, ClassType)
```

- ❑ For example:
  - For example, assuming we like to convert an object of the type **Object** to type **clsCustomer**:

```
Dim objAnObject As New Object
Dim objCustomer As New clsCustomer

'Convert DictionaryEntry object to type Customer using CType()
objCustomer = CType(objAnObject, clsCustomer)

'Convert DictionaryEntry object to type Customer using DirectCast()
objCustomer = DirectCast(objAnObject, clsCustomer)
```

- ❑ **CType()** and **DirectCast()** perform the same operations. **DirectCast()** can only be used when there is an Inheritance relationship exists between the source (object to convert) and destination (converted object)
- ❑ **DirectCast()** is faster and has better performance when called from a loop that executes a very large number of times.

### Data Conversion & Option Strict ON/OFF

- ❑ When using Collections, this conversion is NOT necessary in most cases. How do we know when to do this?
- ❑ That depends if we have in our code Option Strict set to ON or OFF:

**Option Strict On**

**Option Strict Off**

- ❑ The rules for most of the **INDEX** or **LIST** based Collections is as follows:
  - **Option Strict ON** – If we have Option Strict set to **ON**, we need to convert the object store in the collection every time we access the object stored in the collection.
  - **Option Strict OFF** – If we have Option Strict set to **OFF**, we DON'T need to convert the object store in the collection every time we access the object.
- ❑ The rules for most of the **KEY** or **DICTIONARY** based Collections is as follows:
  - **Option Strict ON** – If we have Option Strict set to **ON**, we need to convert the object store in the collection every time we access the object stored in the collection.

- **Option Strict OFF** – If we have Option Strict set to **OFF**, we DON'T need to convert the object store in the collection every time we access the object, with the exception of the FOR-EACH LOOP! For these types of Collections, we need to convert using *CType()* when using the FOR-EACH LOOP.

□ We will see examples of this in later lectures.

## 1.5 The For..Each..Next Loop

### Introduction

- The **For ..Each..Next** is a loop supported by the Collections Classes in the Collections library.
- The **For ..Each..Next** loop allows you to iterate through a loop without having to know how many objects are in the Collection. In other words using a **For ..Each..Next** loop you don't need to use the collection *Count* or *Item* properties in order to iterate through the loop like you would have to do in a regular **For..Next..Loop**
- This **For ..Each..Next** loop is a more efficient way of iterating through a collection than using a standard **For..Next**.
- The syntax to the **For Each...Next** loop is as follows:

**For Each** *Object* **In** *Collection*

[*Statements*]

[**Exit For**]

[*Statements*]

**Next** [*Object*]

- *Object* – Object variable that will store the object retrieved by the **For ..Each..Next** loop.
- *Collection* – The Collection Object you are searching
- *Statements* – VB Code the will manipulate the *Object* returned by the **For ..Each..Next** loop.

### How does the For...Each...Next Loop Work

- The **For Each...Next** loop is entered if there is at least one element in the *Collection*. This is important to keep in mind for troubleshooting. If the **For Each..Next** Statement does not run, it is because the *Collection* is empty.
- Once the loop has been entered, the statements are executed for the first element in the *Collection*. A copy of this element is assigned to an object which you must create.
- If there are more elements in the *Collection*, the statements in the loop continue to execute for each element.
- When there are no more elements, the loop is terminated and execution continues with the statement following the **Next** statement

### Using the For...Each...Next Loop

- To use the **For Each...Next** loop, create an object or element that will hold the value returned by the loop, and apply what ever operation you desire on this object
- For example:
  - For example, assuming we like to retrieve the elements of the Collection *colCustomerList* using the **For..Each..Next** the syntax will be as follows:

```
Dim objCustomer As New clsCustomer
...
For Each objCustomer In colCustomerList

    'do here what ever you want, retrieve/set object, property, invoke Methods
    etc.
    objCustomer.Print ()
    .....

Next objCustomer
```

## Data Type Conversions when using For...Each...Next Loop (Important)

- ❑ As stated earlier, for some Collections, the objects stored in these collections may be converted and stored using a different class type than that of the original object.
- ❑ You may need to convert these objects using the CType() method back to the original data type when retrieving from Collections.
- ❑ This also applies to using the *For..Each..Next* loop. The object used as part of the syntax to retrieve the data in the **For Each...Next** loop must be of the data type used to stored the object in the collection.
- ❑ So this object needs to be declared of the type used to stored object in the collection.
- ❑ For example:
  - For example, assuming we like to retrieve the elements of a IList Type Collection which stores objects using the *Object* class from the Collection *colCustomerList* using the **For..Each..Next** the syntax will be as follows:

```
'Object needed for storing objects returned from collection
Dim objTempObject As New Object ()

'Object needed for using objects returned from collection
Dim objCustomer As New clsCustomer
...
For Each objTempObject In colCustomerList

'Convert data type
objCustomer = CType(objTempObject, clsCustomer)

'do here what ever you want, retrieve/set object, property, invoke Methods etc.
objCustomer.Print()
.....
Next objCustomer
```

## 1.6 Collections & Exception Handling

### General Errors

- ❑ The Collection also generate exceptions when an error occurs
- ❑ As always when working with exceptions, it is the responsibility of the programmer to trap for these exceptions when using a collection.

### Collection Specific Errors

- ❑ Unfortunately in most cases when you attempt to do something with a collection and the operations is not supported, or the Collection sends us a response or message, this response is in the form of an *Exception*.
- ❑ Let me explain this further, for example if we search for an object inside a collection, and it does not exist in the collection, instead of the collection returning a 0 or 1 or Nothing, it simply throws an exception. This exception is NOT a general exception but specific type of exception listed in the Collection Class description.
- ❑ For example, lets look at some exceptions generated by collections:
  - [NotSupportedException](#) – When array is read-only
  - [ArgumentOutOfRangeException](#) – When index = -1
  - [ArgumentException](#) – KEY already exist or DUPLICATE KEY
  - [ArgumentNullException](#) – Key is a NULL or NOTHING
- ❑ **Again, it is the job of the programmer to trap for these exceptions. The programmer needs to read the class descriptions where these exceptions are listed and handle them accordingly!**



## 1.6 The System.Collection NameSpace

- ❑ Now that we have a basic understanding of the basic concepts lets look at some of the Collections offered in the namespace.
- ❑ The VB.NET System.Collection Namespace contains quite a number of Collections available for many uses. Since our time is limited in this course we will only show and cover a few.
- ❑ The following table is a snapshot of a few of the Collection Classes available in the **System.Collection** Namespace:

Class	Description
<a href="#">ArrayList</a>	<ul style="list-style-type: none"> <li>▪ Implements a LIST similar to using an array whose size is dynamically increased as required.</li> <li>▪ Elements in the list are stored based on an integer <b>index</b> value, just like a standard array.</li> <li>▪ You can simply create object from these class and use the <i>properties</i> and <i>methods</i>.</li> </ul>
<a href="#">CollectionBase</a>	<ul style="list-style-type: none"> <li>▪ Provides an abstract <b>MustInherit</b> <u>base class</u> for a strongly typed collection.</li> <li>▪ Implements a LIST base on <b>index</b> as with ArrayList.</li> <li>▪ This is a <b>MustInherit</b> 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.</li> </ul>
<a href="#">SortedList</a>	<ul style="list-style-type: none"> <li>▪ Implements a LIST whose size is dynamically increased, but is based on a collection of a string key-and-value pairs.</li> <li>▪ This Collection is sorted by the keys and is accessible by <b>key</b> and by <b>index</b>.</li> <li>▪ Note that you must manipulate this Collection via the string Key, but you can access the objects via the <b>key</b> or <b>index</b>.</li> <li>▪ You can simply create object from these class and use the <i>properties</i> and <i>methods</i>.</li> <li>▪ This Collection is of the type <b>Dictionary</b> that use Key-Value pair, which means that the <b>DictionaryEntry</b> object is used to stored the objects inside the collection</li> <li>▪ In some cases, when using the properties and methods, type conversion is required using the CType() method.</li> </ul>
<a href="#">DictionaryBase</a>	<ul style="list-style-type: none"> <li>▪ Provides an abstract <b>MustInherit</b> <u>base class</u> for a strongly typed collection.</li> <li>▪ Implements a LIST base on <b>key</b> as with the <b>SortedList</b>.</li> <li>▪ This is a <b>MustInherit</b> 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.</li> <li>▪ This Collection is of the type <b>Dictionary</b> that use Key-Value pair, which means that the <b>DictionaryEntry</b> object is used to stored the objects inside the collection</li> <li>▪ In some cases, when using the properties and methods, type conversion is required using the CType() method.</li> </ul>
<a href="#">Hashtable</a>	<ul style="list-style-type: none"> <li>▪ Represents a collection of key-and-value pairs that are organized based on the hash code of the key.</li> <li>▪ Implements a LIST base on <b>key</b> as with the <b>SortedList</b> but the key must be generated base on a Hast Code.</li> <li>▪ You can simply create object from these class and use the <i>properties</i> and <i>methods</i>.</li> </ul>
<a href="#">Queue</a>	<ul style="list-style-type: none"> <li>▪ Represents a first-in, first-out collection of objects.</li> </ul>
<a href="#">Stack</a>	<ul style="list-style-type: none"> <li>▪ Represents a simple last-in-first-out collection of objects.</li> </ul>

## Selecting a Collection Class

- ❑ Choose a Collection Class carefully, otherwise choosing the wrong collection can restrict your use of the collection or make programming the application more difficult than necessary.
- ❑ Consider the following guidelines:
  1. Do you need to **add**, **delete** or manage elements by index?
    - **ArrayList**
    - **Collection Base**
    - Others not listed in this document
  2. Do you need to **add**, **delete** or manage elements by key?
    - **SortedList**
    - **Dictionary Base**
    - **HashTable**
    - Others not listed in this document
  3. Do you need to access elements by index? The following Collections support access by index:
    - **ArrayList**
    - **Collection Base**
    - **SortedList** – Access only
    - **Dictionary Base**– Access only
    - Others not listed in this document
  4. Do you need to **sort** the elements differently from how they were entered?
    - **SortedList**
    - **Dictionary Base**
    - **HashTable**
    - Others not listed in this document
  5. Do you need fast search?
    - **SortedList**
    - **Dictionary Base**
    - Others not listed in this document
  6. Do you need a sequential list where the elements can be discarded after its value is retrieved?
    - Use **Queue**
    - Use **Stack**
  7. Do you need to access elements in a certain order, such as first-in-first-out or last-in-last-out or randomly?
    - Use **Queue**
    - Use **Stack**

## Exceptions in Collection Classes

- ❑ Unfortunately when a Collection cannot find an object in the collection the collection does not return a value but simply raises exceptions.
- ❑ Also when something goes wrong with the usage of a collection an exception is raised.
- ❑ Therefore it is your responsibility to add Try/Catch statements to the code when using collection to trap these exceptions and take the appropriate action or display any messages to the user.
- ❑ Consider the following guidelines:

## 2.0 ArrayList

### 2.1 ArrayList Overview

- The ArrayList Class is an index based Collection. It has the following description:

Class	Description
<a href="#">ArrayList</a>	<ul style="list-style-type: none"><li>▪ Implements a LIST similar to using an array whose size is dynamically increased as required.</li><li>▪ Elements in the list are stored based on an integer <b>index</b> value, just like a standard array.</li><li>▪ You can simply create object from these class and use the <i>properties</i> and <i>methods</i>.</li></ul>

- Use an *ArrayList* when the objects we need to store can be access, added, deleted based on an **index**.
- Keep the following information in mind when working with *ArrayLists*:
  - In an application, knowledge of the index must be known or obtained when searching for objects. If the index is readily available when we need to search, than the *ArrayList* is a good choice for implementing the application.
  - On the other hand, if the index is not available, then we need to write more code to obtain the index before finding the object we seek.
  - If the application requires retrieval of the objects base based on another value that is NOT the index, than the *ArrayList* is NOT a good choice, choose another Collection such as *SortedList*.

### 2.2 ArrayList Properties & Methods

- Since using the ArrayList 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 ArrayList Collection:

#### Public Properties

Property	Syntax	Description & Exception Raised
<a href="#">Capacity</a>	<code>colObject.Capacity</code>	Returns or gets or sets the number of elements that the <b>ArrayList</b> can contain at startup.
<a href="#">Count</a>	<code>colObject.Count</code>	Gets the number of elements actually contained in the <b>ArrayList</b> .
<a href="#">Item</a>	Get: <code>Object = colObject.Item(index)</code>  Set: <code>colObject.Item(index) = Object</code>	Gets or sets the element at the specified index.  <b>Exceptions raised:</b> <a href="#">ArgumentOutOfRangeException</a> – when index is negative or less than Count Property

## Public Methods

Method	Syntax	Description & Exception Raised
<a href="#">Add</a>	<code>colObject.Add(Object)</code>	Adds an object to the end of the <b>ArrayList</b> <ul style="list-style-type: none"> <li>▪ <b>Return Value:</b> index of location where object inserted.</li> <li>▪ <b>Exceptions raised:</b> <a href="#">NotSupportedException</a> – When array is read-only</li> </ul>
<a href="#">Clear</a>	<code>colObject.Clear</code>	Removes all elements from the <b>ArrayList</b> .
<a href="#">Contains</a>	<code>colObject.Contains(Object)</code>	Determines whether an element is in the <b>ArrayList</b> . <ul style="list-style-type: none"> <li>▪ <b>Return Value:</b> Boolean (True if exists, False otherwise).</li> </ul>
<a href="#">IndexOf</a>	<code>colObject.IndexOf(Object)</code>	Returns the zero-based <i>index</i> of the first occurrence of the object in the <b>ArrayList</b> <ul style="list-style-type: none"> <li>▪ <b>Return Value:</b> index of location where object resides.</li> </ul>
<a href="#">Insert</a>	<code>colObject.Insert(Index, Object)</code>	Inserts an <i>Object</i> into the <b>ArrayList</b> at the specified index. <ul style="list-style-type: none"> <li>▪ <b>Exceptions raised:</b> <a href="#">ArgumentOutOfRangeException</a> – When index = -1 or index &gt; Count <a href="#">NotSupportedException</a> – When array is read-only</li> </ul>
<a href="#">Remove</a>	<code>colObject.Remove(Object)</code>	Removes the first occurrence of a specific <i>Object</i> from the <b>ArrayList</b> . <ul style="list-style-type: none"> <li>▪ <b>Exceptions raised:</b> <a href="#">NotSupportedException</a> – When array is read-only</li> </ul>
<a href="#">RemoveAt</a>	<code>colObject.RemoveAt(Index)</code>	Removes the element at the specified index of the <b>ArrayList</b> . <ul style="list-style-type: none"> <li>▪ <b>Exceptions raised:</b> <a href="#">ArgumentOutOfRangeException</a> – When index = -1 or index &gt; Count <a href="#">NotSupportedException</a> – When array is read-only</li> </ul>
<a href="#">Reverse</a>	<code>colObject.Reverse()</code>	Reverses the order of the elements in the <b>ArrayList</b> <ul style="list-style-type: none"> <li>▪ <b>Exceptions raised:</b> <a href="#">NotSupportedException</a> – When array is read-only</li> </ul>

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

- ❑ This Class is of LIST Type or index collection. The objects stored in the collection are of type **OBJECCT**. We need to perform data type conversions or Casting when using objects retrieved from the collection in some cases.

### Item Property & For Each ..Next Loop

- ❑ You may need to convert the data in the following cases.
- ❑ **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()**

## 2.4 ArrayList Application Sample Program #1

- ❑ In this section we create a Customer Management example. In the previous examples we used an Array to store and manage Customer Object. A Customer Form was used to Add, Retrieve, Delete and Modify objects stored in the Array.
- ❑ In this example, we will replace the Array with an **ArrayList** to demonstrate the advantages of using an **ArrayList** instead of a regular array.
- ❑ We will also demonstrate some of the disadvantages to using an **ArrayList** as to other Collections such as **SortedList**.

---

### Example 2.1 - Module-Driven Window Application – ArrayList & Customer Objects Example

#### Problem statement:

- ❑ Create a Module-Driven Windows Application (Startup Object = Sub Main()) as in the previous examples.
- ❑ Upgrade the application as follows:

#### Object Management:

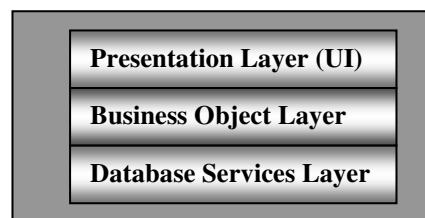
- Replace the Array with an ArrayList Collection. Use the **ArrayList Properties** and **Methods** to retrieve, add, edit and delete Customer Objects.
- Use the **clsPeson** Class with the same requirements from previous example to create Customer Objects.
- No user interaction or UI code should exist in the objects. The **PrintCustomer()** method should write the object's data to a text file name file name **CustomerPrinter.txt**.
- Add exception handling to trap the various Collection errors and situations.
- Set **Option Strict ON**, to see how we must convert data types using **CType()** in order to use objects retrieved from collections.

#### Form Requirements:

- Upgrade the Customer Form from previous example which only retrieved and edited customer objects to be able to add and delete as well.
- Add a **Print** feature to the form via a button control. When the button is clicked the customer whose ID is on the Form will be printed to the text file **PersonPrinter.txt**. Printing is to be done by the Customer object **Print()** method.
- Maintain the Customer Purchase feature using the purchase button and events features of the previous example.

#### Other Requirements:

- The application should be design with network expansion in mind; the design should use an application architecture using Object-Oriented-Programming Technology that will enable such growth. Implement this application with the three-tier application architecture described in class in mind:



- 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. Note that for the moment, we will write most of the processing in the Module. For now this is OK since in the future we can easily moving some of the code inside the classes. But for now, try to keep the UI with as less processing code as possible.

## HOW IT'S DONE:

### Part I – Create The Application:

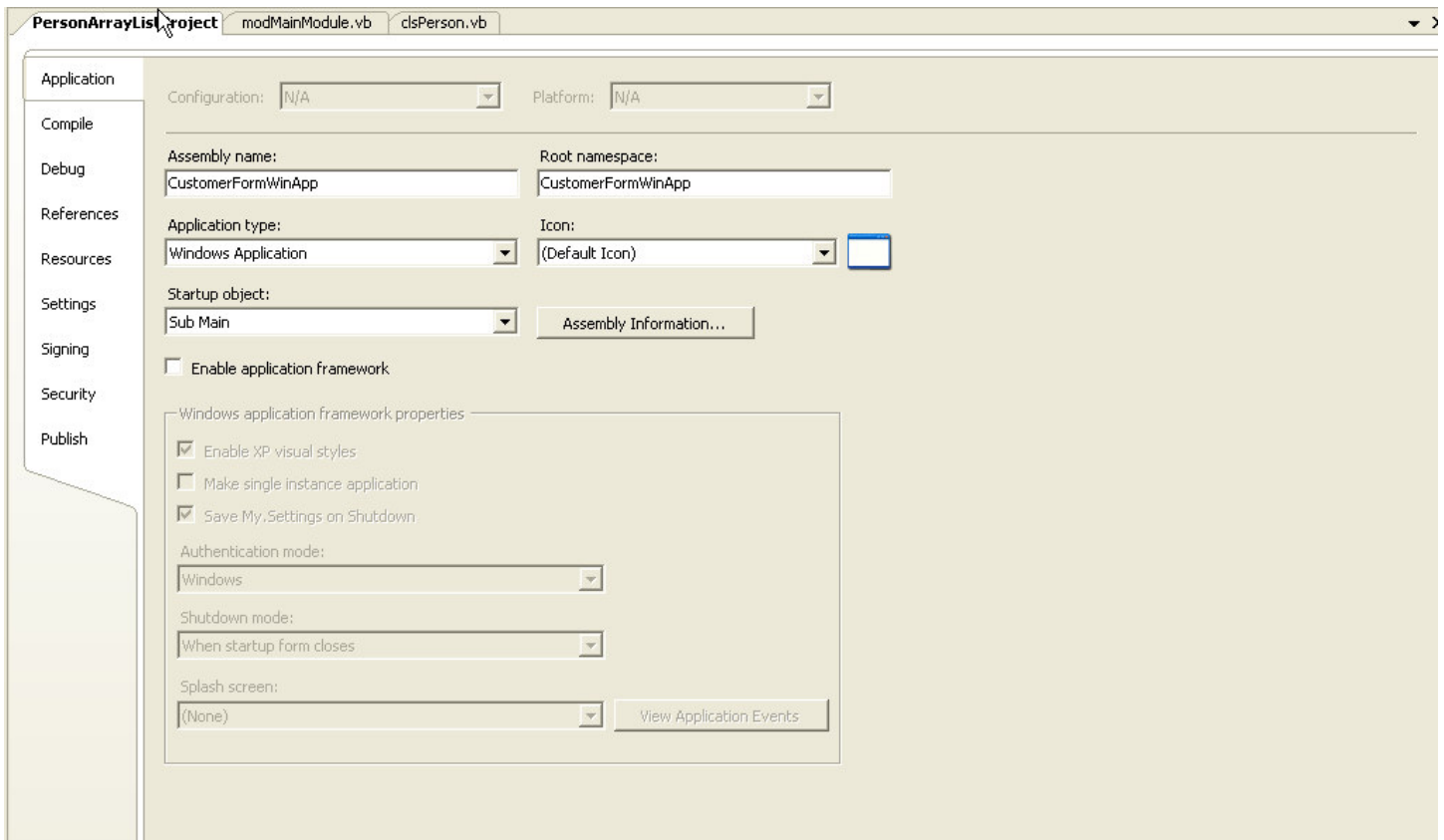
**Step 1: Start a new Windows Application project:**

**Step 2: Add a Form to the project and set its properties as previous example:**

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

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

Object	Property	Value
Project	Name	CustomerFormWinApp
	Startup Object	Sub Main()



## Business Object Layer – Class Objects

### Step 5: Add A NEW Class to the Project

1. In the Project Menu, select Add New Item...
2. Name the class *clsPerson.vb* File and click OK
3. The class is now part of the project and ready to be used!

### Step 6 & 7: Modify the Class as follows:

- Add declaration for importing the System.IO library needed for File Access Code
- Class Module keep private data section and Event declaration as is:

```
Option Strict On

'Impoted Libraries
Imports System.IO 'For file access code
Public Class clsPerson
    '*****
    Private m_Name 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:**

```
*****
'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: In the Class Module code window keep the code for the 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: Modify PrintPerson() Method as follows:**

```
'*****  
'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
```

## Presentation Layer (UI) – Module & Forms

### Part II – Module

#### Overview

- ❑ The module will contain all global code and control of the program via Sub Main()
- ❑ In addition we will place all code required to manage the Collections such as Add, Remove, Edit, GetItem, PrintCustomer etc.
- ❑ By doing this, we only place code in the Form for displaying and retrieving data from the user.
- ❑ In the future we can move all code that manages the list from the module into classes.

#### Step 11: In the Module Add the Following Code:

- ❑ Code any Global & Private Variable declarations and Sub Main()
  1. Set Option **Explicit to ON** and **Strict to Off** for this example. Note that Option **Strict Off** is default.
  2. Import the *System.Collections* Library
  3. Remove Array Declaration and add Global ArrayList Collection Object declaration
  4. Declare Global Customer Form Object

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

Module modMainModule

    'Declare Public Array of Person Objects
    Public CustomerList As New ArrayList()

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

#### Step 12: Create Sub Main as follows:

```
*****
''' <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 to display itself
''' </summary>
''' <remarks></remarks>
Public Sub Main()
    'Initialize the list, if required
    InitializeList()

    'Display Customer Form
    objCustomerForm.ShowDialog()

End Sub
```

**Step 13: Create InitializeList() Method as follows:**

```
*****
''' <summary>
''' Name: InitializeList() Method
''' Populates Collection object with an object
''' Step 1-Creates temp object populated with data
''' Step 2-Calls CollectionObject.Add(Object) Method to Add object.
''' </summary>
''' <remarks></remarks>
Public Sub InitializeList()
    'Create a Person Object
    Dim objDefaultCustomer As New clsPerson("Joe Smith", "111", #1/23/1971#, _
        "333 Jay Street", "718 260-5000")

    'Add object to ArrayList
    CustomerList.Add(objDefaultCustomer)

End Sub
```

**Step 15: Implement the GetItem method to manage the retrieval of objects from the list BASED on the INDEX:**

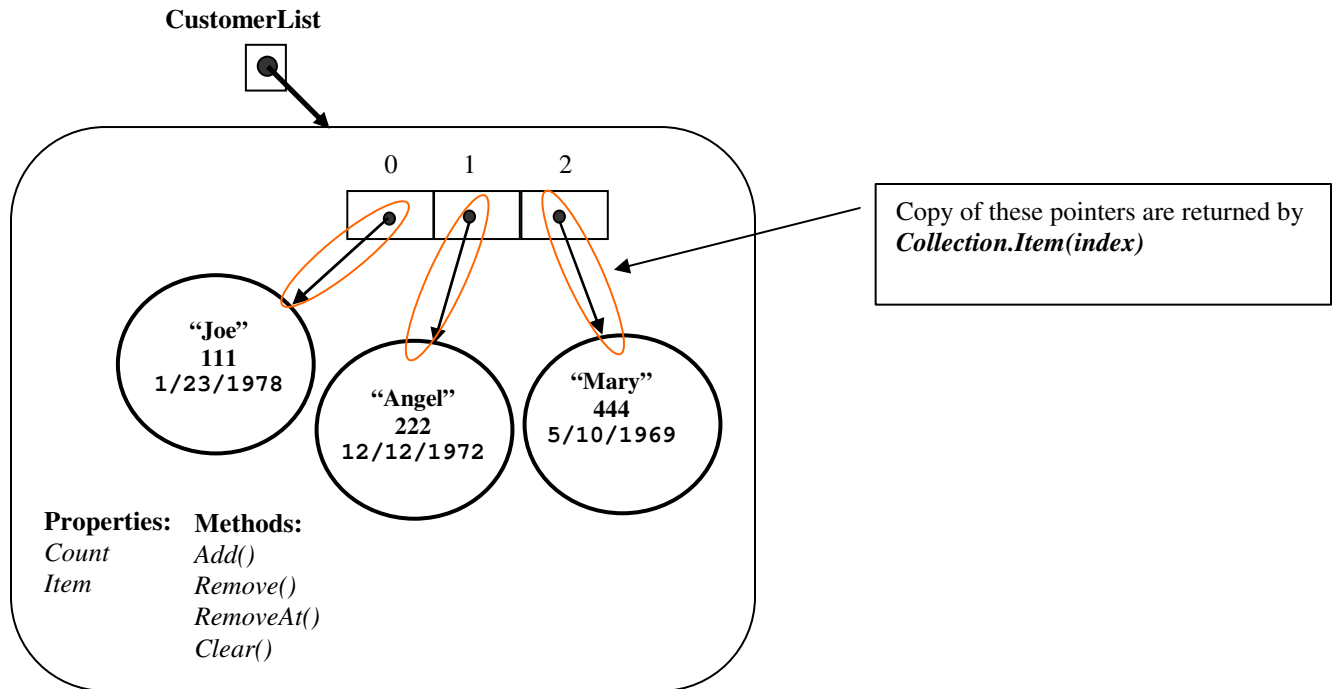
```
*****
''' <summary>
''' Name: Function - GetItem(Index)Function
''' Purpose: Retrieves item from collection
''' Return Type: Returns POINTER to object at the specified index location
''' Step A-Begins Exception handling
''' Step 1-Calls Collection.Item(Index) Method to get object from collection
''' Use CType() function to convert object retrieved from list to clsPerson
''' Step B-Traps for Collection Outofrange exceptions when index is invalid
''' Step C-Throw ArgumentOutOfRangeException. Throw must be trapped in Form
''' Step D-Traps for general exceptions
''' </summary>
''' <param name="index"></param>
''' <returns></returns>
''' <remarks></remarks>
Public Function GetItem(ByVal index As Integer) As clsPerson
    'Step A- Begin Error trapping
    Try

        'Step 1-Calls Collection.Item(Index) Method to get object from collection
        'Use CType() function to convert object retrieved from list to clsPerson
        Return CType(CustomerList.Item(index), clsPerson)

    'Step B-Traps for ArgumentOutOfRangeException exceptions when index is invalid or negative.
    Catch objX As ArgumentOutOfRangeException
        'Step C-Throw Collection ArgumentOutOfRangeException
        Throw New System.ArgumentOutOfRangeException("GetItem 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
```

## Discussion

- Is important that you understand that the GetItem Function returns a copy of the pointer CustomerList(INDEX) pointing to the object in the Collection whose index is INDEX
- The diagram below should make this clear.



**Step 14: Implement ANOTHER VERSION of GetItem method. This one will search based on IDNUMBER of Customer and return the object from the Collection:**

## Discussion

- Objective of this method is also to retrieve the object in the collection. This method is OVERLOADED version of works differently because we search for an object based on it's internal ID NUMBER NOT THE INDEX!!
- Normally in a program of this type where we need to search for a customer, we usually give the program the ID number of the customer. Transaction based programs DO NOT SEARCH ON INDEX. Would you ask a Customer walking into your store for Business for their INDEX NUMBER TO THE LIST OR DATABASE? Of course NOT!
- So we are forced to provide a method to get a Customer based on the ID NUMBER.

```

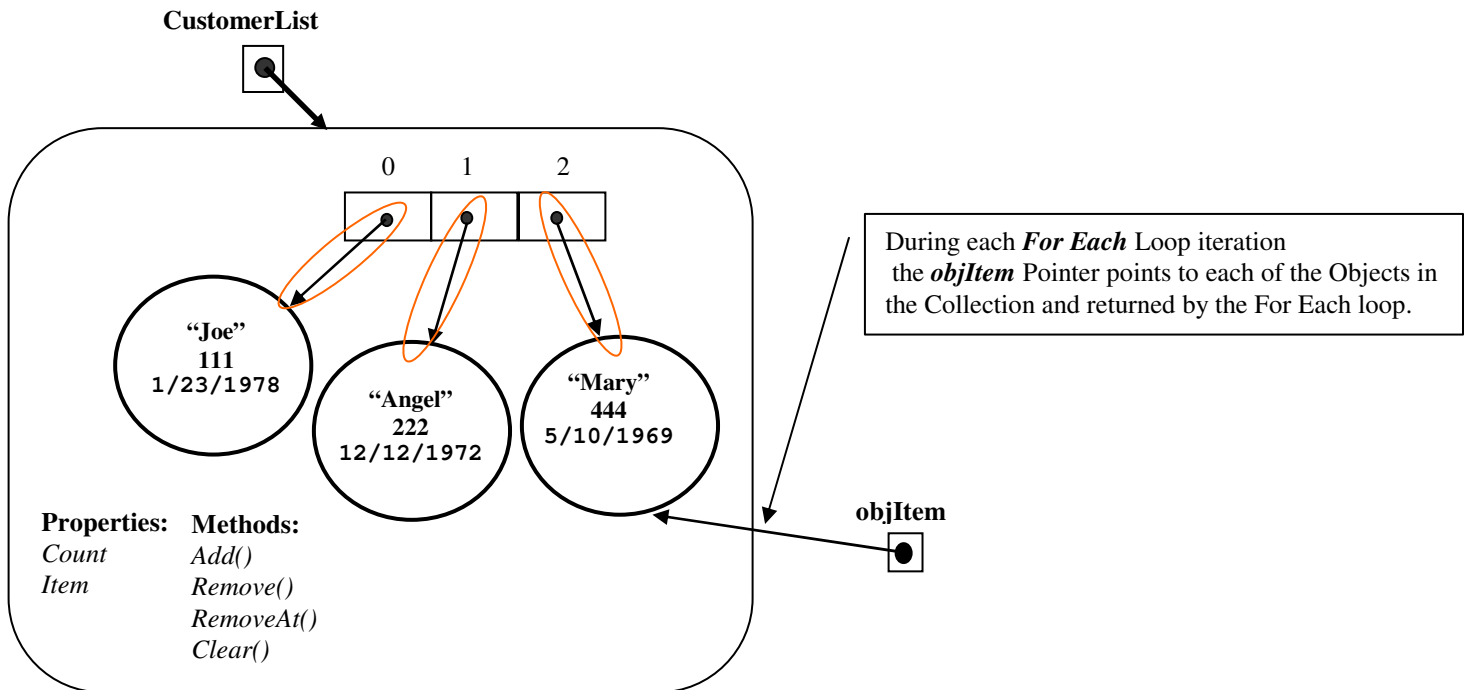
'*****
''' <summary>
''' Name:OVERLOADED GetItem(IDNum)Function
''' Purpose: Retrieves item from collection BASE ON IDNUMBER OF OBJECT
''' Return Type: Returns POINTER to object with IDNumber
''' </summary>
''' <param name="strIDNum"></param>
''' <returns></returns>
''' <remarks></remarks>
Public Function GetItem(ByVal strIDNum As String) As clsPerson
'Step A- Begin Error trapping
Try
'Step 1-Create Temporary OBJECT POINTER!!!!
Dim objItem As clsPerson

'Step 2-Iterates through Collection based on Collection.Count property.
'Temp Pointer points to each object during every iteration.
For Each objItem In CustomerList
'Step 3-Verifies if object.ID is the one being searched, if so return object
If objItem.IDNumber = strIDNum Then
'Step 4-Found therefore return object pointer & Exit
Return objItem
End If
Next

'Step 5-Return Nothing since not found & Exit
Return Nothing

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

```



**Step 15: Implement a method named GetIndexByOfItemByID. This one will search based on IDNUMBER of Customer and return the INDEX or LOCATION of object in the Collection:**

### Discussion

- Objective of this method is also to retrieve the INDEX of the Customer in the collection BASED ON THE ID NUMBER. This method is necessary because most of the METHODS provided by the ArrayList work based INDEX, NOT ID NUMBER!!
- So, in order to perform Deletes, Edits etc. we need the INDEX of the OBJECT in the Collection, therefore this method will return the INDEX if you give it the ID NUMBER of a Customer!
- Once we have the INDEX, we can then perform some of the functionality on the objects such as Remove, Edit etc.
- So we are forced to provide a method to get the INDEX of a Customer in the array based on the ID NUMBER.
- In this example we also use a regular FOR LOOP instead of the FOR EACH, to give you example of using another loop.

```
'*****  
' <summary>  
' Name: GetIndexOfItemByID(IDNum)Function  
' Purpose: Retrieves INDEX of Object in collection BASE ON IDNUMBER OF OBJECT  
' Return Type: Returns Object found in list  
' </summary>  
' <param name="strIDNum"></param>  
' <returns></returns>  
' <remarks></remarks>  
Public Function GetIndexOfItemByID(ByVal strIDNum As String) As Integer  
    'Step A- Begin Error trapping  
    Try  
        Dim i As Integer  
        'Step 1-Create Temporary OBJECT POINTER!!!!  
        Dim objItem As clsPerson  
  
        'Step 2-Iterates through Collection based on Collection.Count property.  
        For i = 0 To CustomerList.Count - 1  
            'Step 3-Calls Collection.Item(Index) to get pointer to object in collection  
            'Use CType() function to convert object retrieved from list to clsPerson  
            objItem = CType(CustomerList.Item(i), clsPerson)  
            'Step 4-Verifies if object.ID is the one being searched  
            If objItem.IDNumber = strIDNum Then  
                'Step 5-Found therefore return Index to location of object  
                Return i  
            End If  
        Next  
  
        'Step 6-Destroy temp object  
        objItem = Nothing  
  
        'Step 7-Return a -1 object not found  
        Return -1  
  
        'Step B-Traps for ArgumentOutOfRangeException when index is invalid or negative.  
        Catch objX As ArgumentOutOfRangeException  
            'Step C-Throw Collection ArgumentOutOfRangeException  
            Throw New System.ArgumentOutOfRangeException("GetIndexOfItem Error: " & objX.Message)  
            'Step D-Traps for general exceptions.  
            Catch objE As Exception  
                'Step E-Throw an general exceptions  
                Throw New System.Exception("GetIndexOfItem Error: " & objE.Message)  
            End Try  
    End Function
```

## Step 16: Implement the Add method to manage the addition of objects into the list:

### Discussion

- The objective of this method is to ADD a NEW object to the collection.
- I will show two different implementation of ADD. This will be VERSION 1
- In this case, passed as argument is the NEW OBJECT to be added

```
*****  
''' <summary>  
''' Name: Add(object)Function  
''' Purpose: Adds new object to the Collection  
''' </summary>  
''' <param name="objItem"></param>  
''' <remarks></remarks>  
Public Sub Add(ByVal objItem As clsPerson)  
    'Step A- Begin Error trapping  
    Try  
  
        'Step 1-Calls Collection.Add(Object) Method to Add object passed as argument  
        CustomerList.Add(objItem)  
  
        'Step B-Traps for Collection Object NotSupportedException exceptions.  
        Catch objX As NotSupportedException  
            'Step C-Throw an NotSupportedException  
            Throw New System.NotSupportedException("Add Method Error: " & objX.Message)  
            'Step D-Traps for general exceptions.  
        Catch objE As Exception  
            'Step E-Throw an general exceptions  
            Throw New System.Exception("Add Method Error: " & objE.Message)  
        End Try  
    End Sub
```

**Step 17: Implement the SECOND VERSION or OVERLOADED Add method to Add objects to the list**

- Objective of this method is also to ADD object to the collection. This method is OVERLOADED version of Add().
- This version 2 performs the same functionality of version1, which is to add the object; the main difference here is that the individual values or properties are being passed as argument. THE NEW OBJECT IS CREATED inside the method and POPULATED with the values passed as arguments.

```
*****  
''' <summary>  
''' Name: Overloaded Add(value1, value2..)Method  
''' Purpose: Adds new object to the Collection  
''' </summary>  
''' <param name="strName"></param>  
''' <param name="strIDNum"></param>  
''' <param name="dBDate"></param>  
''' <param name="strAddress"></param>  
''' <param name="strPhone"></param>  
''' <remarks></remarks>  
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 Temp Object  
        Dim objItem As New clsPerson  
  
        'Step 2-Populates object it with data passed as argument  
        With objItem  
            .Name = strName  
            .IDNumber = strIDNum  
            .BirthDate = dBDate  
            .Address = strAddress  
            .Phone = strPhone  
        End With  
  
        'Step 3-Use Collection Add Method add Object to Collection  
        CustomerList.Add(objItem)  
  
        'Step B-Traps for Collection Object NotSupportedException exceptions.  
        Catch objX As NotSupportedException  
            'Step C-Throw an NotSupportedException  
            Throw New System.NotSupportedException("Add Method Error: " & objX.Message)  
            'Step D-Traps for general exceptions.  
        Catch objE As Exception  
            'Step E-Throw an general exceptions  
            Throw New System.Exception("Add Method Error Error: " & objE.Message)  
        End Try  
    End Sub
```



**Step 18: Implement the EditItem method to manage the process of modifying objects in the list (VERSION 1):**

**Discussion**

- The objective of this method is to modify the object in the collection.
- This version of the implementation REPLACES THE ORIGINAL OBJECT with the MODIFIED VERSION!!!
- Passed as argument is the index were the object is located, and the NEW OBJECT to replace the original

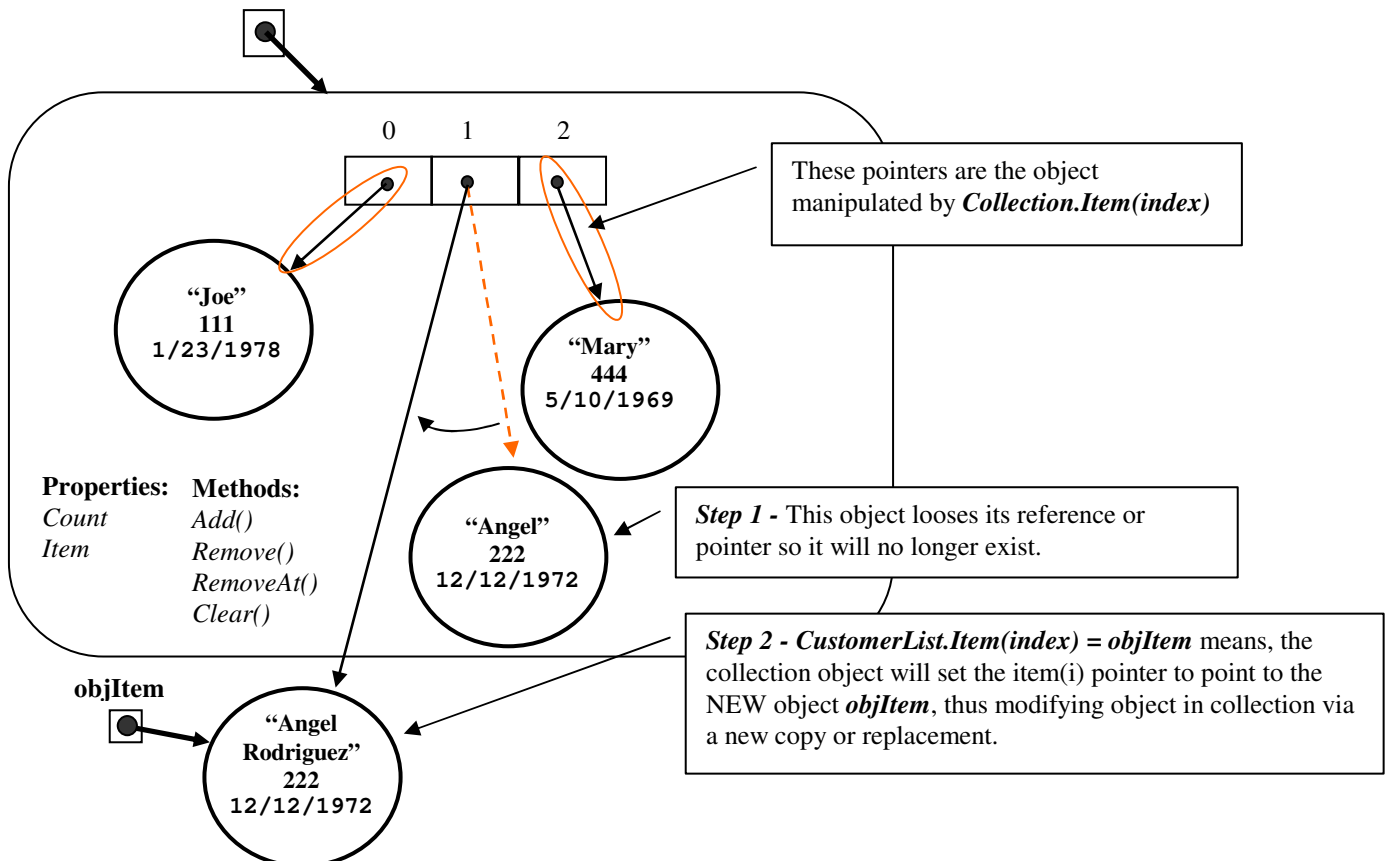
```

'*****
''' <summary>
''' Name:EditItem(Index, object)Method
''' </summary>
''' <param name="index"></param>
''' <param name="objItem"></param>
''' <remarks></remarks>
Public Sub EditItem(ByVal index As Integer, ByVal objItem As clsPerson)
    'Step A- Begin Error trapping
    Try
        'Step 1-Calls CollectionObject.Item(Key) = object syntax to perform the replacement
        CustomerList.Item(index) = objItem

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

```

CustomerList Object



**Step 19: Implement the SECOND VERSION of EditItem method to manage the process of modifying objects in the list:**

**Discussion**

- Objective of this method is also to modify the object in the collection. This method is OVERLOADED version of edititem.
- This version 2 performs the same functionality of version1, which is to modify the object, the main difference here is that THE ORIGINAL OBJECT IS BEING MODIFIED VIA ITS PROPERTIES, NOT A COPY OR NEW OBJECT!

```
*****
''' <summary>
''' EditItem(value1, value2..)Method
''' Purpose: Sets or overwrites object located at specified index in the Collection
''' </summary>
''' <param name="index"></param>
''' <param name="strName"></param>
''' <param name="strIDNum"></param>
''' <param name="dBDate"></param>
''' <param name="strAddress"></param>
''' <param name="strPhone"></param>
''' <remarks></remarks>
Public Sub EditItem(ByVal index As Integer, 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-Create temporary pointer
        Dim objItem As clsPerson

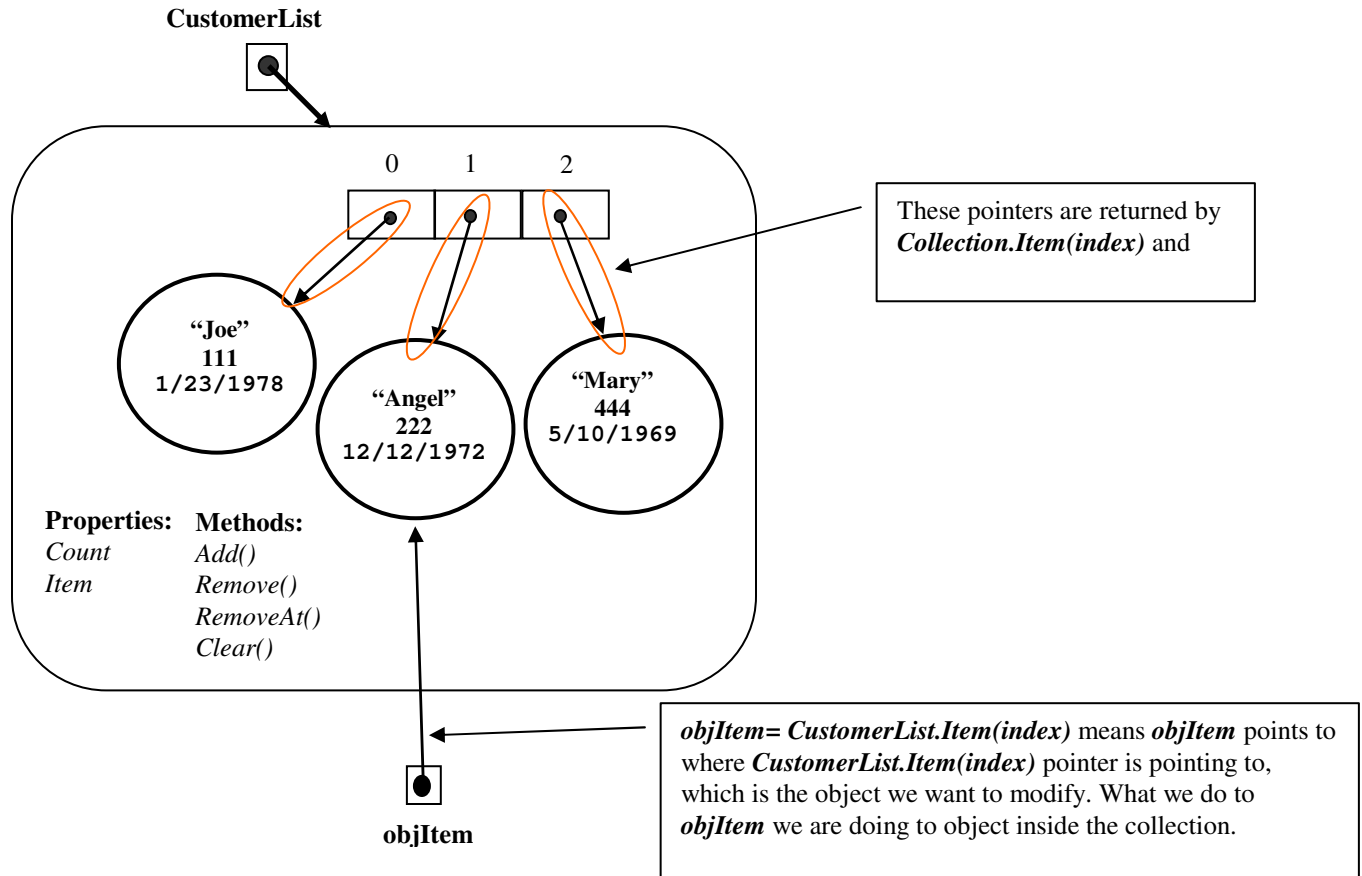
        'Step 2-Get a Reference of pointer to the actual object inside the collection.
        'Convert data type of object reference
        objItem = CType(CustomerList.Item(index), clsPerson)

        'Step 3-Sets individual properties of actual object inside the collection.
        objItem.Name = strName
        objItem.IDNumber = strIDNum
        objItem.BirthDate = dBDate
        objItem.Address = strAddress
        objItem.Phone = strPhone

        'Step B-Traps for Collection ArgumentOutOfRangeException exceptions
        Catch objX As ArgumentOutOfRangeException
            'Step C-Throw an ArgumentOutOfRangeException to calling programs.
            Throw New System.ArgumentOutOfRangeException("EditItem Method Error: " & objX.Message)
        'Step D-Traps for general exceptions.
        Catch objE As Exception
            'Step E-Throw an general exceptions
            Throw New System.Exception("EditItem Method Error: " & objE.Message)
        End Try
    End Sub
```

- This process of MODIFYING THE ORIGINAL OBJECT is important. Compared to version 1, which replaces with a COPY, Is usually not desirable to replace with a NEW object because we may lose some of the data of the original, data which are not set through properties but are internal to the original.
- In this version, passed as argument is the index where the object is located, and the individual values or properties to be replaced in the ORIGINAL OBJECT!

- This algorithm can be confusing if you don't understand POINTERS!!
- We give the Collection the INDEX and it returns a POINTER to the object INSIDE THE COLLECTION. What ever we do to the pointer, we are doing to the exiting OBJECT IN THE COLLECTION.



**Step 20: Implement the Remove method to manage the removal of objects from the list:**

```

'*****
''' <summary>
''' Name: Remove(IDNumber)Sub Method
''' Purpose: Remove object from collection based on CustomerID.
''' </summary>
''' <param name="Index"></param>
''' <remarks></remarks>
Public Sub Remove (ByVal Index As Integer)
    'Step A- Begin Error trapping
    Try

        'Step 1-Calls CollectionObject.RemoveAt (Index) Method
        CustomerList.RemoveAt (Index)

        'Step B-Traps for Collection ArgumentOutOfRangeException exceptions
        Catch objX As ArgumentOutOfRangeException
            'Step C-Throw an ArgumentOutOfRangeException to calling programs
            Throw New System.ArgumentOutOfRangeException("Remove Method Error: " & objX.Message)
        'Step D-Traps for general exceptions.
        Catch objE As Exception
            'Step E-Throw an general exceptions
            Throw New System.Exception("Remove Method Error: " & objE.Message)
        End Try
    End Sub

```

**Step 21: Implement the PrintCustomer method to manage the process of printing an objects to File:**

```

'*****
''' <summary>
''' Name: PrintCustomer(index)Sub Method
''' Purpose: Prints object from collection to Printer File
''' </summary>
''' <param name="Index"></param>
''' <remarks></remarks>
Public Sub PrintCustomer(ByVal Index As Integer)
'Step A- Begin Error trapping
Try
'Step 1-Create temporary pointer
Dim objItem As New clsPerson

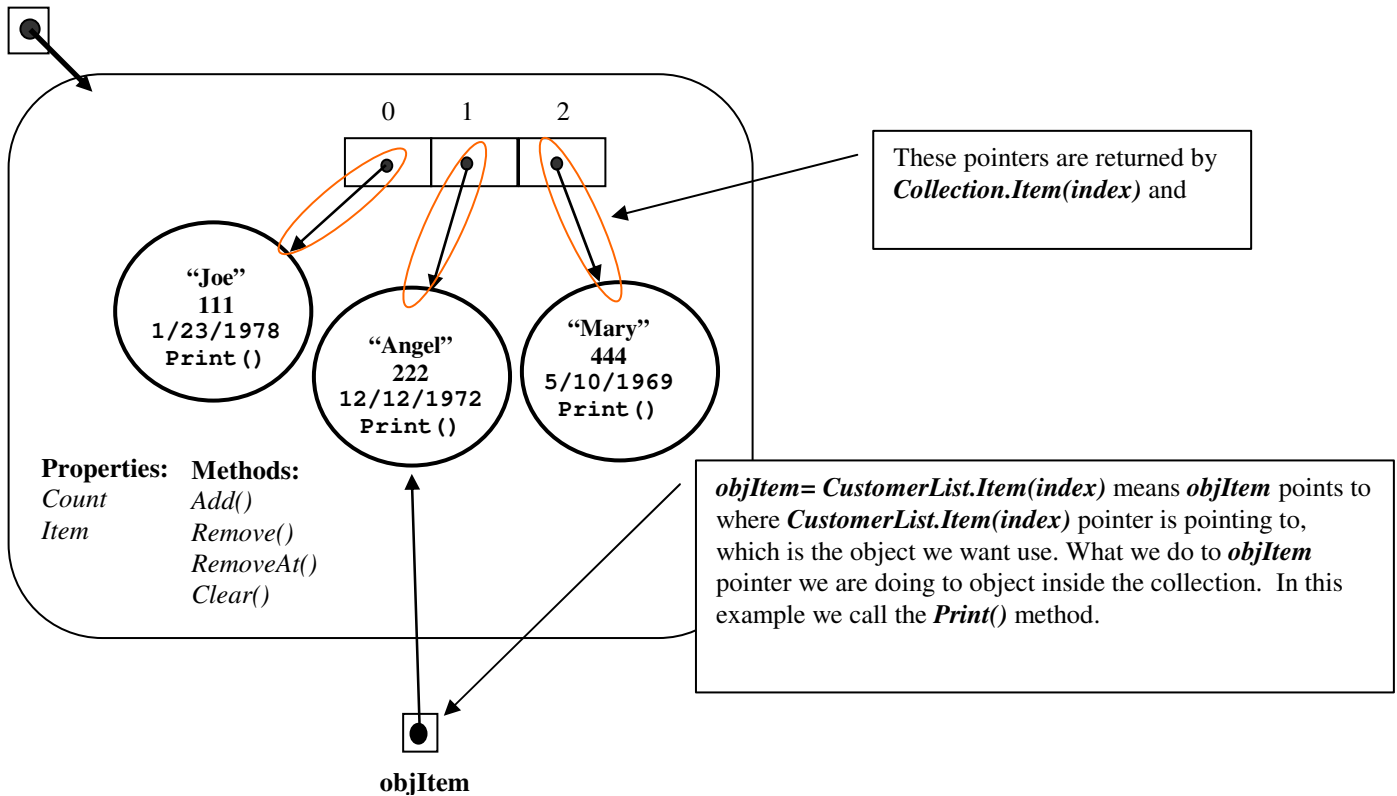
'Step 2-Get object in collection, convert & assing to temp pointer
objItem = CType(CustomerList.Item(Index), clsPerson)

'Step 3-Calls Temp Object.Print Method to print the object to file
objItem.Print()

'Step B-Trap collection ArgumentOutOfRangeException exceptions
Catch objX As ArgumentOutOfRangeException
'Step C-Throw an ArgumentOutOfRangeException
Throw New System.ArgumentOutOfRangeException("PrintCustomer Error: " & objX.Message)
'Step D-Traps for general exceptions.
Catch objE As Exception
'Step E-Throw an general exceptions
Throw New System.Exception("PrintCustomer Method Error: " & objE.Message)
End Try
End Sub

```

CustomerList



**Step 22: Implement the PrintAllCustomers method to print all the Customers in the list to File:**

```
*****
''' <summary>
''' Name: PrintAllCustomers() Sub Method
''' Purpose: Prints object from collection to Printer File
''' </summary>
''' <remarks></remarks>
Public Sub PrintAllCustomers()
    'Step A- Begin Error trapping
    Try
        'Step 1-Create temporary pointer
        Dim objItem As clsPerson

        'Step 2-Use For..Each loop to iterate through arraylist
        'Temp Pointer points to each object during every iteration.
        For Each objItem In CustomerList
            'Step 3-Calls Temp Object.PrintPerson Method to print the object to file
            objItem.Print()
        Next

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

End Module
```

## Part III – User Interface Form

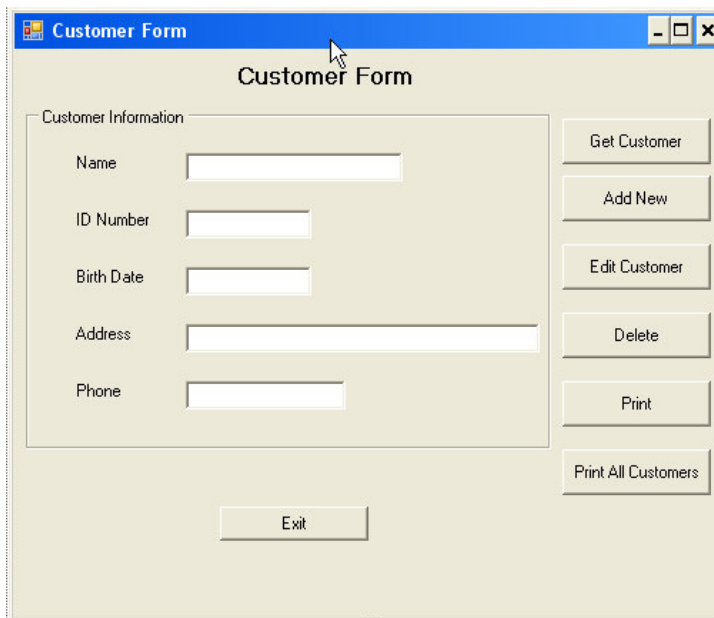
### Overview

- ❑ We will reuse the Customer Form from the previous example.
- ❑ We will add buttons for the new Add, Delete and Print() functionalities.
- ❑ Note that the code in the Form will be keep minimal since most of the processing is done in the Module.

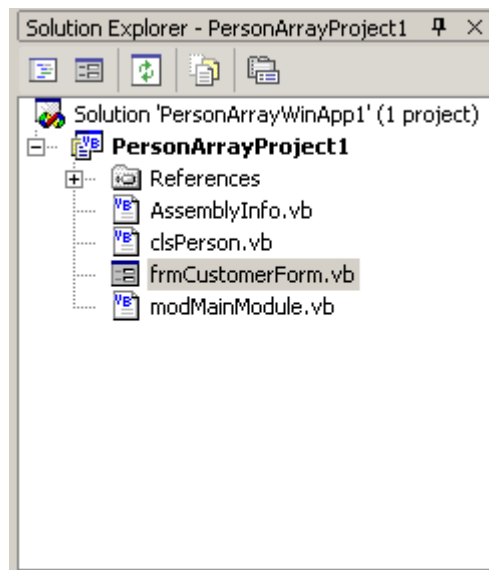
**Step 24: Add the following Controls to the frmCustomerForm. Set their properties accordingly:**

Object	Property	Value
Form1	Name	<b>frmCustomerForm</b>
	Text	<b>Customer Form</b>

- ❑ Note that the Form now includes a buttons for Adding, Editing, Deleting, retrieving, purchasing and printing the Customer records.



**Step 25: At this Point the Project should look as follows:**



**Step 26: In the Form frmCustomerForm declare a Temp object to use in various Form operations and create it in the load handler. Also destroy it in the Form.Close() handler:**

```
Option Explicit On
Option Strict On

Public Class frmCustomerForm
    Inherits System.Windows.Forms.Form

    'Declare Form Level Object
    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
```

**Step 27: In the OK button event-handler simply Close the form:**

### Discussion

- Note how very little code is done in the OK button. We don't want to add processing code on the OK event.

```
'*****
''' <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
```

## Step 28: Modify the GetCustomer Event to use the ArrayList management methods in the Module:

### Discussion

#### □ The GetCustomer Click Event:

- In this Event-Handler we will call the search method of the module to search for the customer object in the array by ID
- Note that the OVERLOADED *GetItem(IDNUM)* method returns the object or Customer in the list.
- Also note that the result of the *GetItem ()* can return a NULL POINTER OR NOTHING if not found.
- From the results we either populate the text boxes with the customer data or display that the Customer was not found.
- In addition, we need to trap for the errors generated by the COLLECTION as well as GENERAL ERRORS.

```
'*****  
' <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 Overloaded GetItem(ID) to search for object that match ID  
        objCustomer = GetItem(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  
                txtName.Text = .Name  
                txtIDNumber.Text = .IDNumber  
                txtBirthDate.Text = CStr(.BirthDate)  
                txtAddress.Text = .Address  
                txtPhone.Text = .Phone  
            End With  
        End If  
  
        'Step B-Traps for Overflow exceptions and displays appropriate messages  
        Catch objX As ArgumentOutOfRangeException  
            MessageBox.Show(objX.Message)  
        'Step D-Traps for General exceptions and displays appropriate message  
        Catch objE As Exception  
            MessageBox.Show("Unexpected Error: " & objE.Message)  
        End Try  
End Sub
```



## Step 30: Implement an Add Event to use the ArrayList management methods in the Module to Add the new Customer:

### Discussion

#### □ The Add Click Event:

- In this Event-Handler call the OVERLOADED module's *Add(value1, value2, etc.)* method to perform the operation.
- Data is passed as argument to this method
- We need to trap for the errors generated by the COLLECTION as well as GENERAL ERRORS.
- The user is also informed of the errors via message boxes

```
'*****  
' <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-Begins Exception handling.  
    Try  
  
        'Step 1-Calls Add() method of module, object is passed as argument  
        Add(txtName.Text, txtIDNumber.Text, CDate(txtBirthDate.Text), _  
            txtAddress.Text, txtPhone.Text)  
  
        'Step B-Traps for NotSupportedException exceptions and displays messages  
        Catch objX As NotSupportedException  
            MessageBox.Show(objX.Message)  
        'Step D-Traps for General exceptions and displays appropriate message  
        Catch objE As Exception  
            MessageBox.Show("Unexpected Error: " & objE.Message)  
        End Try  
End Sub
```

### Step 31: Implement the EditCustomer Event to modify the customer in the ArrayList via method in the Module:

#### Discussion

□ *The EditCustomer Click Event:*

- In this Event-Handler we modify an object in the list.
- Again we are forced to call the module's *GetIndexOfItemByID(ID)* method to search for the object via it's ID and get the index of the object in the list.
- This is another indication why the **ArrayList** is not a good choice for this type of program.
- Once we have the index then we can call the module's *EditItem(Index, Value1, Value2...)* method to perform the operation.
- We need to trap for the errors generated by the COLLECTION as well as GENERAL ERRORS.
- The user is also informed of the errors via message boxes

```
'*****  
'<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  
    Dim index As Integer  
    'Step A-Begins Exception handling.  
    Try  
  
        'Step 1-Call GetIndexOfItem(ID) to search for ID in list and return the index  
        index = GetIndexOfItemByID(txtIDNumber.Text)  
  
        'Step 2-If result of search is = -1, then display customer is not found  
        If index = -1 Then  
            MessageBox.Show("Customer Not Found")  
        Else  
            'Step 3-Call Module EditItem(index,x,y,z,...) method with textbox data  
            EditItem(index, txtName.Text, txtIDNumber.Text, _  
                CDate(txtBirthDate.Text), txtAddress.Text, txtPhone.Text)  
        End If  
  
        'Step B-Traps for ArgumentOutOfRangeException and displays message  
        Catch objX As ArgumentOutOfRangeException  
            MessageBox.Show(objX.Message)  
        'Step D-Traps for General exceptions and displays appropriate message  
        Catch objE As Exception  
            MessageBox.Show("Unexpected Error: " & objE.Message)  
        End Try  
End Sub
```

## Step 32: Implement the Delete Event to delete the customer from the ArrayList via method in the Module:

### Discussion

#### □ *The Delete Click Event:*

- In this Event-Handler we delete an object from the list.
- Again we are forced to call the module's *GetIndexOfItem(ID)* method to search for the object via it's ID and get the index of the object in the list.
- This is another indication why the **ArrayList** is not a good choice for this type of program.
- Once we have the index then we can call the module's *Remove(Index)* method to perform the operation.
- We also trap for the errors generated by the COLLECTION as well as GENERAL ERRORS and prompt the user

```
'*****  
' ' <summary>  
' ' Name: Event-Handler for btnDelete button  
' ' Purpose: To delete an object from the collection base on ID or Key  
' ' </summary>  
' ' <param name="sender"></param>  
' ' <param name="e"></param>  
' ' <remarks></remarks>  
Private Sub btnDelete_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)  
Handles btnDelete.Click  
    Dim Index As Integer  
    'Step A-Begins Exception handling.  
    Try  
  
        'Step 1-Call GetIndexOfItem(ID) to search for ID in list and return the index  
        Index = GetIndexOfItemByID(txtIDNumber.Text)  
  
        'Step 2-If result of search is = -1, then display customer is not found  
        If Index = -1 Then  
            MessageBox.Show("Customer Not Found")  
  
            'Step 3-Clear all controls  
            txtName.Text = ""  
            txtIDNumber.Text = ""  
            txtBirthDate.Text = ""  
            txtAddress.Text = ""  
            txtPhone.Text = ""  
        Else  
            'Step 4-Calls Remove() method of module  
            Remove(Index)  
        End If  
  
        'Step B-Traps for ArgumentOutOfRangeException exceptions & displays messages  
        Catch objX As ArgumentOutOfRangeException  
            MessageBox.Show(objX.Message)  
            'Step D-Traps for General exceptions and displays appropriate message  
        Catch objE As Exception  
            MessageBox.Show("Unexpected Error: " & objE.Message)  
        End Try  
End Sub
```

### Step 33: Implement the Print Event to print to file the customer via method in the Module:

#### Discussion

##### □ The Print Click Event:

- In this Event-Handler we print an object in the list to file.
- Again we are forced to call the module's *GetIndexOfItemByID (ID)* method to search for the object via it's ID and get the index of the object in the list.
- This is another indication why the ArrayList is not a good choice for this type of program.
- Once we have the index then we can call the module's *PrintCustomer(Index)* method to perform the operation.
- We also trap for the errors generated by the COLLECTION as well as GENERAL ERRORS and prompt the user

```
*****  
' ' <summary>  
' ' Name: Event-Handler for btnPrint button  
' ' Purpose: Prints Object in the list  
' ' </summary>  
' ' <param name="sender"></param>  
' ' <param name="e"></param>  
' ' <remarks></remarks>  
Private Sub btnPrint_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)  
Handles btnPrint.Click  
    Dim Index As Integer  
    'Step A-Begins Exception handling.  
    Try  
  
        'Step 1-Call GetIndexOfItem(ID) to search for ID in list and return the index  
        Index = GetIndexOfItemByID(txtIDNumber.Text)  
  
        'Step 2-If result of search is = -1, then display customer is not found  
        If Index = -1 Then  
            MessageBox.Show("Customer Not Found")  
  
            'Step 3-Clear all controls  
            txtName.Text = ""  
            txtIDNumber.Text = ""  
            txtBirthDate.Text = ""  
            txtAddress.Text = ""  
            txtPhone.Text = ""  
        Else  
            'Step 4-Calls Remove() method of module. Index from search is passed as argument  
            PrintCustomer(Index)  
        End If  
  
        'Step B-Traps for ArgumentOutOfRangeException exceptions & display message  
        Catch objA As ArgumentOutOfRangeException  
            MessageBox.Show(objA.Message)  
        'Step D-Traps for General exceptions and displays appropriate message  
        Catch objE As Exception  
            MessageBox.Show("Unexpected Error: " & objE.Message)  
        End Try  
  
End Sub
```

## Step 34: Implement the PrintAll Event to print to file the customer via method in the Module:

### Discussion

□ *The PrinAllCustomer* Click Event:

- Call the module's *PrintAllCustomers()* method to perform the operation.
- In the User-Interface, we need to trap for the errors generated by the COLLECTION as well as GENARAL ERRORS.
- The user is also informed of the errors via message boxes

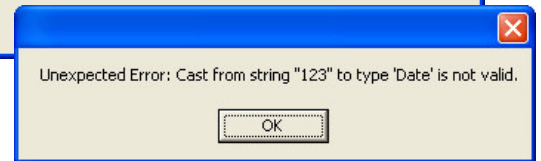
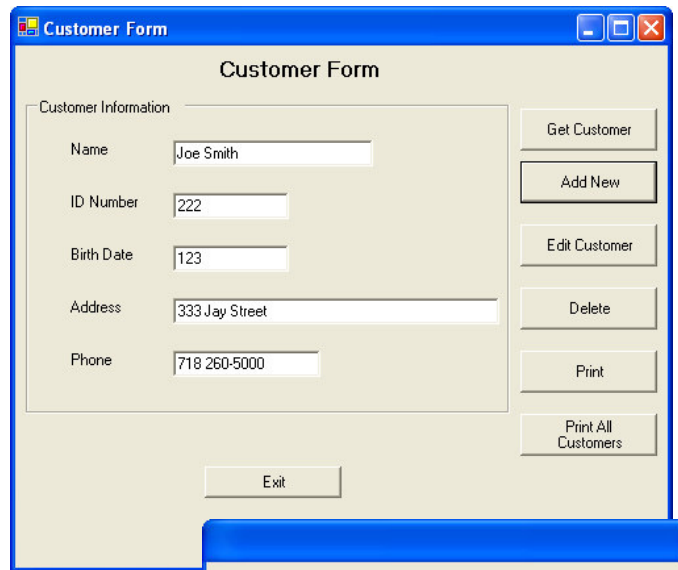
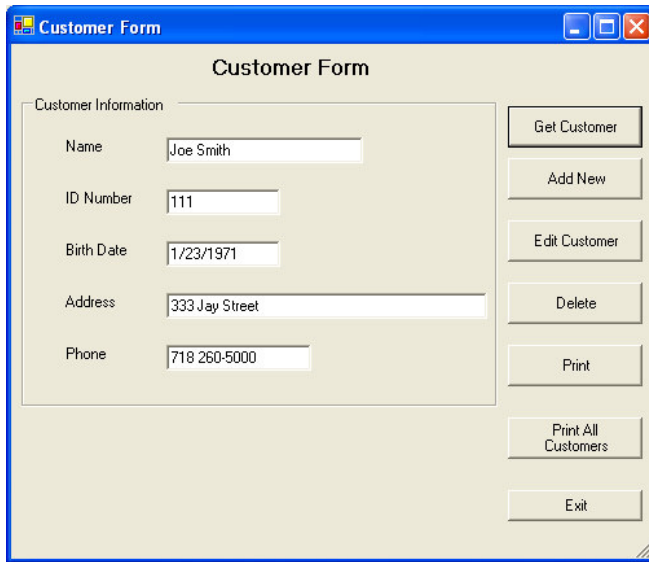
```
'*****  
''' <summary>  
''' Name: Event-Handler for btnPrintAllCustomers button  
''' Purpose: Prints all Objects in the list  
''' </summary>  
''' <param name="sender"></param>  
''' <param name="e"></param>  
''' <remarks></remarks>  
Private Sub btnPrintAll_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)  
Handles btnPrintAll.Click  
    'Step A-Begins Exeception handling.  
    Try  
        'Step 1-Calls Remove() method of module. Index from search is passed as argument  
        PrintAllCustomers()  
        'Step B-Traps for ArgumentOutOfRangeException exceptions, display message  
        Catch objX As ArgumentOutOfRangeException  
            MessageBox.Show(objX.Message)  
        'Step D-Traps for General exceptions and displays appropriate message  
        Catch objE As Exception  
            MessageBox.Show("Unexpected Error: " & objE.Message)  
        End Try  
    End Sub  
  
End Class
```

## Part IV – Output & Summary

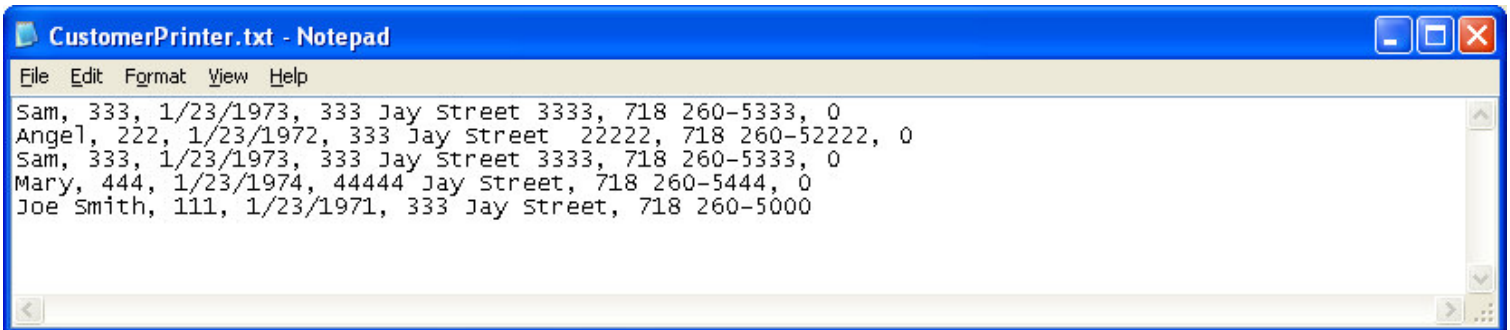
### Summary

- ❑ 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 choosing the ArrayList for this 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:



```
File Edit Format View Help
Sam, 333, 1/23/1973, 333 Jay Street 3333, 718 260-5333, 0
Angel, 222, 1/23/1972, 333 Jay Street 22222, 718 260-52222, 0
Sam, 333, 1/23/1973, 333 Jay Street 3333, 718 260-5333, 0
Mary, 444, 1/23/1974, 44444 Jay Street, 718 260-5444, 0
Joe Smith, 111, 1/23/1971, 333 Jay Street, 718 260-5000
```