CS608 Project 2 – Video Management System (Version 2)

Take Home Project

Professor A. Rodriguez

Program Statement

- You **re-hired** as a consultant by ***NYCTech Solutions Inc***., (Represented by Prof. Rodriguez) to **UPDATE** the **Video Management System** application for **MediaBusters** Inc

- The *client* (Prof. Rodriguez) has the right to **fire you** (Fail you) at any time or reduce the amount of payment (Low Grade) during the development of the system, if the system does not work, you fall behind schedule (Late) and most important **NOT** following the program ***REQUIREMENTS***. **Warning!** NOT following the requirements can quickly lead to the **unemployment line**!

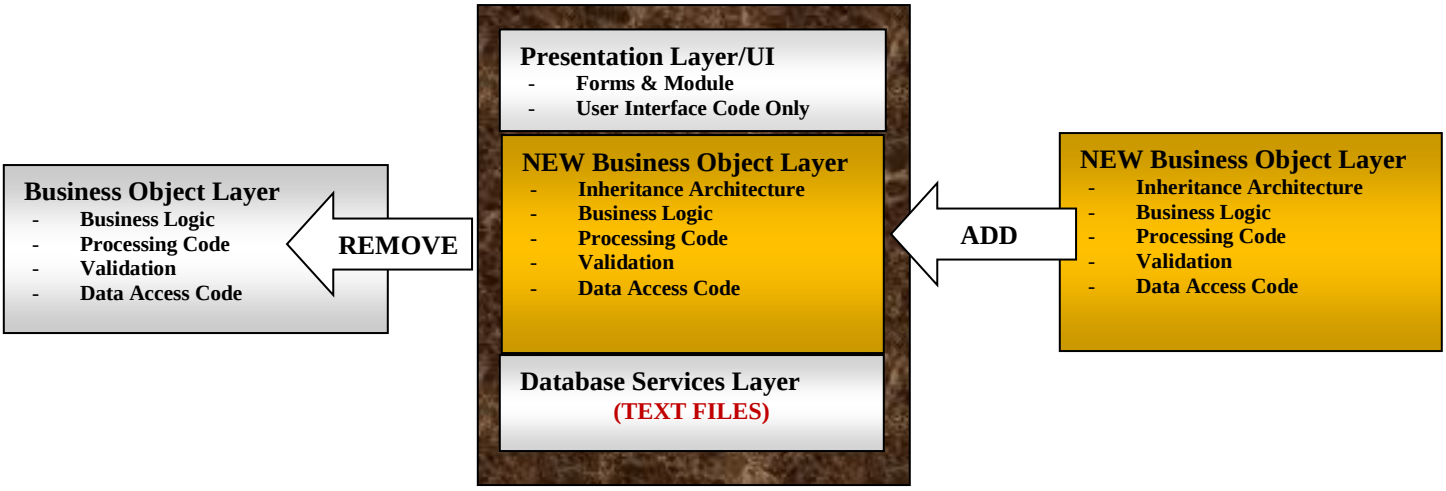
General Programming Requirements

- 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. Note that the owner has already hired a high-end consultant to design the object model. Therefore the class/object model is already designed for you. Thus your job is to take this model and create or code the application. You can add any functionality you required to the object model, **BUT you should NOT remove any of the current object model class specifications, without the approval of the your contractor (Prof Rodriguez)!**

Application Architecture

- The main objectives of this project are to CREATE A NEW BUSINESS OBJECT LAYER.
- We are going to swap the BO LAYER and in theory the USER-INTERFACE SHOULD STAY THE SAME.
- Continue to implement the ***3-tiered Client/Server Application Architecture***, nevertheless this time you are to IMPLEMENT a NEW BUSINESS OBJECT LAYER USING INHERITANCE BETWEEN a PERSON & CUSTOMER CLASS.

    - This is an example of why this architecture is so flexible. We are able to change the middle or ***Business Object Layer*** with a new one with little or no modification to the ***Presentation/User-Interface Layer*** or ***Database layers***



Overall Requirements Summary

- **There are a total of 12 Requirements you must satisfy to get an FULL PAYMENT (A GRADE) in this project**.
- **But in reality, it is ONLY REQUIREMENTS #2 & #3 YOU WILL BE IMPLEMENTING**. The remaining requirements ARE FOR User-Interface Layer and Data Service Layer which WERE IMPLEMENTED IN PROJECT #1

Requirement #1 – Upgrade Back-End Management System but Keep all Functionality & Requirements of Project 1

- In this second phase (PROJECT 2), you are to UPGRADE the following:

  - **Back-end Management System**
    - ***Customer Management***,
    - ***DVD Management***
    - ***Video Game Management***
    - ***Employee Management***

- All features and functionality of Project 1 should work in Project 2 (DLL, etc.).
- We are not taking away functionality but upgrading with better business object layer mechanism etc.,
- Features & Requirements of PROJECT #1 must be maintained:

  - **Requirement 1(a) – Module-driven Windows Application Using the Three-Layer Scalable Application Architecture**

  - **Requirement 1(b) – Module-driven Windows Application Using the Three-Layer Scalable Application Architecture**

  - **Requirement 1(c) – ABSOLUTELY NO USER-INTERFACE CODE INSIDE THE BUSINESS OBJECT LAYER OR CLASSES (Classes)**

    - In situation in which you need to flag an issue or communicate outside of the objects, `Throw` a specialized Exceptions such as an `ApplicationException` or `NotSupportedException`.

  - **Requirement 1(d) – ABSOLUTELY NO PROCESSING-CODE INSIDE THE USER-INTERFACE LAYER (FORMS & MODULES) (UI code only)**

  - **Requirement 1(e) – Employees Personal Renting of Merchandise Is not allowed**

  - **Requirement 1(f) – Printing in the program is done to A PRINTER FILE that APPENDS data every time it prints. Use File Access Code to manage the printer file. Name the file Network_Printer.txt**

  - **Requirement 1(g) – CREATE FILE ACCESS CODE for 4 ARRAY CLASSES (EmployeeList, CustomerList, DVDList & VideoGameList) to Support Permanent Data Storage to TEXT FILES (EmployeeData.txt, CustomerData.txt, DVDData.tx & VideoGameData.txt)**

  - **Requirement 1(h) – DATABASE LAYER is simulated by Text files (EmployeeData.txt, CustomerData.txt, DVDData.tx & VideoGameData.txt)**

  - **Requirement 1(i) – User-Interface Requirements the same for Login Form, Main Form, Back-End Management Form, Employee Management Form, Customer Management Form, DVD Management Form, VideoGame Management Form & POS Form.**

Requirement #2 (NEW) – No under Aged Employee can be hired. Must be 16 and Older

- This company only hires employee over 16 years old. Implement this policy as follows:

  - If an employee using the system attempts to enter a birth date which makes the employee under 16, the property should `Throw` a specialized Exceptions such as an `ApplicationException` or `NotSupportedException`.
  - This feature should be implemented in the Business Object Layer Employee class, Birthdate Property. More on this below

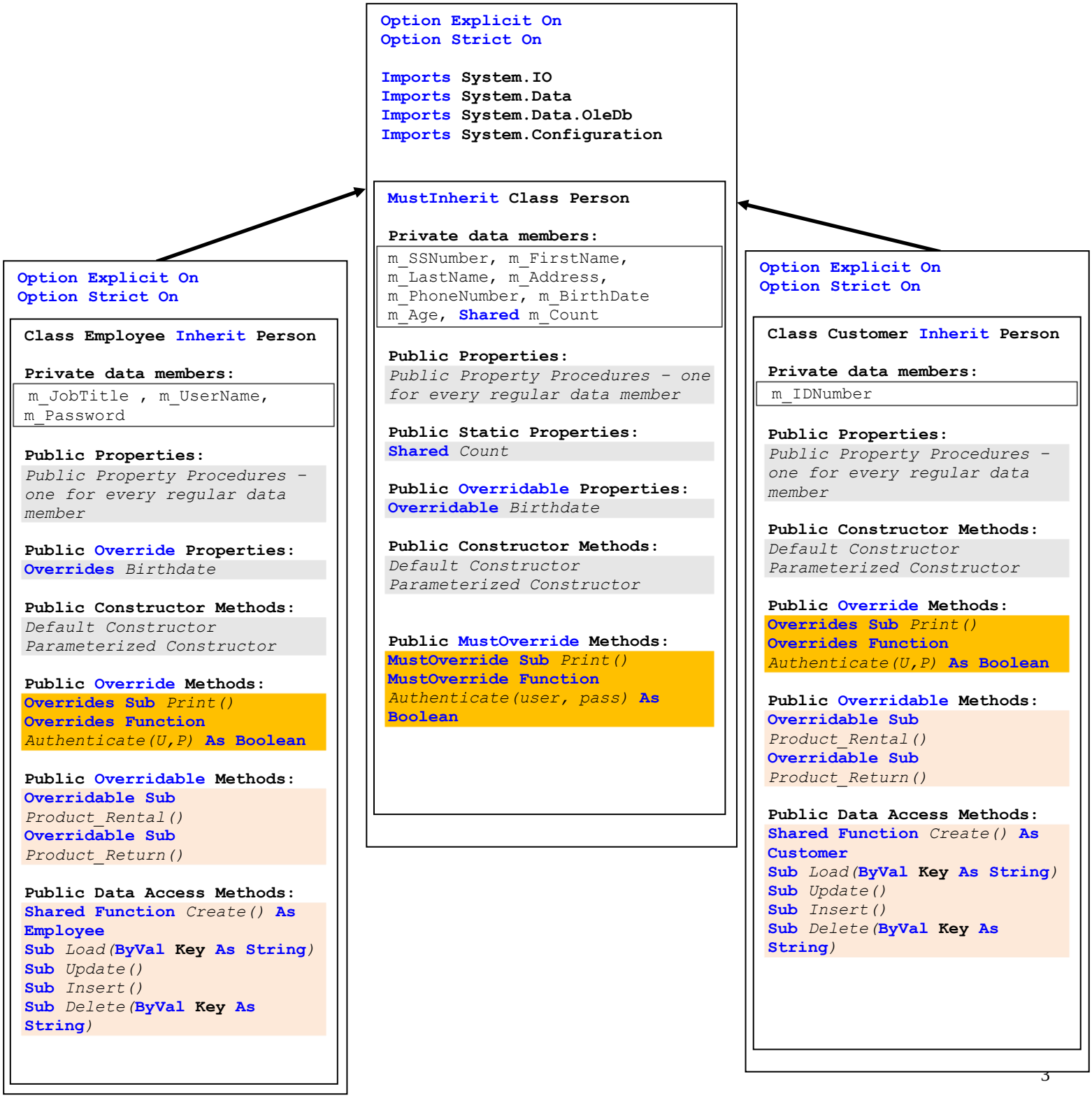
Requirement #3 (UPGRADE) – Implement the following NEW Object Model

- ❑ This section lists the required classes needed to implement this phase of the project.

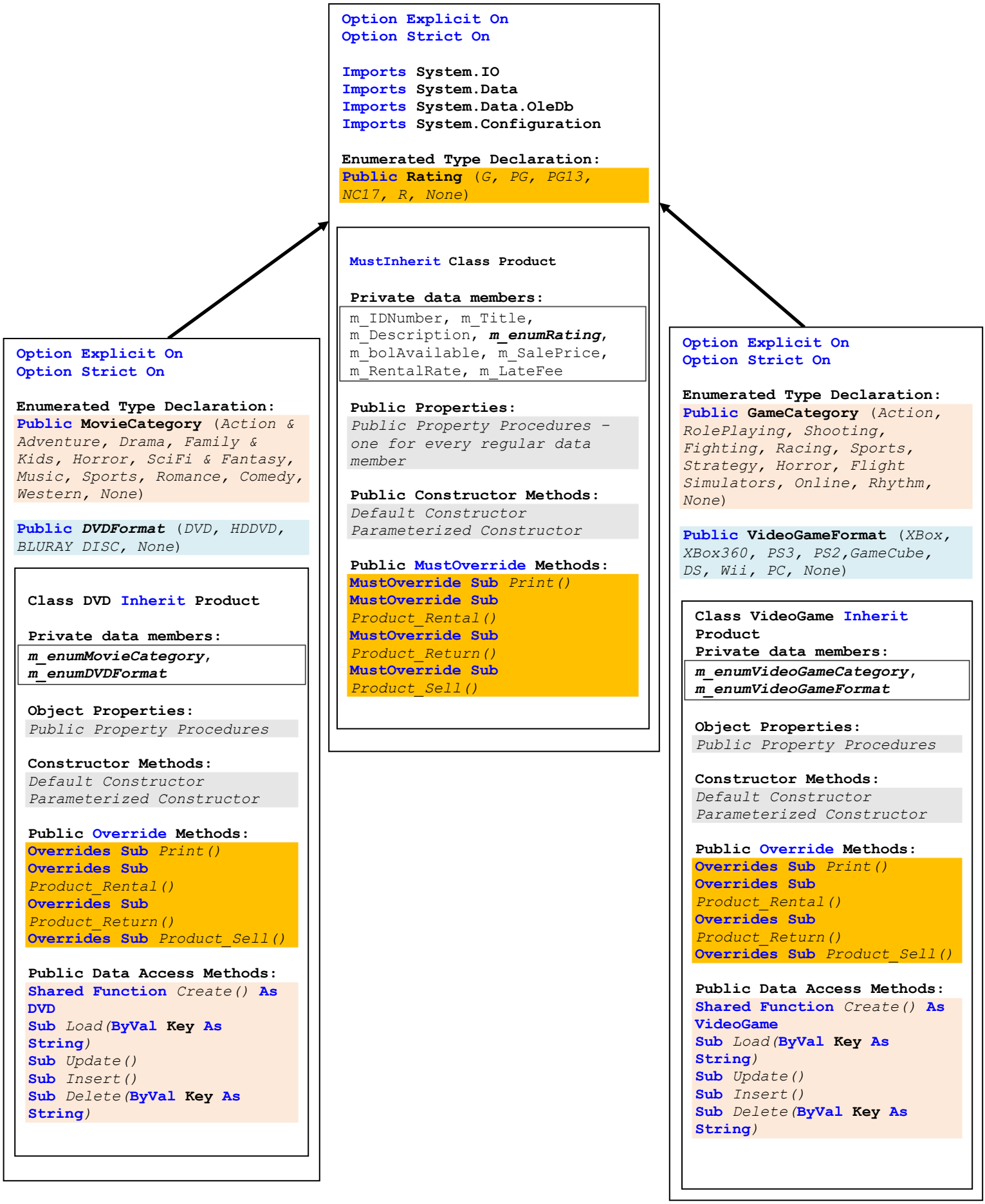
Business Object Layer

Business Object Architecture & Class Object Model:

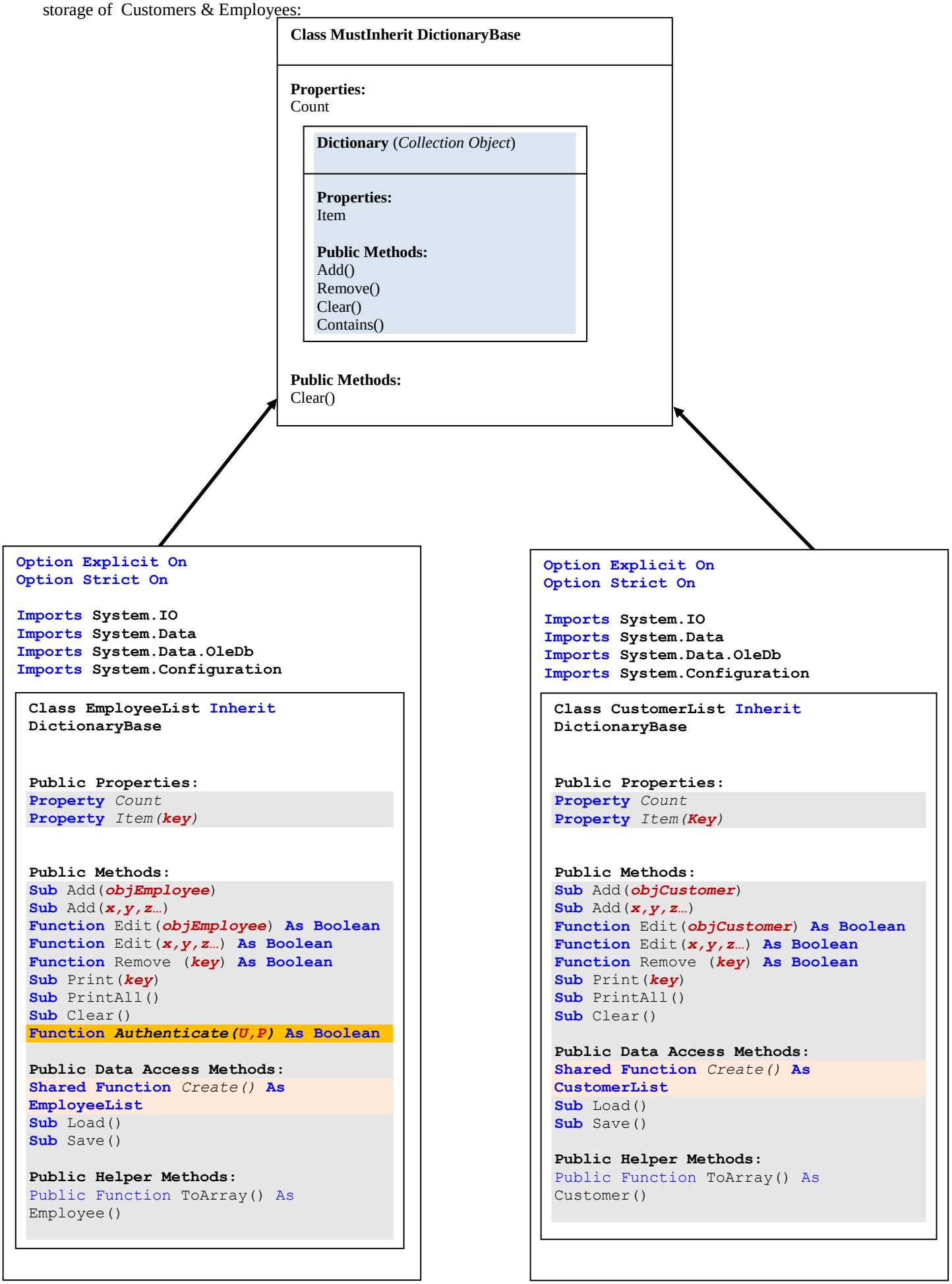
1) The Object-Model requires the following CLASS INHERITANCE hierarchy to represent the ***Person***, ***Employee*** & ***Customer***:



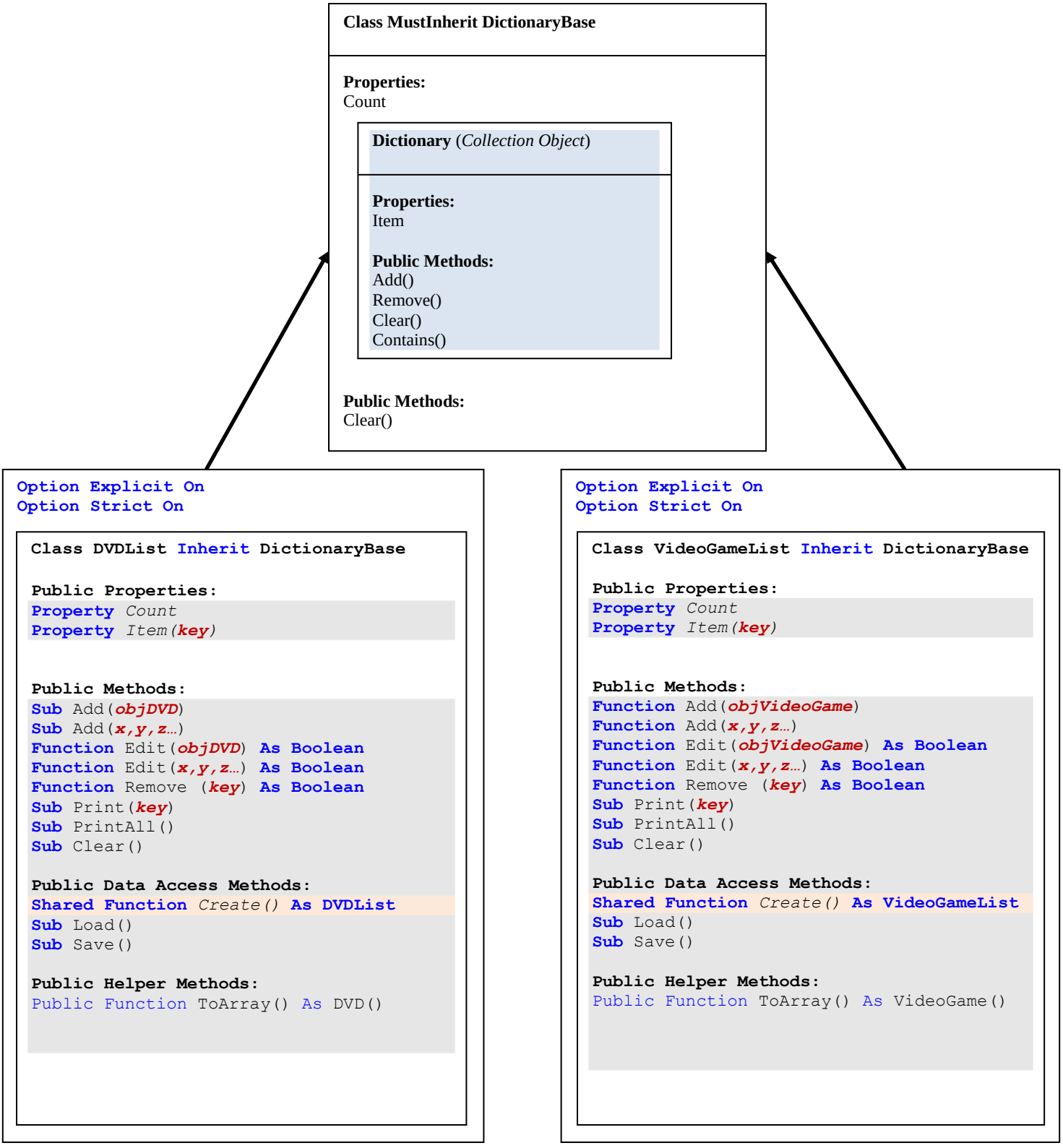
2) The Object-Model requires the following CLASS INHERITANCE hierarchy to represent the ***Product***, ***DVD*** and ***VideoGame***:



3) The Object-Model require the following CUSTOM COLLECTION CLASSES to REPRESENT OUR DATABASE or memory storage of Customers & Employees:



4) The Object-Model require the following CUSTOM COLLECTION CLASSES to REPRESENT OUR DATABASE or memory storage of ***DVD*** and ***VideoGame*** Objects:



Object Model Requirements Summary (CLASS DETAILS IN APPENDIX A)

- As shown in the Object-Model, Class requirements and details:
  - You are required to create these 10 classes: **Employee**, **Customer**, **DVD** and **VideoGame, CustomerList, EmployeeList, DVDList & VideoGameList**
  - **IMPORTANT! Follow this object model to the letter, the methods, properties and data should be NAMED as shown in diagram**.
  - DO NOT CHANGE THE OBJECT MODEL UNLESS YOU HAVE A VALID REASON AND PERMISSION FROM THE PROJECT MANAGER (Prof Rodriguez
  - **NOTE THAT THE BIRTHDATE PROPERTY IN THE EMPLOYEE CLASS IS OVERRIDDEN TO IMPLEMENT THE UNDER 16 YEAR OLD POLICY**.
  - Details on each of the class components ARE LISTED IN THE APPENDIX SECTION OF THIS DOCUMENT
  - **OPTION STRICT & EXPLICIT = ON for all classes.**
  - **APPENDIX SECTION OF THIS DOCUMENT PROVIDES THE DETAILED DESCRIPTION AND DATA, PROPERTIES & METHODS OF EACH OF THE CLASSES. BASE YOU CLASSES ON THE INFORMATION PROVIDED IN THE APPENDIX A**.

Main Classes Summary & Requirements

- Explanation of the ***Person***, ***Employee***, ***Customer***, ***Product***, ***DVD*** & ***VideoGame*** Classes is as follows (**DETAILS IN APPENDIX A**):

Person Class Explanation:

- The ***Person*** Class details are as follows:
  1. `MustInherit` class which will serve as the BASE CLASS used as template for creating all employees and customers
  2. ***Person Object*** data – The Person Class is. It is composed of *first name*, *last name*, *social security number*, *birth date*, *age*, *address & phone*.
  3. The designer of this class realized that in the future, inherited classes may need to ***OVERRIDE*** the some of its properties and methods. So a decision was made to make the **Birthdate** property `Overridable`.
  4. ***Age Calculation*** – The Age is calculated via the BIRTHDATE Property. As the Birthdate Property is SET, the age is determined and the value assigned to its private data.
  5. `MustOverride` Methods – This class also contains two `MustOverride` Methods ***Print()*** & ***Authenticate(U,P)*** to that MUST BE IMPLEMENTED BY derived classes.

Employee Class Explanation:

- The ***Employee*** Class details are as follows:
  1. ***Employee Class*** `Inherits` from ***Person***
  2. ***Employee Class*** **data** – The Employee record is composed of all public properties inherited by the Person class and adds three new data elements: *Job Title*, *Username*, & *Password*. Therefore a full employee record is composed of *first name*, *last name*, *social security number*, *birth date*, *age*, *address*, *phone*, *Job Title*, *Username*, & *Password*
  3. A NEW policy has been implemented that Employees under 16 years of age cannot work in this company. `Overrides` the *Base class* **Birthdate** property in the **Employee** Class and implement this feature. In the Set portion of the property, an ***exception*** should be **thrown** when an under aged employee is being attempted to be set or added to the system, else simply set the property as usual. Note that this exception MUST be trapped in the user-interface/module sections of the program.
  4. ***Print Method()*** – The Employee Object Print Method must be `Overrides` as mandated by Base Class ***Person***. Print should simulate printing by saving the *employee's* data to a TEXT FILE NAMED ***Network_Printer.TXT*** as a COMMA-DELIMITED-STRING.
  5. ***Authenticate Method()*** – In addition, the Employee object should be able to Authenticate itself via an ***Authenticate()*** method, which must be `Overrides` as mandated by Base Class ***Person***. Authenticate take two parameters a user/pass determine authenticity by returning either a **True** or **False**.
  6. ***Product_Rental & Product_Return*** – `Overridable` methods implement the rental process. They are `Overridable` to allow future classes that which to inherit from employee to be able to overrides these methods if desired. Company policy does not allow Employees to Rent; therefore this methods must be created for future use only, BUT DISABLED VIA EXCEPTIONS. Create and leave this method empty with required code to satisfy the compiler & use Throw Statement to disable.
  7. ***Data Access Methods*** – Class contains data access methods to *Create*, *Load*, *Update*, *Insert* & *Delete* Objects from Database. At this time this methods are empty. You are required to create the empty methods for future use.

Customer Class Explanation:

- The ***Customer*** class details are as follows:
  1. ***Customer Class*** `Inherits` from ***Person***
  2. ***Customer Class*** – Customer records is composed of all public properties inherited by the Person class and one new data elements: *IDNumber*. Therefore a customer record is composed of *first name*, *last name*, *social security number*, *birth date*, *age*, *address*, *phone*, and *IDNumber*.
  3. ***Print Method()*** – The Customer Object Print Method must be `Overrides` as mandated by Base Class ***Person***. Print should simulate printing by saving the *employee's* data to a TEXT FILE NAMED ***Network_Printer.TXT*** as a COMMA-DELIMITED-STRING.
  4. ***Authenticate Method()*** – This method is for future use, Customers do not need to authenticate themselves. `Overrides` the method as mandated by Base Class ***Person***, but leave it empty with required code to satisfy the compiler.
  5. ***Product_Rental & Product_Return*** – – `Overridable` methods implement the rental process. They are `Overridable` to allow future classes that which to inherit from employee to be able to overrides these methods if desired, Create and leave this method empty with required code to satisfy the compiler.
  6. ***Data Access Methods*** – Class contains data access methods to *Create*, *Load*, *Update*, *Insert* & *Delete* Objects from Database. At this time this methods are empty. Create and leave this method empty with required code to satisfy the compiler for future use.

Product Class Explanation:

- The ***Product*** Class requirements are as follows:
  1. `MustInherit` class which will serve as the BASE CLASS used as template for creating all ***DVD*** & ***VideoGame*** Classes.
  2. ***Product Object*** **data** – A product record contains the following attributes: *IDNumber, Title, Description, Rating (Enumerated Data Type with pre-defined values) , Available, Sales Price, Rental Rate & Late Fees.*
  3. The designers of this class realize that in the future, inherited classes may need to ***OVERRIDE*** the some of its methods. So a decision was made to make the *Print()* methods ***OVERRIDABLE***
  4. `MustOverride` Methods – This class also contains a `MustOverride` Methods ***Print()*** , ***Product_Rental()***, ***Product_Return()*** & ***Product_Sell()*** that MUST BE IMPLEMENTED BY derived classes.

DVD Class Explanation:

- The ***DVD*** Class requirements are as follows:
  1. ***DVD Class*** `Inherits` from ***Product***
  2. ***DVD Class Data*** – DVD record represents the DVDs to be rented and sold. It is composed of all public properties inherited by the Product class and TWO new data elements: *MovieCategory(Enumerated Data Type with pre-defined values) and DVDFormat(Enumerated Data Type with pre-defined values)*. Therefore a DVD record is composed of: *IDNumber, Title, Description, Rating (Enumerated Data Type with pre-defined values) , Available, Sales Price, Rental Rate & Late Fees* and *MovieCategory and DVDFormat*.
  3. ***Print Method()*** – The DVD Object Print Method must be `Overrides` as mandated by Base Class ***Product***. Print should simulate printing by saving the *DVD's* data to a TEXT FILE NAMED ***Network_Printer.TXT*** as a COMMA-DELIMITED-STRING.
  4. ***Product_Rental, Product_Return & Product_Sell*** – these methods implement the rental & Selling process in future implementation. `Overrides` the method as mandated by Base Class ***Product***, but leave it empty with required code to satisfy the compiler
  5. ***Data Access Methods*** – Class contains data access methods to *Create*, *Load*, *Update*, *Insert* & *Delete* Objects from Database. At this time this methods are empty. Create and leave this method empty with required code to satisfy the compiler for future use.

VideoGame Class Explanation:

- The ***VideoGame*** Class requirements are as follows:
  1. ***VideoGame Class*** `Inherits` from ***Product***
  2. ***VideoGame Class Data*** – DVD record is composed of all public properties inherited by the Product class and TWO new data elements: *GameCategory(Enumerated Data Type with pre-defined values) and VideoGameFormat(Enumerated Data Type with pre-defined values)*. Therefore a DVD record is composed of: *IDNumber, Title, Description, Rating (Enumerated Data Type with pre-defined values) , Available, Sales Price, Rental Rate & Late Fees* and *GameCategory and VideoGameFormat*.
  3. ***Print Method()*** – The Customer Object Print Method must be `Overrides` as mandated by Base Class ***Product***. Print should simulate printing by saving the *employee's* data to a TEXT FILE NAMED ***Network_Printer.TXT*** as a COMMA-DELIMITED-STRING.
  4. ***Product_Rental, Product_Return & Product_Sell*** – these methods implement the rental & Selling process in future implementation. `Overrides` the method as mandated by Base Class ***Product***, but leave it empty with required code to satisfy the compiler
  5. ***Data Access Methods*** – Class contains data access methods to *Create*, *Load*, *Update*, *Insert* & *Delete* Objects from Database. At this time this methods are empty. Create and leave this method empty with required code to satisfy the compiler for future use.

Custom Collection Classes Details and Data Management(CLASS DETAILS IN APPENDIX A):

- Data will be managed in the program by the **4 CUSTOM COLLECTION CLASSES THAT WILL STORE OUR OBJECTS IN MEMORY**. The requirements are as follows:
- These four ***Collection Classes*** will be managed by User-Interface Forms: *Customer*, *Employee* , *DVD & VideoGame Management* Forms (More details in User-Interface section).

- The four ***Collection Classes*** have the following names and functionality:

EmployeeList Class Details:

- The ***EmployeeList*** Class details are as follows:
  1. *EmployeeList i*s inherited from *DICTIONARYBASE LIBRARY*
  2. *EmployeeList* data – An internal ***Collection Object or MYBASE.DICTIONARY*** stores CHILD Employee Objects.
  3. *EmployeeList* has Properties & Methods to get COUNT of objects in List, ADD, REMOVE, EDIT, CLEAR, PRINT, PRINTALL, AUTHENTICATE, LOAD & SAVE.
  4. Objects stored inside this class are loaded and saved via **Save()** & ***Load ()*** methods to data to a TEXT FILE NAMED ***EMPLOYEEDATA.TXT*** as a **COMMA-DELIMITED-STRING**

CustomerList Class Details:

- The ***CustomerList*** Class details are as follows:
  1. *CustomerList* is inherited from *DICTIONARYBASE LIBRARY*
  2. *CustomerList* data – An internal ***Collection Object or MYBASE.DICTIONARY*** stores CHILD Customer Objects.
  3. *CustomerList* has Properties & Methods to get COUNT of objects in List, ADD, REMOVE, EDIT, CLEAR, PRINT, PRINTALL, AUTHENTICATE, LOAD & SAVE.
  4. Objects stored inside this class are loaded and saved via **Save()** & ***Load ()*** methods to data to a TEXT FILE NAMED ***CUSTOMERDATA.TXT*** as a **COMMA-DELIMITED-STRING**

DVDList Class Details:

- The ***DVDList*** Class details are as follows:
  1. *DVDList* is inherited from *DICTIONARYBASE LIBRARY*
  2. *DVDList* data – An internal ***Collection Object or MYBASE.DICTIONARY*** stores CHILD ***DVD*** Objects.
  3. *DVDList* has Properties & Methods to get COUNT of objects in List, ADD, REMOVE, EDIT, CLEAR, PRINT, PRINTALL, AUTHENTICATE, LOAD & SAVE.
  4. Objects stored inside this class are loaded and saved via **Save()** & ***Load ()*** methods to data to a TEXT FILE NAMED ***DVDLIST.TXT*** as a **COMMA-DELIMITED-STRING**

VideoGameList Class Details:

- The ***VideoGameList*** Class details are as follows:
  1. *VideoGameList* is inherited from *DICTIONARYBASE LIBRARY*
  2. *VideoGameList* data – An internal ***Collection Object or MYBASE.DICTIONARY*** stores CHILD ***VideoGame*** Objects.
  3. *VideoGameList* has Properties & Methods to get COUNT of objects in List, ADD, REMOVE, EDIT, CLEAR, PRINT, PRINTALL, AUTHENTICATE, LOAD & SAVE.
  4. Objects stored inside this class are loaded and saved via **Save()** & ***Load ()*** methods to data to a TEXT FILE NAMED ***VIDEOGAMELIST.TXT*** as a **COMMA-DELIMITED-STRING**

- **(CLASS DETAILS IN APPENDIX A)**

(Optional) Data Binding Support for DataGridView CONTROL in LIST Classes

- If you decide to use DATA BINDING to BIND the ***DictionaryBase*** COLLECTION CLASS OBJECTS (ex. objCustomerList) to A DATAGRIDVIEW CONTROL, note that the ***DictionaryBase*** Base class DOES NOT SUPPORT DATA BINDING DIRECTLY.
- In other words you cannot do the following:

```
'Assuming objCustomerList is an OBJECT of a DICTIONARY COLLECTION CLASS
'WE CANNOT BIND DICTIONARY BASE COLLECTION CLASS TO CONTROLS AS FOLLOWS:
 dataGridCustomerList.DataSource ≠ objCustomerList
```

Solution

- Options: Either or

1) Choose another type of COLLECTION Class from the .NET Library such as ***CollectionBase*** or use ARRAY for your project
2) Or Find a workaround

- In our case, the ***DictionaryBase*** is the ideal Collection for our application, so we will choose OPTION 2 and FIND A WORKAROUND.
- The workaround is:

  - CONVERT THE DICTIONARY COLLECTION TO AN ARRAY AND BIND THE ARRAY TO THE CONTROL..
  - Fortunately, the ***DictionaryBase*** Class ***MyBase.Dictionary*** OBJECT has a built-in Method named **CopyTo()** That convert a **DICTIONARY COLLECTION** to an **ARRAY**:

```
MyBase.Dictionary.Values.CopyTo()
```

  - Therefore we will create a Method in our CUSTOM COLLECTION CLASS named `CopyToArray()` that will do the work for us.
  - For example, the code for the ***CustomerList*** Class is as follows:

```
Public Function ToArray() As Customer()
    'Declare an empty array of Customers
    Dim arrCustomerList(MyBase.Dictionary.Count - 1) As Customer

    'Use CopyTo() Method of Dictionary Class to convert Collection to Array
    MyBase.Dictionary.Values.CopyTo(arrCustomerList, 0)
    Return arrCustomerList
End Function
```

- Use this method and modified as needed for all our LIST CLASSES (***EmployeeList***, ***CustomerList***, ***DVDList***, ***VideoGameList***) as shown in object model

  - **(CLASS DETAILS IN APPENDIX A)**

Using Binding in the Forms:

- To use the Binding in the LIST EVENT-HANDLER OF THE LIST FORMS simply do the following:

```
'Optional- refresh the data grid
dgCustomerList.Refresh()

'Bind Custom Array Object to DataGrid
dgCustomerList.DataSource = objCustomerList.ToArray()
```

- And that is all the code you need in the :

Requirement #10 – Data Storage (Database Simulation)

Database Layer

- In this version of the application will not actually contain a database program such as MS Access or SQL Server, but we will simulate the database using a combination of **COMPUTER MEMORY** as temporary storage and **FILES** as permanent storage devices.
- The data management is to be simulated in memory by **4 CUSTOM-COLLECTION CLASS Objects**.
- The File management is simulated by **4 COMMA-DELIMITED text FILES**.

Memory Storage –Collection:

- Create *four* GLOBAL COLLECTION CLASS **Objects** of the Data Management Classes required by the object model in the MAIN PROGRAM module. Name the OBJECTS as follows:

  - **objEmployeeList** – Management for all *Employee Objects*: This is an Object of the **EmployeeList** Class.
  - **objCustomerList** – Management for all *Customer Objects*: This is an Object of the **CustomerList** Class.
  - **objVideoGameList** – Management for all *VideoGame objects*: This is an Object of the **VideoGameList** Class.
  - **objDVDList** –Management for all *DVD objects*: This is an Object of the **DVDList** Class

  - Note that THESE OBJECTS ARE CREATED IN THE MAIN MODULE AS PUBLIC IN THE DECLARATION SECTION OF THE Module, so they are accessible to the entire project.

Simulated Database Layer -File Storage:

- Using Collections to manage object in memory is ideal for temporary storage and managing the object while the program is executing, but using the collection to store data permanently is ineffective because it has the disadvantage that data is lost when the program execution stops or power is turned off.
- Using files we can permanently store our data from memory.

Text Files and Collection Class Methods to Load and Save Data

- According to the OBJECT MODEL, the **Collection Classes**, contain two methods to get and save data to file: ***Load()*** & ***Save()***.
- These methods will load and save to the following files:

  - **EmployeeData.txt**. – Managed by **objEmployeeList**.*Load()* & **objEmployeeList**.*Save()*
  - **CustomerData.txt**. – Managed by **objCustomerList**.*Load()* & **objCustomerList**.*Save()*
  - **VideoGameData.txt**. – Managed by **objVideoGameList**.*Load()* & **objVideoGameList**.*Save()*
  - **DVDData.txt**. – Managed by **objDVDList**.*Load()* & **objDVDList**.*Save()*

- These Files should be managed by the Load() & Save() methods of the COLLECTION CLASSES as follows:

  1. **Save()** :
     - Method should save the data as **a COMMA-DELIMITED** list. This means that the content of each object should be listed as a **comma-delimited line**.
     - The files should be **OVERWRITTEN** every time a saved is performed, thus eliminating duplicate data. DO NOT APPEND the data.
     - Use the appropriate **VB.NET System.IO library** class to SAVE the data to file, such as the **StreamWriter** class.
     - Call the COLLECTION.SAVE() when the MANAGEMENT FORMS are CLOSED in the FORM_CLOSED() EVENT-HANDLER.

2. **Load()** :
    - Note that this method should read each of the **COMMA-DELIMITED** lines and **PARSE** accordingly, thereafter setting a temporary OBJECT with this data and adding it to the collection. Note that parsing is a very important and popular operation in computer programming. This means taking a delimited string and extracting the data within each of the comma-delimiter. Note that your CS608 review notes at the beginning of the semester contain information on parsing strings
    - Use the appropriate **VB.NET System.IO library** class to READ the data from file, such as the **StreamReader** class
    - In your program, call the COLLECTION.LOAD() in the LOAD() EVENT-HANDLER of the MANAGEMENT FORMS

- By implementing the ***Save()*** and ***Load()*** methods in the collection to use FILES, data will always be available to the program even after the program is closed or computer turned off.

Requirement #11 – User-Interface Requirements (Same as Project 1- Few or zero changes)

Presentation/User-Interface Layer

User Interface (Overview) (FORM DETAILS IN APPENDIX B)

- THE USER INTERFACE LAYER IS THE CODE that includes the WINDOWS FORMS & MODULE code.
- Create the USER INTERFACE SCREENS using WINDOWS FORMS.
- The will need the following screens:

a) ***Login Screen*:**

- Interface that will allow access to system (Already done in HW4)

b) ***Main Video Management Screen*:**

- Offer user the following three options: ***POS System***, ***Back-End Management Options*** & ***Exit*** (Return to Login Form)

c) ***POS Screen*:**

- Provides users with the following Point-of-Sales options: ***Rental***, ***Return***, ***Customer Registration***, ***Customer Information***, & ***Exit*** requests.
- YOU ARE NOT TO IMPLEMENT THESE FEATURES. Only create the screen but do not implement the functionalities.
- Only implement the ***Exit***, so that it navigates to the *main screen*

d) ***Back-End Management Screen*:**

- Displays the following ***4 Management*** transactions options: ***Employee Management***, ***Customer Management***, ***VideoGame Management***, ***DVD Management*** and ***Exit*** (Return back to *Main screen*) Selecting either of these options should invoke the appropriate screen or exit.

e) ***Employee Management***:

- Interface that will allow users to manage the *Employees*.
- Includes functionality for ADD, SEARCH, EDIT, REMOVE, PRINT, PRINT ALL and LIST ALL *employees*.
- Invoked from the ***Back-End Management Options***. Includes ***Exit*** button

f) ***Customer Management*:**

- Interface that will allow users to manage the *Customers*.
- Includes functionality for ADD, SEARCH, EDIT, REMOVE, PRINT, PRINT ALL and LIST ALL *customers*.
- Invoked from the ***Back-End Management Options***. Includes ***Exit*** button

g) ***DVD Management*:**

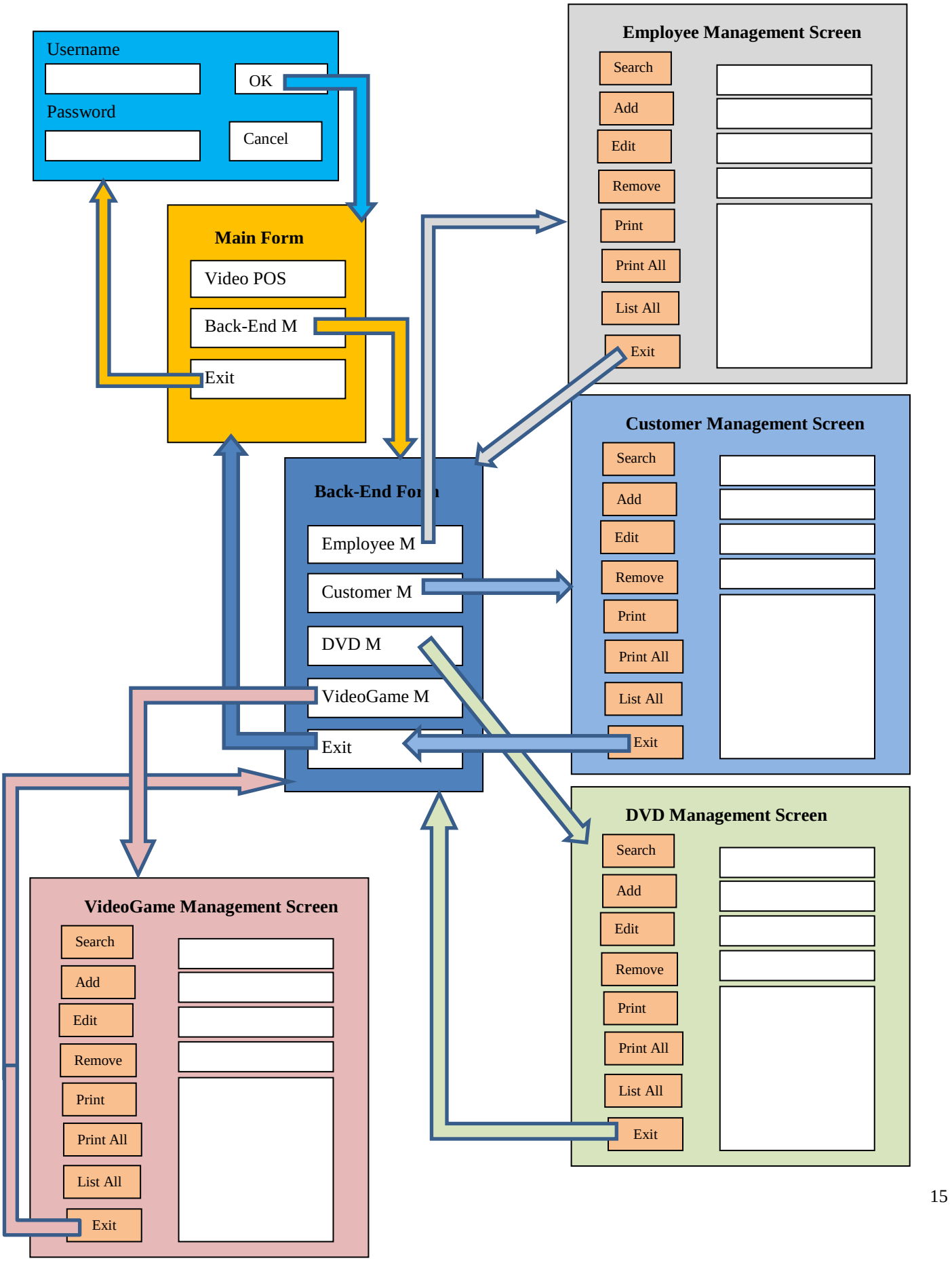
- Interface that will allow users to manage the *DVDs*.
- Includes functionality for ADD, SEARCH, EDIT, REMOVE, PRINT, PRINT ALL and LIST ALL *DVDs*.
- Invoked from the ***Back-End Management Options***. Includes ***Exit*** button

h) ***VideoGame Management*:**

- Interface that will allow users to manage the *VideoGames*.
- Includes functionality for ADD, SEARCH, EDIT, REMOVE, PRINT, PRINT ALL and LIST ALL *VideoGames*.
- Invoked from the ***Back-End Management Options***. Includes ***Exit*** button

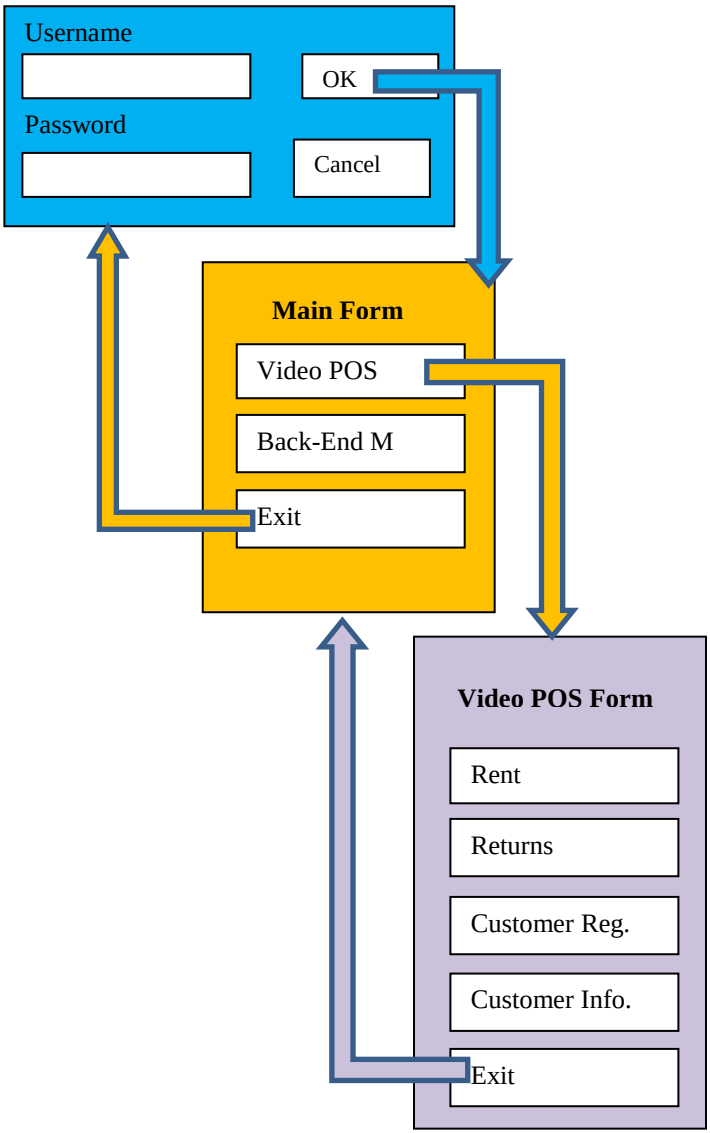
User-Interface Windows Forms & NAVIGATION-FLOW for Back-end Management System

- The DIAGRAM below shows the USER-INTERFACE & Navigation FLOW for the back-end management screens:



User-Interface Windows Forms & NAVIGATION-FLOW for POS System

- The DIAGRAM below shows the USER-INTERFACE & Navigation FLOW for the POS screens:



User-Interface Windows Forms Details (FORM DETAILS IN APPENDIX B)

- You will need a total of 8 WINDOW FORMS or SCREENS..
- YOU HAVE FREEDON AS TO WHAT TYPE OF FORMS TO USE AND WHAT DESIGN TO APPLY WHEN CREATING THE FORMS, WHAT CONTROLS TO USE, ETC.., but YOU MUST FOLLOW THE REQUIREMENTS FOR THE FUNCTIONALITY OF EACH FEATURE.
- *****SEE APPENDIX B FOR DETAIL DESCRIPTION ON FORM CODE & REQUIRED CONTROLS*****
- Below are the details for all the screens.

Employee Login Screens:

- This is the screen the employee will see after the main screen if they wish to use the Video Management System. The screen has the following requirements:

  - ***Provides controls for user to enter Username***
  - ***Provides controls for user to enter Username***
  - ***Provides opton to initiate the login (OK)***
  - ***Provides option to clear all text boxes (Cancel)***

Main Video Management Screen:

- This is the main screen to the program. This is what the employee will see when they are in the system after authenticating. The screen has the following requirements:

  - ***Video POS System Option:*** Selecting this option will take you to the Point-of-Sales Screen
  - ***Back-End Management Options:*** Selecting this option will take you to the Back-End Management screens (described below)
  - ***Exit Option:*** User is ready to exit and return to Login Screen

Video POS Screen:

- (IMPLEMENTED IN FUTURE VERSIONS The Video Point-Of-Sales Screen is where the Employee can process ***Rental***, ***Return***, ***Customer Registration***, ***Customer Information*** & ***Exit*** requests. These features will be implemented in future versions of the program. In this phase you are only to create the ***Video Point-Of-Sales*** screen, with the controls for selecting each option. Only the ***Exit*** option should have code to navigate back to the *Main Screen*. The screen has the following requirements:

  - ***Rent:*** Selecting this option will allow employee to perform rental transaction (Note, this may require other user interfaces to interact with user to handle the request).
  - ***Return:*** Selecting this option will allow employee to perform rental transaction. (Note, this may require other user interfaces to interact with user to handle the request)
  - ***Customer Information:*** Selecting this option will display a customer record (Note, this may require other user interfaces to interact with user to handle the request).
  - ***Customer Registration:*** Selecting this option will register a new customer (Note, this may require other user interfaces to interact with user to handle the request).
  - ***Exit Option:*** User is ready to exit and return to the previous screen.

Backend Management Screen:

- This is the screen employee will use when performing back end management features such as add, remove etc., customers, employees, checking & savings accounts. The screen has the following requirements:

  - ***Employee Management:*** Selecting this option will display the *Employee Management* screen.
  - ***Customer Management:*** Selecting this option will display the *Customer Management* screen.
  - ***DVD Management:*** Selecting this option will display the *Checking Account Management* screen.
  - ***Video Game Management:*** Selecting this option will display the *Savings Account Management* screen.
  - ***Exit Option:*** User is ready to exit and return to the previous screen.

- *****SEE APPENDIX B FOR DETAIL DESCRIPTION ON FORM CODE & REQUIRED CONTROLS*****

MODULE Code Details (User-Interface)

❑ Other USER INTERFACE CODE (MODULE):

- This is the code found in the MODULE or the user interface section.
- This section contains the main method. In addition to the main method you should have the following methods which make up your user interface:

1. IN THE DECLARATION SECTION OF THE MODULE, Create *four* **Objects** of the COLLECTION CLASSES. Name the OBJECTS as follows

    - **objEmployeeList** – Management for all employee objects: Object of the **EmployeeList** Class.
    - **objCustomerList** – Management for all customer objects: Object of the **CustomerList** Class.
    - **objVideoGameList** – Management for all VideoGame objects: Object of the **VideoGameList** Class.
    - **objDVDList** –Management for all DVD object: Object of the **DVDList** Class

2. **Sub Main()** – Controls program execution, loops, performs authentication process code, displays main screen if access granted, else message of "Access denied"
3. **Authenticate()** – Function Method that performs the actual authentication by searching **objEmployeeList** object by calling it's ***Authenticate(U,P)*** method to do the work. From the results of this method call, return a TRUE if found or FALSE otherwise
    - REQUIREMENT! Note that prior to searching the ***objEmployeeList***, this method should load or populate the array with data from file by calling ***objEmployeeList.Load()*** method. When completed, this method should clear the ***objEmployeeList*** by calling the ***objEmployeeList.Load() ()*** method.

Application Processing Flow Description:

Authentication & Main Program Requirements:

1) The following is a description of the functionality of the login and management screens transactions:

1. Program prompt for *UserName* & *Password* using ***Login Form***
2. When values are entered authenticate or interrogate the Employee **database**:
    - If NOT FOUND, prompt user and reject access and returns to step 1.
    - Else proceed to next step
3. Loads ***Main Screen*** if access granted.
4. The ***Main Screen*** will allow navigation to the **Video POS**, **Back-End Management** & **Exit** (Return to Login Screen:
5. The ***Video POS*** will allow navigation to the options: *Rental*, *Return*, *Customer Registration*, *Customer Information*, & *Exit* requests. Only *Exit* should be implemented so that it returns to the Main Form.
6. The ***Back-End Management*** Screen will allow navigation to the **Customer Management**, **Employee Management**, **DVD Management**, & **VideoGame Account Management** screens. Each of the management screens will exit back to the ***Back-End Management screen***.

Requirement #12 – Authentication Requirements

Access to Backend Management & Video System POS System:

1) Customers DO NOT have access to the ***Video System Point-of-Sales*** or ***Back-End Management*** systems or need to interact with it. Only Authorized store employees via a username and password can have access to the system.
2) The login requirements for the BACK-END MANAGEMENT & POINT-OF-SALES systems are SIMILAR to the HWs in class. Here is a summary:

Authentication Program Flow

- A Login Screen should display and prompts the employee for his/her username/password. If the username/password is invalid, the system should not allow access and display a message box stating "*Access denied*" and continue to request the login screen. Only until a valid username/password is entered, should the program proceed and display the "***Video Management System screen***", otherwise access to system is **NOT allowed!**
- The program should continuously loop forever, either displaying the "***Video Management System screen***" allowing employees to use the ***Back-end management*** & ***Point-of-Sales systems*** or a message box "***Access Denied***".
- The program can only end using the following secret username & passwords: User = "1", Pass = "-1", at this point the program loop should terminate, thus ending the program.
- Authentication is done by searching a CUSTOM CLASS ARRAY OBJECT named ***EmployeeList***
- This ***EmployeeList*** Object stores the employee's records and interrogating each of the *Employee Objects* to validate their username & password via each object's ***authenticating mechanism***. This process must be handled by a special method inside the *Employee Object* named ***Authenticate()*** in the main program (more details later).
- SO FAR THESE REQUIREMENTS ARE MET BY YOUR PREVIOUS HWS WITH THE EXCEPTION THAT THE MAIN AUTHENTICATE CALLS THE ***EmployeeList.Authenticate(U,P)*** to do the work and return to it a **true** or **false (See object model below)**.

Administrative Employee Object

- The program should contain a built-in employee account with *username/password* ***admin/999***. This administrative password is available when the system first goes live or into production in order to allow access into the system for the first time. We assume when the system goes live, the database is empty, therefore employees, customers, Savings Account and Checking Accounts will be entered as business progresses.
- We need a built-in account that will allow access to the system the first time the system is started; this is the purpose of ***admin/999***. This administrative Employee Object should be available every time the program runs. Populating this object will be discussed below

APPENDIX A – CLASS DETAILS

Detailed Class Descriptions:

Class Person

- Represents a Person. Intended to be used as a base class
- The key properties/methods are described as follows:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

General Class Information	Description
`Public MustInherit Person`	▪ `MustInherit` Base Class – Designed for inheritance only. No objects of this class should or can be created. ▪ Represents the Persons to be either employees or customers to the business.

Private Data	Description
`Private` m_SSNumber	Purpose: Represents person's social security number. Data type: `String`
`Private` m_FirstName	Purpose: Represents person's name. Data type: `String`
`Private` m_LastName	Purpose: Represents person's last name. Data type: `String`
`Private` m_Address	Purpose: Represents person's address. Data type: `String`
`Private` m_Phone	Purpose: Represents person's phone number. Data type: `String`
`Private` m_BirthDate	Purpose: Represents person's birth date. Data type: `Date`
`Private` m_Age	Purpose: Represents person's age. Data type: `Integer`
`Private Shared` m_Count = 0	Purpose: `Shared` variable use to keep count of customer objects created. Private data is initialized to 0 on creation. Data type: `Integer`

Public Properties (GET/SET)	Description
`Public` FirstName	GET/SET **m_FirstName** private data
`Public` LastName	GET/SET **m_LastName** private data
`Public` SSNumber	GET/SET **m_SSNumber** private data
`Public` Address	GET/SET **m_Address** private data
`Public` Phone	GET/SET **m_Phone** private data
`Public Overridable` BirthDate	▪ Declared `Overridable` to allow derived classes to override ▪ SETS & GETS the **m_BirthDate** private data ▪ **Also Calculates the Age based on Birthdate value and today's date as follows:** ○ Calculates the age by subtracting Today's date from the Birthdate Value being SET. ○ Assigns the calculated age to the ***m_Age*** private data or property.
`Public ReadOnly` Age	GET **m_Age** private data
`Public Shared` Count	`Shared` GET/SET **m_Count** private data

Constructors	Parameters	Return Type	Description
`Public New()`	None	`N/A`	▪ **Default Constructor**. ▪ Should initialize the private data members with appropriate default values. The data ***m_Count*** is incremented only. ▪ Sets to appropriate default values: - m_SSNumber, m_FirstName, m_LastName =”” - m_Birthdate = #1/1/1900# - m_Address , m_PhoneNumber =”” - m_Age = 0
`Public New(x, y, z etc..)`	▪ A parameter to Set each of the following **Properties** only (AGE & COUNT not included): - SSNumber - FirstName - LastName - Birthdate - Address - PhoneNumber ▪ Should have a total of **6** parameters: - **par1 to Par6** ▪ Name the parameters as you see fit. ▪ All parameters are Pass-by-Value	`N/A`	▪ Parameterized Constructor 1) Sets parameter list to all data members via **PUBLIC PROPERTIES** EXCEPT the Username, Password, Age & COUNT. 2) Match parameter list appropriately: - SSNumber = **par1** - FirstName = **par2** - LastName = **par3** - Birthdate = **par4** - Address = **par5** - PhoneNumber = **par6** 3) Age is handled by Birthdate Property, no need to address in the constructor

Public MustOverride Methods	Parameters	Return Type	Description
`Public MustOverride Sub Print()`	None	`None`	▪ `MustOverride` **Print()** method. ▪ Intended for derived classes to print their data ▪ **Declaration only**. Must be implemented in derived classes.
`Public MustOverride Function Authenticate(user ,pass)`	`String user, String pass`	`True` `False`	▪ `MustOverride` **Authenticate(u,p)** method. ▪ Intended for derived classes to authenticate themselves ▪ **Declaration only**. Must be implemented in derived classes.

Class Employee

- Represents the employees of the company.
- The key properties/methods are described as follows:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

General Class Information	Description
`Public class Employee` `Inherits Person`	Inherits from Person class

Private Data	Description
`Private` m_JobTitle	Purpose: stores employee's job title. Data type: `String`
`Private` m_UserName	Purpose: stores employee's username. Data type: `String`
`Private` m_Password	Purpose: stores employee's password. Data type: `String`

Public EVENT	Description
Event SecurityAlert(ByVal username, ByVal password)	▪ Event to handle any SECURITY ALERTS related with an Employee. ▪ Trigger or raise the event in the following method: - *Trigger* or ***raise*** this event ONLY inside the *Authenticate()* method **prior** to the verification of the username & password. - Pass into the RAISEEVENT call the parameters **username** & **password** passed into the method.

Public Properties (GET/SET)	Description
`Public` JobTitle	GET/SET **m_JobTitle** private data
`Public` UserName	GET/SET **m_UserName** private data
`Public` Password	GET/SET **m_Password** private data
`Public Overrides` `Birthdate`	`Overrides` the Birthdate property in order to SET/GET the BASE CLASS Birthdate PROPERTY and implement the following company policy: ▪ SETS & GETS the m_ Birthdate private data from Base class ▪ Implement the following company policy: ○ Under aged employees cannot work for this company. Employees under 16 years of age cannot work in this business. ○ **Validate that employee MUST BE 16 YEARS OF AGE or older based on Birthdate value and today's date otherwise THROW AND EXCEPTION**. ○ **Leverage the Person.Birthdate property in order to store the date & set the age**.

Constructors	Parameters	Return Type	Description
`Public New()`	None	`N/A`	▪ **Default Constructor**. ▪ Should initialize the private data members ▪ Sets to appropriate default values: - m_JobTitle =”” - m_Username =”” - m_Password =”” ▪ **INCREMENTS the Person Class `Shared` *Count*.**
`Public New(x, y, z etc..)`	▪ A parameter to Set each of the following Employee Class & Person Base Class **Properties**: - SSNumber - FirstName - LastName - Birthdate - Address - PhoneNumber - JobTitle - Username - Password ▪ Should have a total of **7** parameters: - **par1 to Par7** ▪ Name the parameters as you see fit. ▪ All parameters are Pass-by-Value	`N/A`	▪ **Parameterized Constructor** ▪ Sets the **PROPERTIES** of the Person Base Class and this Employee Class matching parameter list. ▪ The data ***m_Count*** is incremented only 1. Calls **Base Class Parameterized Constructor** to handle the following parameters: - SSNumber = **par1** - FirstName = **par2** - LastName = **par3** - Birthdate = **par4** - Address = **par5** - PhoneNumber = **par6** 2. Sets the **PROPERTIES** of this class with the following parameters: - JobTitle = **par7** 3. The **Username** & **Password** are NOT part of the parameters and should be defaulted as follows: - Username = **“”** - Password = **“”** 4. **INCREMENTS the Person Class `Shared` *Count*.**

Public Overrides Methods	Parameters	Return Type	Description
`Public Overrides Sub Print()`	None	`None`	▪ **Implementation of Print()** method. ▪ The *Print()* Method WRITES ALL OBJECT'S DATA TO THE PRINTER FILE as follows: **1.** Opens ***Network_Printer.txt*** file for APPENDING. **2.** Write each object's property/data in the following FORMAT: Printing ***Employee*** ..... ID Number = ***value*** Name = ***value*** Social Security = ***value*** Birthday = ***value*** Address = ***value*** Age = ***value*** Etc…. **3.** Close the file ▪ Add Error-Handling code using `try-catch-finally` to handle all required exceptions for any file access, array or general exceptions ▪ Follow best practice of trapping for unexpected general errors in addition to specific errors ▪ Use `throw` statement to re-throw all exceptions
`Public Overrides Function Authenticate(user, pass)`	`String user, String user`	`True` `False`	▪ Method to Authenticate Employee Username & Password as follows: 1) *Trigger* or ***raise*** this **SecurityAlert(U,P)** Event **prior** to the verification of the username & password 2) *Process:* Compares each of the TWO parameters (U, P) values to the private ***m_username*** & ***m_password*** variables 3) **Returns** a **TRUE** if both of these values match, otherwise it returns a **FALSE**. ▪ **The objective of this function is to AUTHENTICATE THE EMPLOYEE OBJECT ITSELF**.

Public Overridable Methods	Parameters	Return Type	Description
`Public Overridable Sub Product_Rental()`	None	`None`	▪ **Implementation of Product_Rental()** method. ▪ `Overridable` so any future derived classes can override. ▪ Handles future product rental transactions for an employee who wishes to rent. ▪ NO IMPLEMENTATION, STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY. ▪ **This is a VIOLATION OF CURRENT COMPANY POLICY.** ▪ **MUST DISABLED WITH Use `Throw` NotSupported Exception statement**
`Public Overridable Sub Product_Return()`	None	`None`	▪ **Implementation of Product_Return()** method. ▪ `Overridable` so any future derived classes can override. ▪ Handles future product returns transactions for an employee who wishes to rent. ▪ NO IMPLEMENTATION, STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY. ▪ **This is a VIOLATION OF CURRENT COMPANY POLICY.** ▪ **MUST DISABLED WITH Use `Throw` NotSupported Exception statement**

Public Data Access Methods	Parameters	Return Type	Description
`Public Shared Function Create()`	`None`	`Employee Reference or Pointer`	▪ **Implementation of `Shared Function Create()`** method. ▪ OBJECT FACTORY METHOD. ▪ Creates & RETURNS FULLY READY AND INIALIZED EMPLOYEE OBJECTS. - Objects created may include initialized data from database. ▪ STUB FUNCTION METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY & RETURN KEYWORD.
`Public Sub Load(key)`	`String key`	`None`	▪ **Implementation of Load()** method. ▪ Intended for this class objects to be able to LOAD themselves from database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Update()`	None	`None`	▪ **Implementation of Update()** method. ▪ Intended for this class objects to be able to UPDATE or save their data to database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Insert()`	None	`None`	▪ **Implementation of Insert()** method. ▪ Intended for this class objects to be able to ADD or insert a new record to database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Delete(key)`	`String key`	`None`	▪ **Implementation of Delete()** method. ▪ Intended for this class objects to be able to DELETE or remove its record from database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.

Class Customer

- Represents the customer.
- The Customer class has the following key properties/methods:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

General Class Information	Description
`Public class Customer Inherits Person`	Inherits from Person class

Private Data	Description
`Private` m_IDNumber	Purpose: stores customer's ID number. Data type: `String`

Public Properties (GET/SET)	Description
`Public` IDNumber	GET/SET **m_IDNumber** private data

Constructors	Parameters	Return Type	Description
`Public New()`	None	`N/A`	▪ **Default Constructor**. ▪ Should initialize the private data members with appropriate default values for this ***Customer*** Class and the ***Person*** Base Class. ▪ Calls **Base Class Default Constructor** ▪ The data ***m_Count*** is incremented only ▪ Sets to appropriate default values: - m_IDNumber ="" ▪ **INCREMENTS the Person Class `Shared` *Count*.**
`Public New(x, y, z etc..)`	▪ A parameter to Set each of the following Employee Class & Person Base Class **Properties**: - IDNumber - FirstName - LastName - SSNumber - Birthdate - Address - PhoneNumber ▪ Should have a total of 7 parameters: - **par1 to Par7** ▪ Name the parameters as you see fit. ▪ All parameters are Pass-by-Value	`N/A`	▪ **Parameterized Constructor** ▪ Sets the **PROPERTIES** of the ***Person*** Base Class and this ***Customer*** Class matching parameter list. ▪ The data ***m_Count*** is incremented only ▪ Calls **Base Class Parameterized Constructor** to handle the following parameters: - FirstName = **par2** - LastName = **par3** - SSNumber = **par4** - Birthdate = **par5** - Address = **par6** - PhoneNumber = **par7** ▪ Sets the **PROPERTIES** with the following parameters: - IDNumber = **par1** ▪ **INCREMENTS the Person Class `Shared` *Count*.**

Public Overrides Methods	Parameters	Return Type	Description
`Public Overrides Sub Print()`	None	`None`	▪ **Implementation of Print()** method. ▪ The *Print()* Method WRITES ALL OBJECT'S DATA TO THE PRINTER FILE as follows: **1.** Opens ***Network_Printer.txt*** file for APPENDING. **2.** Write each object's property/data in the following FORMAT: Printing ***Customer*** ..... ID Number = ***value*** Name = ***value*** Social Security = ***value*** Birthday = ***value*** Address = ***value*** Age = ***value*** Etc…. **3.** Close the file ▪ Add Error-Handling code using `try-catch-finally` to handle all required exceptions for any file access, array or general exceptions ▪ Follow best practice of trapping for unexpected general errors in addition to specific errors ▪ Use `throw` statement to re-throw all exceptions
`Public Overrides Function Authenticate(user, pass)`	`String user, String user`	`True False`	▪ **Implementation of Authenticate(u,p)** method. ▪ **Company Policy does not dictate Customers need to authenticate. Create the method to satisfy the COMPILER, but do not implement this method.** ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY & RETURN KEYWORD.

Public Overridable Methods	Parameters	Return Type	Description
`Public Overridable Sub Product_Rental()`	None	`None`	▪ **Implementation of Product_Rental()** method. ▪ `Overridable` so any future derived classes can override. ▪ Handles future product rental transactions for an employee who wishes to rent. ▪ NO IMPLEMENTATION, STUB METHOD FOR FUTURE UPGRADE. ▪ **Create the method to satisfy the COMPILER, but do not implement this method** ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Overridable Sub Product_Return()`	None	`None`	▪ **Implementation of Product_Return()** method. ▪ `Overridable` so any future derived classes can override. ▪ Handles future product returns transactions for an employee who wishes to rent. ▪ NO IMPLEMETATION, STUB METHOD FOR FUTURE UPGRADE ▪ **Create the method to satisfy the COMPILER, but do not implement this method**. ▪ CREATE THE HEADER WITH AN EMPTY BODY.

Public Data Access Methods	Parameters	Return Type	Description
`Public Shared Function Create()`	`None`	`Customer Reference or Pointer`	▪ **Implementation of `Shared Function Create()`** method. ▪ OBJECT FACTORY METHOD. ▪ Creates & RETURNS FULLY READY AND ANIMALIZED CUSTOMER OBJECTS. - Objects created may include initialized data from database. ▪ STUB FUNCTION METHOD FOR FUTURE UPGRADE. ▪ **Create the method to satisfy the COMPILER, but do not implement this method** ▪ CREATE THE HEADER WITH AN EMPTY BODY & RETURN KEYWORD.
`Public Sub Load(key)`	`String key`	`None`	▪ **Implementation of Load()** method. ▪ Intended for this class objects to be able to LOAD themselves from database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ **Create the method to satisfy the COMPILER, but do not implement this method** ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Update()`	None	`None`	▪ **Implementation of Update()** method. ▪ Intended for this class objects to be able to UPDATE or save their data to database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ **Create the method to satisfy the COMPILER, but do not implement this method** ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Insert()`	None	`None`	▪ **Implementation of Insert()** method. ▪ Intended for this class objects to be able to ADD or insert a new record to database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ **Create the method to satisfy the COMPILER, but do not implement this method** ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Delete(key)`	`String key`	`None`	▪ **Implementation of Delete()** method. ▪ Intended for this class objects to be able to DELETE or remove its record from database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ **Create the method to satisfy the COMPILER, but do not implement this method** ▪ CREATE THE HEADER WITH AN EMPTY BODY.

Class EmployeeList

- ***EmployeeList*** COLLECTION Class. Object of this class will store and manage **Employee Objects** in memory and load/save them from ***TEXT FILE***.
- This class encapsulates a ***DICTIONARY COLLECTION*** OBJECT by ***INHERITING*** from the ***DICTIONARYBASE CLASS***.
- The key properties/methods are described as follows:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

General Class Information	Description
`Public Class EmployeeList` `Inherits DictionaryBase`	▪ Class that encapsulates a DICTIONARY COLLECTION OBJECT. ▪ This class is inherited from the ***DictionaryBase*** Collection Library. ▪ Note that you must import the ***System.Collection*** library.

Public Properties (GET/SET)	Description
`Public Shadows ReadOnly Property Count() As Integer`	READ-ONLY GET: returns NUMBER OF ELEMENTS OR OBJECT IN THE COLLECTION DICTIONARY. Property should shadow the base class equivalent. ▪ GET Algorithm: **1.** Calls BASE CLASS `MyBase.Dictionary.Count` Property.
`Public Property Item(ByVal key As String) As Employee`	▪ **Wrapper PROPERTY** that **GET & SET** *Employee* **OBJECTS in the DICTIONARY COLLECTION** ▪ This Property GETS the POINTER to the OBJECT in the COLLECTION based on its KEY. ▪ This Property SETS the OBJECT WHO'S KEY is passed as parameter with the OBJECT being assigned. ▪ GET Algorithm: **1.** CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. **2.** Use `CType()` Function to convert to native data type of DICTIONARY collection to ***Employee*** Class type. ▪ SET Algorithm: **1.** CALLS the BASE CLASS `MyBase.Dictionary.Contains(key)` Method determine if KEY exists. **2.** If exists it calls `MyBase.Dictionary.Item(key)` to do the work of SETTING the VALUE. **3.** Else, THROWS `Throw New System.ArgumentException("ID Not found")` EXCEPTION indicating KEY WAS NOT FOUND.

Public Methods	Parameters	Return Type	Description
`Public Sub Add(ByVal key As String, ByVal objEmployee As Employee)`	`Employee objEmployee`	`None`	▪ **Implementation of Add(object)** method. ▪ ***Wrapper Method*** that ADDS the object & its associated KEY passed as argument into the collection. ▪ In this case, the KEY is the ***SSNumber*** PROPERTY of the Object. ▪ It is assumed the Object if CREATED & POPULATED in the User-Interface or calling program when passed as argument to the method call. Method simply adds the object to the COLLECTION. ▪ Algorithm: 1. Calls `MyBase.Dictionary.Add(key, objEmployee)` Method to add the object to Collection. 2. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Sub Add(ByVal x, ByVal y,ByVal z, etc.)`	`Variable for each required Parameter to SET PROPERTY of the Employee Class Object. Normally the paramters of the Parameterized Constructor.`	`None`	▪ **Implementation of Add(x,y,z etc.)** method. ▪ ***Wrapper Method*** that ADDS object into the collection. ▪ Same functionality as previous ADD, except NO populated OBJECT is passed as parameter, but the individual values that make up the OBJECT. ▪ The method itself creates the Object, populates it with the values from parameters and then ADDS the object to the COLLECTION with the KEY. ▪ In this case, the KEY is the ***SSNumber*** PARAMETER value of the method's parameter list. ▪ This version of ADD, requires less programming in the User-Interface. ▪ Algorithm: 1. Creates either DEFAULT Temporary ***Employee*** OBJECT & SETS the appropriate properties based on parameters VALUES passed to method. Or CREATES PARAMETERIZED Constructor OBJECT, passing to the OBJECT the VALUES of the parameters passed to method. 2. Calls `MyBase.Dictionary.Add(key, objEmployee)` Method to add the object to Collection. The KEY being the ***SSNumber*** PROPERTY of the object created. 3. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Function Edit(ByVal key As String, ByVal objEmployee As Employee) As Boolean`	`Employee objEmployee`	`True` `False`	▪ **Implementation of Edit(object)** method. ▪ ***Wrapper Method*** that EDITS the OBJECT in the COLLECTION whose KEY is passed as parameter to method. ▪ In this case, the KEY is the ***SSNumber*** PROPERTY of the Object ▪ The OBJECT in COLLECTION is edited by SETTING its PROPERTIES, with the values or the PROPERTIES of the OBJECT passed as parameter. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***Employee*** Class type. 3. Returns a FALSE if POINTER returned by `MyBase.Dictionary.Item(key)` Property is a ***Nothing***. 4. Else, **GETS PROPERTIES** of Object passed as parameter & **SETS PROPERTIES** of OBJECT in the COLLECTION whose POINTER was returned by `Dictionary.Item(key)` Property & **Returns** a **True**. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Function Edit(ByVal x, ByVal y,ByVal z, etc.) As Boolean`	Variable for each required Parameter to SET PROPERTY of the ***Employee*** Class Object. To be EDITED.	`True` `False`	▪ **Implementation of Edit(x,y,z)** method. ▪ ***Wrapper Method*** that EDITS the OBJECT in the COLLECTION who's associated KEY is passed as ONE of the parameter variables. The OBJECT in COLLECTION is edited by SETTING its PROPERTIES with the values passed as parameters to method, except the KEY parameter. ▪ In this case, the KEY is the ***SSNumber*** PARAMETER value of the method's parameter list. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***Employee*** Class type. 3. Returns a FALSE if POINTER returned by `MyBase.Dictionary.Item(key)` Property is a ***Nothing***. 4. Else **SETS PROPERTIES of OBJECT** in the COLLECTION whose POINTER was returned by `MyBase.Dictionary.Item(key)` with values passed as parameters, except the value that represents the KEY & **Returns** a **True** 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Function Remove(ByVal key As String) As Boolean`	`String Key`	`True` `False`	▪ **Implementation of Remove(Key)** method. ▪ ***Wrapper Method*** that REMOVES the object from COLLECTION who's associated KEY is passed as parameter to method. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Contains(key)` Method determine if KEY exists. 2. If exists it calls `MyBase.Dictionary.Remove(key)` to do the work of REMOVING OBJECT from COLLECTION and returns a **TRUE**. 3. Else, if not exists, then it returns a **FALSE**. 4. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Function Print(ByVal key As String) As Boolean`	`String Key`	`True` `False`	▪ **Implementation of Print(Key)** method. ▪ ***Method*** that PRINTS the content of the OBJECT in the COLLECTION who's associated KEY is passed as parameter. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***Employee*** Class type. 3. Returns a FALSE if POINTER returned by `Dictionary.Item(key)` Property is a ***Nothing***. 4. Else CALLS the `object.Print()` Method of the OBJECT in the COLLECTION whose POINTER was returned by `MyBase.Dictionary.Item(key)`. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Sub PrintAll()`	None	**None**	▪ **Implementation of PrintAll()** method. ▪ ***Method*** that PRINTS the content of ALL THE OBJECT in the COLLECTION. ▪ Algorithm: 1. Uses **`For Each objDictionaryEntry In MyBase.Dictionary`** LOOP to iterate through the collection. 2. Use **`CType()`** Function to convert to native data type of `DictionaryEntry` object to ***Employee*** Class type. 3. CALLS the **`object.Print()`** Method of EACH OF THE OBJECT in the COLLECTION. 4. Add Error-Handling code using **`Try-Catch-Finally`** to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Shadows Sub Clear()`	None	**None**	▪ **Implementation of Clear()** method. ▪ **Method** uses **Shadows** keyword to Shadow the Base Class Method which already implements this functionality. We are simply repeating the process here. ▪ ***Method*** that CLEARS or DELETES ALL OBJECTS in the COLLECTION. ▪ Algorithm: 1. CALLS **`MyBase.Dictionary.Clear()`** to do the work. 2. Add Error-Handling code using **`Try-Catch-Finally`** to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
Public Function Authenticate(ByVal user As String, ByVal pass As String) As Boolean	**string user, string user**	**True False**	▪ **Implementation of Authenticate(u,p)** method. ▪ This method authenticates the username & password parameters, by Employee SEARCHING the COLLECTION and calling each ***Obect.Authenticate(U,P)*** to verify authenticity. ▪ Algorithm: 1. ***LOADS*** the COLLECTION with OBJECTS FROM FILE. 2. Uses **For Each objDictionaryEntry In MyBase.Dictionary** LOOP to iterate through the collection. 3. Use **CType()** Function to convert to native data type of **DictionaryEntry** object to ***Employee*** Class type. 4. CALLS the **object.Authenticate(user, pass)** Method of EACH OF THE OBJECT in the COLLECTION to verify authenticity. 5. IF FOUND, ***CLEARS*** THE COLLECTION & Returns a TRUE 6. If NOT FOUND, ***CLEARS*** THE COLLECTION & returns a FALSE. 7. Add Error-Handling code using **Try-Catch-Finally** to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

`#Region "Helper Methods"`
 	
`'Methods used to assist other methods or maintenance`
`Public Function ToArray() As Employee()`
`'Declare an empty array of Employee`
`Dim arrEmployeeList(MyBase.Dictionary.Count - 1) As Customer`
 	
`'Use Shared CopyTo() Method of Dictionary Class to convert Collection to Array`
 	
`MyBase.Dictionary.Values.CopyTo(arrEmployeeList, 0)`
`Return arrEmployeeList`
`End Function`
 	
`#End Region`
	None	**Pointer Employee to ARRAY Employee()**	▪ **Implementation of ToArray()** method.
			▪ **OPTIONAL FOR THOSE WHO WANT TO USED DATAVIEWGRIDS AND USE DATA BINDING.**
			▪ **Method** CONVERTS **MyBase.Dictionary** DICTIONARY COLLECTION OBJECT TO ARRAY by leveraging the **MyBase.Dictionary.Values.CopyTo()** method of the DICTIONARY CLASS.
			▪ Method the **Return** ARRAY so it can be used for data binding.
			▪ **CODE IS PROVIDED IN THE FIRST COLUNM, SIMPLY COPY AND USE**.
			▪ Algorithm:
			1. Creates a temporary ARRAY that holds ***Employee*** Objects.
			2. CALLS **MyBase.Dictionary.Values.CopyTo()** method to CONVERT THE COLLECTION TO ARRAY.
			3. **Return** the CONVERTED ARRAY.

Public Data Access Methods	Parameters	Return Type	Description
`Public Shared Function Create()`	`None`	`EmployeeList Reference or Pointer`	▪ **Implementation of `Shared Function Create()`.** ▪ OBJECT FACTORY METHOD. ▪ Creates & RETURNS FULLY READY AND INIALIZED ***EMPLOYEELIST*** OBJECT. - Objects created may include initialized data from database. ▪ STUB FUNCTION METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER, EMPTY BODY & RETURN KEYWORD WITH CODE TO SATISFY COMPILER.
`Public Sub Load()`	None	`None`	▪ **Implementation of Load()** method. ▪ LOAD the OBJECTS from the `EmployeeData.txt` file and ADDS them to the COLLECTION. ▪ Algorithm: 1. USE `File.Exists("EmployeeData.txt")` to verify if FILE EXISTS. 2. If FILE DOES NOT EXISTS IT CREATES IT using `File.Create("EmployeeData.txt")` Method. 3. Open File for READING. 4. Read a LINE from file & PARSE each comma-delimited line. 5. Create new temporary OBJECT if the ***Employee*** Class. 6. SET OBJECT WTH values from LINE READ FROM FILE. 7. ADD OBJECT TO COLLECTION. 8. Repeat this process until EOF. 9. Add Error-Handling code using `Try-Catch-Finally` to handle a ***GENERAL*** Exception.
`Public Sub Save()`	None	`None`	1. **Implementation of Save()** method. 2. SAVES the OBJECTS IN THE COLLECTION to the `EmployeeData.txt` file. 3. Algorithm: 4. Open File for WRITTING. 5. Uses `For Each objDictionaryEntry In MyBase.Dictionary` LOOP to iterate through the collection. 6. Use `CType()` Function to convert to native data type of `DictionaryEntry` object to ***Employee*** Class type 7. GETS ALL THE PROPERTIES FOR EACH OBJECT IN ARRAY and CREATES A ***Comma-delimited string*** from ALL THE PROPERTIES OF THE OBJECT IN ARRAY. 8. CALLS THE STREAMREADER OBJECT `WriteLine()` Method TO WRITE THE ***Comma-delimited string*** LINE TO FILE. 9. Repeat this process until ALL OBJECTS IN ARRAY HAVE BEEN VISITED AND ITS PROPERTIES WRITTEN TO THE FILE AS A ***Comma-delimited string***. 10. Add Error-Handling code using `Try-Catch-Finally` to handle a ***GENERAL*** Exception.

Class CustomerList

- ***CustomerList*** COLLECTION Class. Object of this class will store and manage **Customer Objects** in memory and load/save them from ***TEXT FILE***.
- This class encapsulates a ***DICTIONARY COLLECTION*** OBJECT by ***INHERITING*** from the ***DICTIONARYBASE CLASS***.
- The key properties/methods are described as follows:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

General Class Information	Description
`Public Class CustomerList` `Inherits DictionaryBase`	▪ Class that encapsulates a DICTIONARY COLLECTION OBJECT. ▪ This class is inherited from the ***DictionaryBase*** Collection Library. ▪ Note that you must import the ***System.Collection*** library.

Public Properties (GET/SET)	Description
`Public Shadows ReadOnly Property Count() As Integer`	READ-ONLY GET: returns NUMBER OF ELEMENTS OR OBJECT IN THE COLLECTION DICTIONARY. Property should shadow the base class equivalent. ▪ GET Algorithm: **1.** Calls BASE CLASS `MyBase.Dictionary.Count` Property.
`Public Property Item(ByVal key As String) As Customer`	▪ **Wrapper PROPERTY** that **GET & SET** *Customer* **OBJECTS in the DICTIONARY COLLECTION** ▪ This Property GETS the POINTER to the OBJECT in the COLLECTION based on its KEY. ▪ This Property SETS the OBJECT WHO'S KEY is passed as parameter with the OBJECT being assigned. ▪ GET Algorithm: **1.** CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. **2.** Use `CType()` Function to convert to native data type of DICTIONARY collection to ***Customer*** Class type. ▪ SET Algorithm: **1.** CALLS the BASE CLASS `MyBase.Dictionary.Contains(key)` Method determine if KEY exists. **2.** If exists it calls `MyBase.Dictionary.Item(key)` to do the work of SETTING the VALUE. **3.** Else, THROWS `Throw New System.ArgumentException("ID Not found")` EXCEPTION indicating KEY WAS NOT FOUND.

Public Methods	Parameters	Return Type	Description
`Public Sub Add(ByVal key As String, ByVal objCustomer As Customer)`	`Customer objCustomer`	`None`	▪ **Implementation of Add(object)** method. ▪ ***Wrapper Method*** that ADDS the object & its associated KEY passed as argument into the collection. ▪ In this case, the KEY is the ***IDNumber*** PROPERTY of the Object. ▪ It is assumed the Object if CREATED & POPULATED in the User-Interface or calling program when passed as argument to the method call. Method simply adds the object to the COLLECTION. ▪ Algorithm: 1. Calls `MyBase.Dictionary.Add(key, objCustomer)` Method to add the object to Collection. 2. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Sub Add(ByVal x, ByVal y,ByVal z, etc.)`	`Variable for each required Parameter to SET PROPERTY of the Customer Class Object. Normally the paramters of the Parameterized Constructor.`	`None`	▪ **Implementation of Add(x,y,z etc.)** method. ▪ ***Wrapper Method*** that ADDS object into the collection. ▪ Same functionality as previous ADD, except NO populated OBJECT is passed as parameter, but the individual values that make up the OBJECT. ▪ The method itself creates the Object, populates it with the values from parameters and then ADDS the object to the COLLECTION with the KEY. ▪ In this case, the KEY is the ***IDNumber*** PARAMETER value of the method's parameter list. ▪ This version of ADD, requires less programming in the User-Interface. ▪ Algorithm: 1. Creates either DEFAULT Temporary ***Customer*** OBJECT & SETS the appropriate properties based on parameters VALUES passed to method. Or CREATES PARAMETERIZED Constructor OBJECT, passing to the OBJECT the VALUES of the parameters passed to method. 2. Calls `MyBase.Dictionary.Add(key, objCustomer)` Method to add the object to Collection. The KEY being the ***IDNumber*** PROPERTY of the object created. 3. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Function Edit(ByVal key As String, ByVal objCustomer As Customer) As Boolean`	`Customer objCustomer`	`True` `False`	▪ **Implementation of Edit(object)** method. ▪ ***Wrapper Method*** that EDITS the OBJECT in the COLLECTION whose KEY is passed as parameter to method. ▪ In this case, the KEY is the ***IDNumber*** PROPERTY of the Object ▪ The OBJECT in COLLECTION is edited by SETTING its PROPERTIES, with the values or the PROPERTIES of the OBJECT passed as parameter. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***Customer*** Class type. 3. Returns a FALSE if POINTER returned by `MyBase.Dictionary.Item(key)` Property is a ***Nothing***. 4. Else, **GETS PROPERTIES** of Object passed as parameter & **SETS PROPERTIES** of OBJECT in the COLLECTION whose POINTER was returned by `Dictionary.Item(key)` Property & **Returns** a **True**. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Function Edit(ByVal x, ByVal y,ByVal z, etc.) As Boolean`	Variable for each required Parameter to SET PROPERTY of the ***Customer*** Class Object. To be EDITED.	`True` `False`	▪ **Implementation of Edit(x,y,z)** method. ▪ ***Wrapper Method*** that EDITS the OBJECT in the COLLECTION who's associated KEY is passed as ONE of the parameter variables. The OBJECT in COLLECTION is edited by SETTING its PROPERTIES with the values passed as parameters to method, except the KEY parameter. ▪ In this case, the KEY is the ***IDNumber*** PARAMETER value of the method's parameter list. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***Customer*** Class type. 3. Returns a FALSE if POINTER returned by `MyBase.Dictionary.Item(key)` Property is a ***Nothing***. 4. Else **SETS PROPERTIES** of OBJECT in the COLLECTION whose POINTER was returned by `MyBase.Dictionary.Item(key)` with values passed as parameters, except the value that represents the KEY & **Returns** a **True**. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Function Remove(ByVal key As String) As Boolean`	`String Key`	`True` `False`	▪ **Implementation of Remove(Key)** method. ▪ ***Wrapper Method*** that REMOVES the object from COLLECTION who's associated KEY is passed as parameter to method. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Contains(key)` Method determine if KEY exists. 2. If exists it calls `MyBase.Dictionary.Remove(key)` to do the work of REMOVING OBJECT from COLLECTION and returns a **TRUE**. 3. Else, if not exists, then it returns a **FALSE**. 4. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Function Print(ByVal key As String) As Boolean`	`String Key`	`True` `False`	▪ **Implementation of Print(Key)** method. ▪ ***Method*** that PRINTS the content of the OBJECT in the COLLECTION who's associated KEY is passed as parameter. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***Customer*** Class type. 3. Returns a FALSE if POINTER returned by `Dictionary.Item(key)` Property is a ***Nothing***. 4. Else CALLS the `object.Print()` Method of the OBJECT in the COLLECTION whose POINTER was returned by `MyBase.Dictionary.Item(key)`. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Sub PrintAll()`	None	**None**	▪ **Implementation of PrintAll()** method. ▪ ***Method*** that PRINTS the content of ALL THE OBJECT in the COLLECTION. ▪ Algorithm: 1. Uses **`For Each objDictionaryEntry In MyBase.Dictionary`** LOOP to iterate through the collection. 2. Use **`CType()`** Function to convert to native data type of `DictionaryEntry` object to ***Customer*** Class type. 3. CALLS the **`object.Print()`** Method of EACH OF THE OBJECT in the COLLECTION. 4. Add Error-Handling code using **`Try-Catch-Finally`** to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Shadows Sub Clear()`	None	**None**	▪ **Implementation of Clear()** method. ▪ **Method** uses **Shadows** keyword to Shadow the Base Class Method which already implements this functionality. We are simply repeating the process here. ▪ ***Method*** that CLEARS or DELETES ALL OBJECTS in the COLLECTION. ▪ Algorithm: 1. CALLS **`MyBase.Dictionary.Clear()`** to do the work. 2. Add Error-Handling code using **`Try-Catch-Finally`** to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

`#Region "Helper Methods"`
 	
`'Methods used to assist other methods or maintenance`
`Public Function ToArray() As Customer()`
`'Declare an empty array of Customers`
`Dim arrCustomerList(MyBase.Dictionary.Count - 1) As Customer`
 	
`'Use Shared CopyTo() Method of Dictionary Class to convert Collection to Array`
 	
`MyBase.Dictionary.Values.CopyTo(arrCustomerList, 0)`
`Return arrCustomerList`
`End Function`
 	
`#End Region`
	None	**`Pointer to Customer ARRAY Customer()`**	▪ **Implementation of ToArray()** method.
			▪ **OPTIONAL FOR THOSE WHO WANT TO USED DATAVIEWGRIDS AND USE DATA BINDING.**
			▪ **Method** CONVERTS **`MyBase.Dictionary`** DICTIONARY COLLECTION OBJECT TO ARRAY by leveraging the **`MyBase.Dictionary.Values.CopyTo()`** method of the DICTIONARY CLASS.
			▪ Method the **`Return`** ARRAY so it can be used for data binding.
			▪ **CODE IS PROVIDED IN THE FIRST COLUNM, SIMPLY COPY AND USE**.
			▪ Algorithm:
			1. Creates a temporary ARRAY that holds ***Customer*** Objects.
			2. CALLS **`MyBase.Dictionary.Values.CopyTo()`** method to CONVERT THE COLLECTION TO ARRAY.
			3. **`Return`** the CONVERTED ARRAY.

Public Data Access Methods	Parameters	Return Type	Description
`Public Shared Function Create()`	`None`	`CustomerList Reference or Pointer`	▪ **Implementation of** `Shared Function Create()`. ▪ OBJECT FACTORY METHOD. ▪ Creates & RETURNS FULLY READY AND INIALIZED ***CUSTOMERLIST*** OBJECT. - Objects created may include initialized data from database. ▪ STUB FUNCTION METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER, EMPTY BODY & RETURN KEYWORD WITH CODE TO SATISFY COMPILER.
`Public Sub Load()`	None	`None`	▪ **Implementation of Load()** method. ▪ LOAD the OBJECTS from the `CustomerData.txt` file and ADDS them to the COLLECTION. ▪ Algorithm: 1. USE `File.Exists("CustomerData.txt")` to verify if FILE EXISTS. 2. If FILE DOES NOT EXISTS IT CREATES IT using `File.Create("CustomerData.txt")` Method. 3. Open File for READING. 4. Read a LINE from file & PARSE each comma-delimited line. 5. Create new temporary OBJECT if the Customer Class. 6. SET OBJECT WTH values from LINE READ FROM FILE. 7. ADD OBJECT TO COLLECTION. 8. Repeat this process until EOF. 9. Add Error-Handling code using `Try-Catch-Finally` to handle a ***GENERAL*** Exception.
`Public Sub Save()`	None	`None`	1. **Implementation of Save()** method. 2. SAVES the OBJECTS IN THE COLLECTION to the `CustomerData.txt` file. 3. Algorithm: 4. Open File for WRITTING. 5. Uses `For Each objDictionaryEntry In MyBase.Dictionary` LOOP to iterate through the collection. 6. Use `CType()` Function to convert to native data type of `DictionaryEntry` object to ***Customer*** Class type 7. GETS ALL THE PROPERTIES FOR EACH OBJECT IN ARRAY and CREATES A ***Comma-delimited string*** from ALL THE PROPERTIES OF THE OBJECT IN ARRAY. 8. CALLS THE STREAMREADER OBJECT `WriteLine()` Method TO WRITE THE ***Comma-delimited string*** LINE TO FILE. 9. Repeat this process until ALL OBJECTS IN ARRAY HAVE BEEN VISITED AND ITS PROPERTIES WRITTEN TO THE FILE AS A ***Comma-delimited string***. 10. Add Error-Handling code using `Try-Catch-Finally` to handle a ***GENERAL*** Exception.

Class Product

- Represents ***Products*** to be sold, serviced or rented by a company. Intended to be used as a base class
- The key properties/methods are described as follows:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

PUBLIC Enumerated Data Type Declarations	Description
`Public Enum Rating`	▪ Purpose: This is an enumerated Data type declaration that represents product's rating, example: PG, PG13, and R. ▪ Data type: `Enumerated` ▪ Enumerated Values: *G, PG, PG-13, NC-17, R* , None

General Class Information	Description
`Public MustInherit` Product	▪ `MustInherit` Base Class – Designed for inheritance only. No objects of this class should or can be created. ▪ Represents the products to be sold, rented, etc., by customers.

Private Data	Description
`Private` m_IDNumber	Purpose: Represents product's ID number. Data type: `String`
`Private` m_Title	Purpose: Represents product's name. Data type: `String`
`Private` m_Description	Purpose: Represents product's description. Data type: `String`
`Private` **m_enumRating**	▪ Purpose: This is variable of the `Enum Rating` Data which will store only `Rating` variables. ▪ Data type: `Rating` ▪ Enumerated Values: `Rating`.*G*, `Rating`.*PG*, `Rating`.*PG13*, `Rating`.*NC17*, `Rating`.*R*, `Rating`.*None*.
`Private` m_Available	Purpose: Represents product's availability, in stock (true or false). Data type: `Boolean`
`Private` m_SalePrice	Purpose: Represents product's sales price. Data type: `Decimal`
`Private` m_RentalRate	Purpose: Represents product's daily rental rate. Data type: `Decimal`
`Private` m_LateFee	Purpose: Represents product's daily late fees for rentals. Data type: `Decimal`

Public Properties (GET/SET)	Description
`Public` IDNumber	GET/SET **m_IDNumber** private data
`Public` Title	GET/SET **m_Title** private data
`Public` Description	GET/SET **m_Description** private data
`Public` Rating	GET/SET **m_enumRating** private data (can only GET or SET values of Enumerated type or ***Rating.Values***.
`Public` Available	GET/SET **m_Available** private data.
`Public` SalePrice	GET/SET **m_SalePrice** private data
`Public` RentalRate	GET/SET **m_RentalRate** private data

Public LateFee	GET/SET **m_LateFee** private data

Constructors	Parameters	Return Type	Description
Public New()	None	**N/A**	▪ **Default Constructor**. Should initialize the PRIVATE DATA members with appropriate default values ▪ Sets to appropriate default values: - m_IDNumber, m_Title, m_Description =”” - m_enumRating = **Rating.***None* - m_Available = **True** - m_SalePrice, m_RentalRate, m_LateFee = 0.0
Public New(x, y, z etc..)	▪ A parameter to Set each of the following **Properties** only (AVAILABLE not included): - IDNumber - Title - Description - Rating - SalePrice - RentalRate - LateFee ▪ Should have a total of **7** parameters: - **par1 to Par7** ▪ Name the parameters as you see fit. ▪ All parameters are Pass-by-Value	**N/A**	▪ **Parameterized Constructor** ▪ Sets the following **PROPERTIES** to matching parameter list, EXCEPT the AVAILABLE which should be set by its' Private Data directly: - IDNumber = **par1** - Title = **par2** - Description = **par3** - Rating = **par4** - SalePrice = **par5** - RentalRate = **par6** - LateFee = **par7** ▪ The Available **PROPERTY** is not part of the parameter so it needs to be defaulted: - **m_Available** = **True**

Public MustOverride Methods	Parameters	Return Type	Description
Public MustOverride Sub Print()	None	**None**	▪ **MustOverride Print()** method. ▪ Intended for derived classes to print their data ▪ **Declaration only**. Must be implemented in derived classes.
Public MustOverride Sub Product_Rental()	None	**None**	▪ **MustOverride Product_Rental()** method. ▪ Intended for derived classes to be able to perform Product Rentals. ▪ **Declaration only**. Must be implemented in derived classes.
Public MustOverride Sub Product_Return()	None	**None**	▪ **MustOverride Product_Return()** method. ▪ Intended for derived classes to be able to perform Product Returns as part of rental process. ▪ **Declaration only**. Must be implemented in derived classes.
Public MustOverride Sub Product_Sell()	None	**None**	▪ **MustOverride Product_Sell()** method. ▪ Intended for derived classes to be able to SELL the Products of the business. ▪ **Declaration only**. Must be implemented in derived classes.

Class DVD

- Represents the DVD to be sold and rented.
- The key properties/methods are described as follows:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

PUBLIC Enumerated Data Type Declarations	Description
`Public Enum MovieCategory`	Purpose: Enumerated Data type that represents the movie category such as: **Action, Drama, Comedy etc.** Data type: `enum` Enumerated Values: ***Action_Adventure, Drama, Famil_Kids, Horror, Sci-Fi_Fantasy, Music, Sports, Romance, Comedy, Western, None***
`Public Enum DVDFormat`	Purpose: Enumerated Data type that represents the DVD movie format: **DVD, HDVD, Blue-ray etc.** Data type: `enum` Enumerated Values: ***DVD, HD-DVD, BLU-RAY DISC, None***

General Class Information	Description
`Public class DVD Inherits Product`	Inherits from Product class and defines blueprints for ***DVD*** Objects

Private Data	Description
`Private` **m_enumMovieCategory**	▪ Purpose: Enumerated Data type that represents the movie category such as: **Action, Drama, Comedy etc.** ▪ Data type: `MovieCategory` ▪ Enumerated Values: `MovieCategory`***.Action_Adventure,*** `MovieCategory`***.Drama,*** `MovieCategory`***.Family_Kids,*** `MovieCategory`***.Horror,*** `MovieCategory`***.SciFi_Fantasy,*** `MovieCategory`***.Music,*** `MovieCategory`***.Sports,*** `MovieCategory`***.Romance,*** `MovieCategory`***.Western,*** `MovieCategory`***.Comedy,*** `MovieCategory`***.None***
`Private` **m_enumDVDFormat**	▪ Purpose: Enumerated Data type that represents the DVD movie format: **DVD, HDVD, Blue-ray etc.** ▪ Data type: `DVDFormat` ▪ Enumerated Values: `DVDFormat`***.DVD,*** `DVDFormat`***.HDDVD,*** `DVDFormat`***.BLURAY DISC,*** `DVDFormat`***.None***

Public Properties (GET/SET)	Description
`Public` **Category**	▪ **GET/SET enumMovieCategory private data** ▪ Enumerated values that can be assigned to this property: `MovieCategory`***.Action_Adventure,*** `MovieCategory`***.Drama,*** `MovieCategory`***.Family_Kids,*** `MovieCategory`***.Horror,*** `MovieCategory`***.SciFi_Fantasy,*** `MovieCategory`***.Music,*** `MovieCategory`***.Sports,*** `MovieCategory`***.Romance,*** `MovieCategory`***.Western,*** `MovieCategory`***.Comedy,*** `MovieCategory`***.None***
`Public` **Format**	▪ **GET/SET enumDVDFormat private data** ▪ Enumerated values that can be assigned to this property: `DVDFormat`***.DVD,*** `DVDFormat`***.HDDVD,*** `DVDFormat`***.BLURAY DISC,*** `DVDFormat`***.None***

Constructors	Parameters	Return Type	Description
`Public New()`	None	`N/A`	▪ **Default Constructor**. ▪ Should initialize the private data members with appropriate default values for this Class and the Base Class. ▪ Calls **Base Class Default Constructor** ▪ Sets to appropriate default values: - m_enumMovieCategory = `MovieCategory`.***None*** - m_enumDVDFormat = `DVDFormat`.***None***
`Public New(x, y, z etc..)`	▪ A parameter to Set each of the following Base Class & THIS class **Properties** (AVAILABLE not included): - IDNumber - Title - Description - Rating - SalePrice - RentalRate - LateFee - Category - Format ▪ Should have a total of 9 parameters: - **par1 to Par9** ▪ Name the parameters as you see fit. ▪ All parameters are Pass-by-Value	`N/A`	▪ **Parameterized Constructor** ▪ Sets the **PROPERTIES** of the Person Base Class and this ***DVD*** Class matching parameter list. ▪ Calls **Base Class Parameterized Constructor** to handle the following parameters: - IDNumber = **par1** - Title = **par2** - Description = **par3** - Rating = **par4** - SalePrice = **par5** - RentalRate = **par6** - LateFee = **par7** ▪ Sets the **PROPERTIES** of this class with the following parameters: - **Category** = **par8** - **Format** = **par9**

Public Overrides Methods	Parameters	Return Type	Description
`Public Overrides Sub Print()`	None	`None`	▪ **Implementation of Print()** method. ▪ The *Print()* Method WRITES ALL OBJECT'S DATA TO THE PRINTER FILE as follows: **1.** Opens ***Network_Printer.txt*** file for APPENDING. **2.** Write each object's property/data in the following FORMAT: Printing ***DVD*** ..... ID Number = ***value*** Title = ***value*** Description = ***value*** Rating = ***value*** SalePrice = ***value*** RentalRate = ***value*** Etc…. **3.** Close the file ▪ Add Error-Handling code using `try-catch-finally` to handle all required exceptions for any file access, array or general exceptions ▪ Follow best practice of trapping for unexpected general errors in addition to specific errors ▪ Use `throw` statement to re-throw all exceptions
`Public Overrides Sub Product_Rental()`	None	`None`	▪ **Implementation of Product_Rental()** method. ▪ `Overrides` to satisfy `MustOverride` Base Class requirements. ▪ Handles future product rental transactions. ▪ NO IMPLEMETATION, STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Overrides Sub Product_Return()`	None	`None`	▪ **Implementation of Product_Return()** method. ▪ `Overrides` to satisfy `MustOverride` Base Class requirements. ▪ Handles future product returns transactions as part of rental process. ▪ NO IMPLEMETATION, STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Overrides Sub Product_Sell()`	None	`None`	▪ **Implementation of Product_Sell()** method. ▪ `Overrides` to satisfy `MustOverride` Base Class requirements. ▪ Handles future product selling transactions. ▪ NO IMPLEMETATION, STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.

Public Data Access Methods	Parameters	Return Type	Description
`Public Shared Function Create()`	`None`	`Employee Reference or Pointer`	▪ **Implementation of `Shared Function Create()`** method. ▪ OBJECT FACTORY METHOD. ▪ Creates & RETURNS FULLY READY AND INIALIZED ***DVD*** OBJECTS. - Objects created may include initialized data from database. ▪ STUB FUNCTION METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY & RETURN KEYWORD.
`Public Sub Load(key)`	`String key`	`None`	▪ **Implementation of Load()** method. ▪ Intended for this class objects to be able to LOAD themselves from database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Update()`	None	`None`	▪ **Implementation of Update()** method. ▪ Intended for this class objects to be able to UPDATE or save their data to database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Insert()`	None	`None`	▪ **Implementation of Insert()** method. ▪ Intended for this class objects to be able to ADD or insert a new record to database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Delete(key)`	`String key`	`None`	▪ **Implementation of Delete()** method. ▪ Intended for this class objects to be able to DELETE or remove its record from database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.

Class VideoGame

- Represents the Video Game to be sold and rented.
- The key properties/methods are described as follows:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

PUBLIC Enumerated Data Type Declarations	Description
`Public Enum VideoGameCategory`	▪ Purpose: Enumerated Data type that represents the video game category such as: **Action, shooting, Racing etc.** ▪ Data type: `Enum` ▪ Enumerated Values: ***Action, Roleplaying, Shooting, Fighting, Racing, Sports, Strategy, Horror, Flight Simulators, Online, Rhythm, None***
`Public Enum VideoGameFormat`	▪ Purpose: Enumerated Data type that represents the video game format or type of Game Stations the game is intended for: **XBox, Play Station etc.** ▪ Data type: `enum` ▪ Enumerated Values: ***XBox, XBox 360, PS3, PS2,GameCube, DS, Wii, PC, None***

General Class Information	Description
`Public class VideoGame` `Inherits Product`	Inherits from Product class and defines blueprints for ***VideoGame*** Objects

Private Data	Description
`Private` **m_enumVideoGameCategory**	▪ Purpose: Variable of the `enum VideoGameCategory` Enumerated Data type that represents the video game category such as: **Action, shooting, Racing etc.** ▪ Data type: `VideoGameCategory` ▪ Values assigned: `VideoGameCategory`***.Action,*** `VideoGameCategory`***.Role-playing,*** `VideoGameCategory`***.Shooting,*** `VideoGameCategory`***.Fighting,*** `VideoGameCategory`***.Racing,*** `VideoGameCategory`***.Sports,*** `VideoGameCategory`***.Strategy,*** `VideoGameCategory`***.Horror,*** `VideoGameCategory`***.Flight Simulators,*** `VideoGameCategory`***.Online,*** `VideoGameCategory`***.Rhythm,*** `VideoGameCategory`***.None***
`Private` **m_enumVideoGameFormat**	▪ Purpose: Enumerated Data type that represents the video game format or type of Game Stations the game is intended for: **XBox, Play Station etc.** ▪ Data type: `VideoGameFormat` ▪ Enumerated Values: `VideoGameFormat`***.XBox,*** `VideoGameFormat`***.XBox 360,*** `VideoGameFormat`***.PS3,*** `VideoGameFormat`***.PS2,*** `VideoGameFormat`***.GameCube,*** `VideoGameFormat`***.DS,*** `VideoGameFormat`***.Wii,*** `VideoGameFormat`***.PC,*** `VideoGameFormat`***.None***

Public Properties (GET/SET)	Description
Public **Category**	▪ **GET/SET enumVideoGameCategory private data** ▪ Enumerated values that can be assigned to this property: **VideoGameCategory*.Action*, VideoGameCategory*.RolePlaying*, VideoGameCategory*.Shooting*, VideoGameCategory*.Fighting*, VideoGameCategory*.Racing*, VideoGameCategory*.Sports*, VideoGameCategory*.Strategy*, VideoGameCategory*.Horror*, VideoGameCategory*.Flight Simulators*, VideoGameCategory*.Online*, VideoGameCategory*.Rhythm*, VideoGameCategory*.None***
Public **Format**	▪ **GET/SET enumVideoGameFormat private data** ▪ Enumerated values that can be assigned to this property: **VideoGameFormat*.XBox*, VideoGameFormat*.-Box 360*, VideoGameFormat*.PS3*, VideoGameFormat*.PS2*, VideoGameFormat*.GameCube*, VideoGameFormat*.DS*, VideoGameFormat*.Wii*, VideoGameFormat*.PC*, VideoGameFormat*.None***

Constructors	Parameters	Return Type	Description
Public New()	None	**N/A**	▪ **Default Constructor**. ▪ Should initialize the private data members with appropriate default values for this Class and the Base Class. ▪ Calls **Base Class Default Constructor** ▪ Sets to appropriate default values: - m_enumVideoGameCategory = **VideoGameCategory*.None*** - m_enumVideoGameFormat = **VideoGameFormat*.None***
Public New(x, y, z etc..)	▪ A parameter to Set each of the following Base Class & THIS class **Properties** (AVAILABLE not included): - IDNumber - Title - Description - Rating - SalePrice - RentalRate - LateFee - Category - Format ▪ Should have a total of 9 parameters: - **par1 to Par9** ▪ Name the parameters as you see fit. ▪ All parameters are Pass-by-Value	**N/A**	▪ **Parameterized Constructor** ▪ Sets the **PROPERTIES** of the Person Base Class and this ***VideoGame*** Class matching parameter list. ▪ Calls **Base Class Parameterized Constructor** to handle the following parameters: - IDNumber = **par1** - Title = **par2** - Description = **par3** - Rating = **par4** - SalePrice = **par5** - RentalRate = **par6** - LateFee = **par7** ▪ Sets the **PROPERTIES** of this class with the following parameters: - **Category** = **par8** - **Format** = **par9**

Public Overrides Methods	Parameters	Return Type	Description
`Public Overrides Sub Print()`	None	`None`	▪ **Implementation of Print()** method. ▪ The *Print()* Method WRITES ALL OBJECT'S DATA TO THE PRINTER FILE as follows: **1.** Opens ***Network_Printer.txt*** file for APPENDING. **2.** Write each object's property/data in the following FORMAT: Printing ***VideoGame*** ..... ID Number = ***value*** Title = ***value*** Description = ***value*** Rating = ***value*** SalePrice = ***value*** RentalRate = ***value*** Etc…. **3.** Close the file ▪ Add Error-Handling code using `try-catch-finally` to handle all required exceptions for any file access, array or general exceptions ▪ Follow best practice of trapping for unexpected general errors in addition to specific errors ▪ Use `throw` statement to re-throw all exceptions
`Public Overrides Sub Product_Rental()`	None	`None`	▪ **Implementation of Product_Rental()** method. ▪ `Overrides` to satisfy `MustOverride` Base Class requirements. ▪ Handles future product rental transactions. ▪ NO IMPLEMETATION, STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Overrides Sub Product_Return()`	None	`None`	▪ **Implementation of Product_Return()** method. ▪ `Overrides` to satisfy `MustOverride` Base Class requirements. ▪ Handles future product returns transactions as part of rental process. ▪ NO IMPLEMETATION, STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Overrides Sub Product_Sell()`	None	`None`	▪ **Implementation of Product_Sell()** method. ▪ `Overrides` to satisfy `MustOverride` Base Class requirements. ▪ Handles future product selling transactions. ▪ NO IMPLEMETATION, STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.

Public Data Access Methods	Parameters	Return Type	Description
`Public Shared Function Create()`	`None`	`Employee Reference or Pointer`	▪ **Implementation of `Shared Function Create()`** method. ▪ OBJECT FACTORY METHOD. ▪ Creates & RETURNS FULLY READY AND INIALIZED ***VIDEOGAME*** OBJECTS. - Objects created may include initialized data from database. ▪ STUB FUNCTION METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY & RETURN KEYWORD.
`Public Sub Load(key)`	`String key`	`None`	▪ **Implementation of Load()** method. ▪ Intended for this class objects to be able to LOAD themselves from database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Update()`	None	`None`	▪ **Implementation of Update()** method. ▪ Intended for this class objects to be able to UPDATE or save their data to database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Insert()`	None	`None`	▪ **Implementation of Insert()** method. ▪ Intended for this class objects to be able to ADD or insert a new record to database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.
`Public Sub Delete(key)`	`String key`	`None`	▪ **Implementation of Delete()** method. ▪ Intended for this class objects to be able to DELETE or remove its record from database ▪ STUB METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER WITH AN EMPTY BODY.

Class DVDList

- ***DVDList*** COLLECTION Class. Object of this class will store and manage **DVD Objects** in memory and load/save them from ***TEXT FILE***.
- This class encapsulates a ***DICTIONARY COLLECTION*** OBJECT by ***INHERITING*** from the ***DICTIONARYBASE CLASS***.
- The key properties/methods are described as follows:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

General Class Information	Description
`Public Class DVDList` `Inherits DictionaryBase`	▪ Class that encapsulates a DICTIONARY COLLECTION OBJECT. ▪ This class is inherited from the ***DictionaryBase*** Collection Library. ▪ Note that you must import the ***System.Collection*** library.

Public Properties (GET/SET)	Description
`Public Shadows ReadOnly Property Count() As Integer`	READ-ONLY GET: returns NUMBER OF ELEMENTS OR OBJECT IN THE COLLECTION DICTIONARY. Property should shadow the base class equivalent. ▪ GET Algorithm: **2.** Calls BASE CLASS `MyBase.Dictionary.Count` Property.
`Public Property Item(ByVal key As String) As DVD`	▪ **Wrapper PROPERTY** that **GET & SET** *DVD* **OBJECTS in the DICTIONARY COLLECTION** ▪ This Property GETS the POINTER to the OBJECT in the COLLECTION based on its KEY. ▪ This Property SETS the OBJECT WHO'S KEY is passed as parameter with the OBJECT being assigned. ▪ GET Algorithm: **1.** CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. **2.** Use `CType()` Function to convert to native data type of DICTIONARY collection to ***DVD*** Class type. ▪ SET Algorithm: **1.** CALLS the BASE CLASS `MyBase.Dictionary.Contains(key)` Method determine if KEY exists. **2.** If exists it calls `MyBase.Dictionary.Item(key)` to do the work of SETTING the VALUE. **3.** Else, THROWS `Throw New System.ArgumentException("ID Not found")` EXCEPTION indicating KEY WAS NOT FOUND.

Public Methods	Parameters	Return Type	Description
`Public Sub Add(ByVal key As String, ByVal objDVD As DVD)`	`DVD objDVD`	`None`	▪ **Implementation of Add(object)** method. ▪ ***Wrapper Method*** that ADDS the object & its associated KEY passed as argument into the collection. ▪ In this case, the KEY is the ***IDNumber*** PROPERTY of the Object. ▪ It is assumed the Object if CREATED & POPULATED in the User-Interface or calling program when passed as argument to the method call. Method simply adds the object to the COLLECTION. ▪ Algorithm: 1. Calls `MyBase.Dictionary.Add(key, objDVD)` Method to add the object to Collection. 2. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Sub Add(ByVal x, ByVal y,ByVal z, etc.)`	`Variable for each required Parameter to SET PROPERTY of the DVD Class Object. Normally the paramters of the Parameterized Constructor.`	`None`	▪ **Implementation of Add(x,y,z etc.)** method. ▪ ***Wrapper Method*** that ADDS object into the collection. ▪ Same functionality as previous ADD, except NO populated OBJECT is passed as parameter, but the individual values that make up the OBJECT. ▪ The method itself creates the Object, populates it with the values from parameters and then ADDS the object to the COLLECTION with the KEY. ▪ In this case, the KEY is the ***IDNumber*** PARAMETER value of the method's parameter list. ▪ This version of ADD, requires less programming in the User-Interface. ▪ Algorithm: 1. Creates either DEFAULT Temporary ***DVD*** OBJECT & SETS the appropriate properties based on parameters VALUES passed to method. Or CREATES PARAMETERIZED Constructor OBJECT, passing to the OBJECT the VALUES of the parameters passed to method. 2. Calls `MyBase.Dictionary.Add(key, objDVD)` Method to add the object to Collection. The KEY being the ***IDNumber*** PROPERTY of the object created. 3. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Function Edit(ByVal key As String, ByVal objDVD As DVD) As Boolean`	`DVD objDVD`	`True` `False`	▪ **Implementation of Edit(object)** method. ▪ ***Wrapper Method*** that EDITS the OBJECT in the COLLECTION whose KEY is passed as parameter to method. ▪ In this case, the KEY is the ***IDNumber*** PROPERTY of the Object ▪ The OBJECT in COLLECTION is edited by SETTING its PROPERTIES, with the values or the PROPERTIES of the OBJECT passed as parameter. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***DVD*** Class type. 3. Returns a FALSE if POINTER returned by `MyBase.Dictionary.Item(key)` Property is a ***Nothing***. 4. Else, **GETS PROPERTIES** of Object passed as parameter & **SETS PROPERTIES** of OBJECT in the COLLECTION whose POINTER was returned by `Dictionary.Item(key)` Property & **Returns** a **True**. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Function Edit(ByVal x, ByVal y,ByVal z, etc.) As Boolean`	Variable for each required Parameter to SET PROPERTY of the ***DVD*** Class Object. To be EDITED.	`True` `False`	▪ **Implementation of Edit(x,y,z)** method. ▪ ***Wrapper Method*** that EDITS the OBJECT in the COLLECTION who's associated KEY is passed as ONE of the parameter variables. The OBJECT in COLLECTION is edited by SETTING its PROPERTIES with the values passed as parameters to method, except the KEY parameter. ▪ In this case, the KEY is the ***IDNumber*** PARAMETER value of the method's parameter list. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***DVD*** Class type. 3. Returns a FALSE if POINTER returned by `MyBase.Dictionary.Item(key)` Property is a ***Nothing***. 4. Else **SETS PROPERTIES** of OBJECT in the COLLECTION whose POINTER was returned by `MyBase.Dictionary.Item(key)` with values passed as parameters, except the value that represents the KEY & **Returns** a **True**. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Function Remove(ByVal key As String) As Boolean`	`String Key`	`True` `False`	▪ **Implementation of Remove(Key)** method. ▪ ***Wrapper Method*** that REMOVES the object from COLLECTION who's associated KEY is passed as parameter to method. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Contains(key)` Method determine if KEY exists. 2. If exists it calls `MyBase.Dictionary.Remove(key)` to do the work of REMOVING OBJECT from COLLECTION and returns a **TRUE**. 3. Else, if not exists, then it returns a **FALSE**. 4. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Function Print(ByVal key As String) As Boolean`	`String Key`	`True` `False`	▪ **Implementation of Print(Key)** method. ▪ ***Method*** that PRINTS the content of the OBJECT in the COLLECTION who's associated KEY is passed as parameter. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***DVD*** Class type. 3. Returns a FALSE if POINTER returned by `Dictionary.Item(key)` Property is a ***Nothing***. 4. Else CALLS the `object.Print()` Method of the OBJECT in the COLLECTION whose POINTER was returned by `MyBase.Dictionary.Item(key)`. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
Public Sub PrintAll()	None	**None**	▪ **Implementation of PrintAll()** method. ▪ ***Method*** that PRINTS the content of ALL THE OBJECT in the COLLECTION. ▪ Algorithm: 1. Uses **For Each objDictionaryEntry In MyBase.Dictionary** LOOP to iterate through the collection. 2. Use **CType()** Function to convert to native data type of DictionaryEntry object to ***DVD*** Class type. 3. CALLS the **object.Print()** Method of EACH OF THE OBJECT in the COLLECTION. 4. Add Error-Handling code using **Try-Catch-Finally** to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
Public Shadows Sub Clear()	None	**None**	▪ **Implementation of Clear()** method. ▪ **Method** uses **Shadows** keyword to Shadow the Base Class Method which already implements this functionality. We are simply repeating the process here. ▪ ***Method*** that CLEARS or DELETES ALL OBJECTS in the COLLECTION. ▪ Algorithm: 1. CALLS **MyBase.Dictionary.Clear()** to do the work. 2. Add Error-Handling code using **Try-Catch-Finally** to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

`#Region "Helper Methods"`
 	
`'Methods used to assist other methods or maintenance`
`Public Function ToArray() As DVD()`
`'Declare an empty array of DVD`
`Dim arrDVDList(MyBase.Dictionary.Count - 1) As DVD`
 	
`'Use Shared CopyTo() Method of Dictionary Class to convert Collection to Array`
 	
`MyBase.Dictionary.Values.CopyTo(arrDVDList, 0)`
`Return arrDVDList`
`End Function`
 	
`#End Region`
	None	**Pointer to DVD ARRAY DVD()**	▪ **Implementation of ToArray()** method.
			▪ **OPTIONAL FOR THOSE WHO WANT TO USED DATAVIEWGRIDS AND USE DATA BINDING.**
			▪ **Method** CONVERTS **MyBase.Dictionary** DICTIONARY COLLECTION OBJECT TO ARRAY by leveraging the **MyBase.Dictionary.Values.CopyTo()** method of the DICTIONARY CLASS.
			▪ Method the **Return** ARRAY so it can be used for data binding.
			▪ **CODE IS PROVIDED IN THE FIRST COLUNM, SIMPLY COPY AND USE**.
			▪ Algorithm:
			1. Creates a temporary ARRAY that holds ***DVD*** Objects.
			2. CALLS **MyBase.Dictionary.Values.CopyTo()** method to CONVERT THE COLLECTION TO ARRAY.
			3. **Return** the CONVERTED ARRAY.

Public Data Access Methods	Parameters	Return Type	Description
`Public Shared Function Create()`	`None`	`DVDList Reference or Pointer`	▪ **Implementation of `Shared Function Create()`.** ▪ OBJECT FACTORY METHOD. ▪ Creates & RETURNS FULLY READY AND INIALIZED ***DVDLIST*** OBJECT. - Objects created may include initialized data from database. ▪ STUB FUNCTION METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER, EMPTY BODY & RETURN KEYWORD WITH CODE TO SATISFY COMPILER.
`Public Sub Load()`	None	`None`	▪ **Implementation of Load()** method. ▪ LOAD the OBJECTS from the `DVDData.txt` file and ADDS them to the COLLECTION. ▪ Algorithm: 1. USE `File.Exists("DVDData.txt")` to verify if FILE EXISTS. 2. If FILE DOES NOT EXISTS IT CREATES IT using `File.Create("DVDData.txt")` Method. 3. Open File for READING. 4. Read a LINE from file & PARSE each comma-delimited line. 5. Create new temporary OBJECT if the DVD Class. 6. SET OBJECT WTH values from LINE READ FROM FILE. 7. ADD OBJECT TO COLLECTION. 8. Repeat this process until EOF. 9. Add Error-Handling code using `Try-Catch-Finally` to handle a ***GENERAL*** Exception.
`Public Sub Save()`	None	`None`	1. **Implementation of Save()** method. 2. SAVES the OBJECTS IN THE COLLECTION to the `DVDData.txt` file. 3. Algorithm: 4. Open File for WRITTING. 5. Uses `For Each objDictionaryEntry In MyBase.Dictionary` LOOP to iterate through the collection. 6. Use `CType()` Function to convert to native data type of `DictionaryEntry` object to ***DVD*** Class type 7. GETS ALL THE PROPERTIES FOR EACH OBJECT IN ARRAY and CREATES A ***Comma-delimited string*** from ALL THE PROPERTIES OF THE OBJECT IN ARRAY. 8. CALLS THE STREAMREADER OBJECT `WriteLine()` Method TO WRITE THE ***Comma-delimited string*** LINE TO FILE. 9. Repeat this process until ALL OBJECTS IN ARRAY HAVE BEEN VISITED AND ITS PROPERTIES WRITTEN TO THE FILE AS A ***Comma-delimited string***. 10. Add Error-Handling code using `Try-Catch-Finally` to handle a ***GENERAL*** Exception.

Class VideoGameList

- ***VideoGameList*** COLLECTION Class. Object of this class will store and manage **VideoGame Objects** in memory and load/save them from ***TEXT FILE***.
- This class encapsulates a ***DICTIONARY COLLECTION*** OBJECT by ***INHERITING*** from the ***DICTIONARYBASE CLASS***.
- The key properties/methods are described as follows:

General Information	Description
`Option Explicit On` `Option Strict On`	▪ Option Explicit and Option Strict should be On

General Class Information	Description
`Public Class VideoGameList Inherits DictionaryBase`	▪ Class that encapsulates a DICTIONARY COLLECTION OBJECT. ▪ This class is inherited from the ***DictionaryBase*** Collection Library. ▪ Note that you must import the ***System.Collection*** library.

Public Properties (GET/SET)	Description
`Public Shadows ReadOnly Property Count() As Integer`	READ-ONLY GET: returns NUMBER OF ELEMENTS OR OBJECT IN THE COLLECTION DICTIONARY. Property should shadow the base class equivalent. ▪ GET Algorithm: **1.** Calls BASE CLASS `MyBase.Dictionary.Count` Property.
`Public Property Item(ByVal key As String) As VideoGame`	▪ **Wrapper PROPERTY** that **GET & SET** *VideoGame* **OBJECTS in the DICTIONARY COLLECTION** ▪ This Property GETS the POINTER to the OBJECT in the COLLECTION based on its KEY. ▪ This Property SETS the OBJECT WHO'S KEY is passed as parameter with the OBJECT being assigned. ▪ GET Algorithm: **1.** CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. **2.** Use `CType()` Function to convert to native data type of DICTIONARY collection to ***VideoGame*** Class type. ▪ SET Algorithm: **1.** CALLS the BASE CLASS `MyBase.Dictionary.Contains(key)` Method determine if KEY exists. **2.** If exists it calls `MyBase.Dictionary.Item(key)` to do the work of SETTING the VALUE. **3.** Else, THROWS `Throw New System.ArgumentException("ID Not found")` EXCEPTION indicating KEY WAS NOT FOUND.

Public Methods	Parameters	Return Type	Description
`Public Sub Add(ByVal key As String, ByVal objVideoGame As VideoGame)`	`VideoGame objVideoGame`	`None`	▪ **Implementation of Add(object)** method. ▪ ***Wrapper Method*** that ADDS the object & its associated KEY passed as argument into the collection. ▪ In this case, the KEY is the ***IDNumber*** PROPERTY of the Object. ▪ It is assumed the Object if CREATED & POPULATED in the User-Interface or calling program when passed as argument to the method call. Method simply adds the object to the COLLECTION. ▪ Algorithm: 1. Calls `MyBase.Dictionary.Add(key, objVideoGame)` Method to add the object to Collection. 2. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Sub Add(ByVal x, ByVal y,ByVal z, etc.)`	`Variable for each required Parameter to SET PROPERTY of the VideoGame Class Object. Normally the paramters of the Parameterized Constructor.`	`None`	▪ **Implementation of Add(x,y,z etc.)** method. ▪ ***Wrapper Method*** that ADDS object into the collection. ▪ Same functionality as previous ADD, except NO populated OBJECT is passed as parameter, but the individual values that make up the OBJECT. ▪ The method itself creates the Object, populates it with the values from parameters and then ADDS the object to the COLLECTION with the KEY. ▪ In this case, the KEY is the ***IDNumber*** PARAMETER value of the method's parameter list. ▪ This version of ADD, requires less programming in the User-Interface. ▪ Algorithm: 1. Creates either DEFAULT Temporary ***VideoGame*** OBJECT & SETS the appropriate properties based on parameters VALUES passed to method. Or CREATES PARAMETERIZED Constructor OBJECT, passing to the OBJECT the VALUES of the parameters passed to method. 2. Calls `MyBase.Dictionary.Add(key, objVideoGame)` Method to add the object to Collection. The KEY being the ***IDNumber*** PROPERTY of the object created. 3. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Function Edit(ByVal key As String, ByVal objVideoGame As VideoGame) As Boolean`	`VideoGame objVideoGame`	`True` `False`	▪ **Implementation of Edit(object)** method. ▪ ***Wrapper Method*** that EDITS the OBJECT in the COLLECTION whose KEY is passed as parameter to method. ▪ In this case, the KEY is the ***IDNumber*** PROPERTY of the Object ▪ The OBJECT in COLLECTION is edited by SETTING its PROPERTIES, with the values or the PROPERTIES of the OBJECT passed as parameter. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***VideoGame*** Class type. 3. Returns a FALSE if POINTER returned by `MyBase.Dictionary.Item(key)` Property is a ***Nothing***. 4. Else, **GETS PROPERTIES** of Object passed as parameter & **SETS PROPERTIES** of OBJECT in the COLLECTION whose POINTER was returned by `Dictionary.Item(key)` Property & **Returns** a **True**. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Function Edit(ByVal x, ByVal y,ByVal z, etc.) As Boolean`	Variable for each required Parameter to SET PROPERTY of the ***VideoGame*** Class Object. To be EDITED.	`True` `False`	▪ **Implementation of Edit(x,y,z)** method. ▪ ***Wrapper Method*** that EDITS the OBJECT in the COLLECTION who's associated KEY is passed as ONE of the parameter variables. The OBJECT in COLLECTION is edited by SETTING its PROPERTIES with the values passed as parameters to method, except the KEY parameter. ▪ In this case, the KEY is the ***IDNumber*** PARAMETER value of the method's parameter list. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***VideoGame*** Class type. 3. Returns a FALSE if POINTER returned by `MyBase.Dictionary.Item(key)` Property is a ***Nothing***. 4. Else **SETS PROPERTIES** of OBJECT in the COLLECTION whose POINTER was returned by `MyBase.Dictionary.Item(key)` with values passed as parameters, except the value that represents the KEY & **Returns** a **True**. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Function Remove(ByVal key As String) As Boolean`	`String Key`	`True` `False`	▪ **Implementation of Remove(Key)** method. ▪ ***Wrapper Method*** that REMOVES the object from COLLECTION who's associated KEY is passed as parameter to method. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Contains(key)` Method determine if KEY exists. 2. If exists it calls `MyBase.Dictionary.Remove(key)` to do the work of REMOVING OBJECT from COLLECTION and returns a **TRUE**. 3. Else, if not exists, then it returns a **FALSE**. 4. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Function Print(ByVal key As String) As Boolean`	`String Key`	`True` `False`	▪ **Implementation of Print(Key)** method. ▪ ***Method*** that PRINTS the content of the OBJECT in the COLLECTION who's associated KEY is passed as parameter. ▪ Algorithm: 1. CALLS the BASE CLASS `MyBase.Dictionary.Item(key)` Property to return the POINTER to the OBJECT in Collection who's KEY is passed as argument. 2. Use `CType()` Function to convert to native data type of DICTIONARY collection to ***VideoGame*** Class type. 3. Returns a FALSE if POINTER returned by `Dictionary.Item(key)` Property is a ***Nothing***. 4. Else CALLS the `object.Print()` Method of the OBJECT in the COLLECTION whose POINTER was returned by `MyBase.Dictionary.Item(key)`. 5. Add Error-Handling code using `Try-Catch-Finally` to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

Public Methods	Parameters	Return Type	Description
`Public Sub PrintAll()`	None	**None**	▪ **Implementation of PrintAll()** method. ▪ ***Method*** that PRINTS the content of ALL THE OBJECT in the COLLECTION. ▪ Algorithm: 1. Uses **`For Each objDictionaryEntry In MyBase.Dictionary`** LOOP to iterate through the collection. 2. Use **`CType()`** Function to convert to native data type of `DictionaryEntry` object to ***VideoGame*** Class type. 3. CALLS the **`object.Print()`** Method of EACH OF THE OBJECT in the COLLECTION. 4. Add Error-Handling code using **`Try-Catch-Finally`** to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions
`Public Shadows Sub Clear()`	None	**None**	▪ **Implementation of Clear()** method. ▪ **Method** uses **Shadows** keyword to Shadow the Base Class Method which already implements this functionality. We are simply repeating the process here. ▪ ***Method*** that CLEARS or DELETES ALL OBJECTS in the COLLECTION. ▪ Algorithm: 1. CALLS **`MyBase.Dictionary.Clear()`** to do the work. 2. Add Error-Handling code using **`Try-Catch-Finally`** to handle all required ***COLLECTION*** & ***GENERAL*** Exceptions

`#Region "Helper Methods"`
 	
`'Methods used to assist other methods or maintenance`
`Public Function ToArray() As VideoGame()`
`'Declare an empty array of VideoGame`
`Dim arr VideoGameList(MyBase.Dictionary.Count - 1) As VideoGame`
 	
`'Use Shared CopyTo() Method of Dictionary Class to convert Collection to Array`
 	
`MyBase.Dictionary.Values.CopyTo(arr VideoGameList, 0)`
`Return arr VideoGameList`
`End Function`
 	
`#End Region`
	None	**`Pointer to DVD ARRAY VideoGame()`**	▪ **Implementation of ToArray()** method.
			▪ **OPTIONAL FOR THOSE WHO WANT TO USED DATAVIEWGRIDS AND USE DATA BINDING.**
			▪ **Method** CONVERTS **`MyBase.Dictionary`** DICTIONARY COLLECTION OBJECT TO ARRAY by leveraging the **`MyBase.Dictionary.Values.CopyTo()`** method of the DICTIONARY CLASS.
			▪ Method the **`Return`** ARRAY so it can be used for data binding.
			▪ **CODE IS PROVIDED IN THE FIRST COLUNM, SIMPLY COPY AND USE**.
			▪ Algorithm:
			1. Creates a temporary ARRAY that holds ***VideoGame*** Objects.
			2. CALLS **`MyBase.Dictionary.Values.CopyTo()`** method to CONVERT THE COLLECTION TO ARRAY.
			3. **`Return`** the CONVERTED ARRAY.

Public Data Access Methods	Parameters	Return Type	Description
`Public Shared Function Create()`	`None`	`VideoGameList` Reference or Pointer	▪ **Implementation of `Shared Function Create()`.** ▪ OBJECT FACTORY METHOD. ▪ Creates & RETURNS FULLY READY AND INIALIZED ***VIDEOGAMELIST*** OBJECT. - Objects created may include initialized data from database. ▪ STUB FUNCTION METHOD FOR FUTURE UPGRADE. ▪ CREATE THE HEADER, EMPTY BODY & RETURN KEYWORD WITH CODE TO SATISFY COMPILER.
`Public Sub Load()`	None	`None`	▪ **Implementation of Load()** method. ▪ LOAD the OBJECTS from the `VideoGameData.txt` file and ADDS them to the COLLECTION. ▪ Algorithm: 1. USE `File.Exists("VideoGameData.txt")` to verify if FILE EXISTS. 2. If FILE DOES NOT EXISTS IT CREATES IT using `File.Create("VideoGameData.txt")` Method. 3. Open File for READING. 4. Read a LINE from file & PARSE each comma-delimited line. 5. Create new temporary OBJECT if the ***VideoGame*** Class. 6. SET OBJECT WTH values from LINE READ FROM FILE. 7. ADD OBJECT TO COLLECTION. 8. Repeat this process until EOF. 9. Add Error-Handling code using `Try-Catch-Finally` to handle a ***GENERAL*** Exception.
`Public Sub Save()`	None	`None`	1. **Implementation of Save()** method. 2. SAVES the OBJECTS IN THE COLLECTION to the `VideoGameData.txt` file. 3. Algorithm: 4. Open File for WRITTING. 5. Uses `For Each objDictionaryEntry In MyBase.Dictionary` LOOP to iterate through the collection. 6. Use `CType()` Function to convert to native data type of `DictionaryEntry` object to ***VideoGame*** Class type 7. GETS ALL THE PROPERTIES FOR EACH OBJECT IN ARRAY and CREATES A ***Comma-delimited string*** from ALL THE PROPERTIES OF THE OBJECT IN ARRAY. 8. CALLS THE STREAMREADER OBJECT `WriteLine()` Method TO WRITE THE ***Comma-delimited string*** LINE TO FILE. 9. Repeat this process until ALL OBJECTS IN ARRAY HAVE BEEN VISITED AND ITS PROPERTIES WRITTEN TO THE FILE AS A ***Comma-delimited string***. 10. Add Error-Handling code using `Try-Catch-Finally` to handle a ***GENERAL*** Exception.

APPENDIX B– User-Interface & FORM DETAILS

User-Interface:

General User-Interface Requirements

- The User-Interface is found in the CLIENT EXECUTABLE.
- It is made up of the following:
  - MODULES
  - FORMS

Module Requirements

- The following Methods are mandatory in the Module:

Public Methods	Parameters	Return Type	Description
`Public Sub Main()`	None	`None`	▪ **Implementation of Sub Main()** method. ▪ ***Method*** that CONTROLS the FLOW of the application from start to end. ▪ Algorithm: **1.** Program prompt for *UserName* & *Password* using ***Login Form*** **2.** Grabs the value from ***Login Form*** and authenticates the username & password by CALLING the MODULE ***Authenticate(u,p)*** Method. **3.** If NOT FOUND, prompt user that "Access is Denied" and returns to the ***Login Form*** of step 1. **4.** If FOUND, loads ***Main Screen***. **5.** After all Main Screen processing is done, it returns back to ***Login Form*** so process is repeated again with the next user. **6.** If required, add Error-Handling code using `Try-Catch-Finally` to handle all required Exceptions.

`Public Function Authenticate(ByVal Username As String, ByVal Password As String) As Boolean`	None	`Boolean`
		`True`
		`False`
			▪ **Implementation of Authenticate()** method.
			▪ **Method** that CONTROLS the authentication of the Username & Password.
			▪ In this implementation, this method simply calls on the `objEmployeeList` Object ***Authenticate(u,p)*** method to do the work.
			▪ Algorithm:
			1. CALLS CUSTOM ARRAY CLASS OBJECT `objEmployeeList.Authenticate(U,P)` **Method** to do the work of authenticating the user.
			2. From the results of the CALL to `objEmployeeList.Authenticate(U,P)` method which should be either a `Tre` or `False`, it decides whether to return a `Tre` or `False` to the calling program.
			3. Add Error-Handling code using `Try-Catch-Finally` to handle all required Exceptions.

Form Design & Navigation:

General User-Interface Requirements

Forms Requirements

- You are required to create the following 8 Forms:
  - Login Form:
    - ***Login Screen***
  - Front-End Main & Application Selection Forms:
    - ***Main Welcome Form***
    - ***Video POS System Form***
    - ***Back-End Management Form***
  - Back-End Management Forms:
    - ***Employee Management Form***
    - ***Customer Management Form***
    - ***DVD Management Form***
    - ***Video Game Management Form***
- You can ADD ANY ADDITIONAL FORM YOU MAY WANT OR NEED IN ORDER TO IMPLEMENT YOUR DESIGN.

Form Design Requirements

- You can be creative as far as the design:
  - Type of Form you choose
  - Control Layout, etc.
  - Color, appearance, etc.

User-Interface Control Requirements

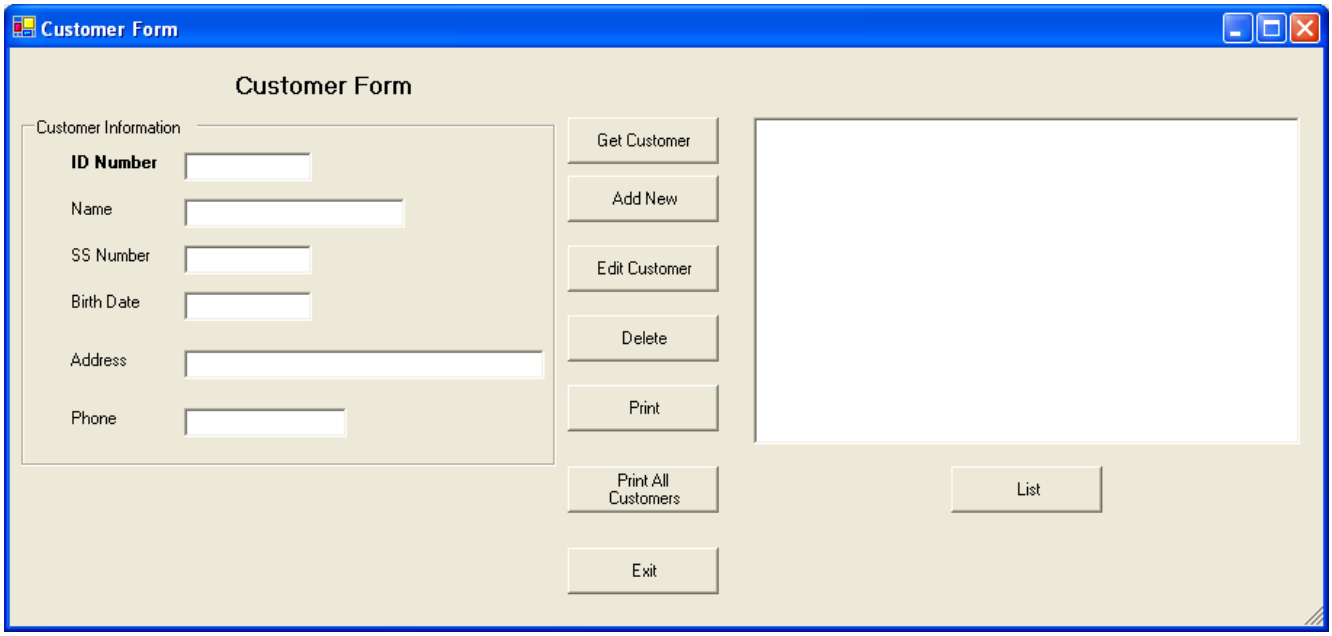
- As far as the control themselves:

Textboxes:

- Textboxes:
  - Textboxes are easy to use and recommended, but you are not oblige to use the in all cases.
  - Textboxes can be used to enter data for submitting or displaying data results:
    - For example, when searching you need to enter either the ***SSNumber*** for ***Employee Management*** or ***IDNumber*** for ***Customer Management***, ***DVD Management*** & ***VideoGame Management***.
    - But the search results return the object's record or properties. Not all the properties, but only what's required. For example, Count, is not a property to display on the Form.
    - Point is you will also need textboxes to display the resultant properties.
    - Your options are:
      - You can create a separate text box to enter the data and separate textboxes to display the results. Think of a browser such as Bing or Google, where there is a search box and results appear below or in other section.

      - Or combined both functionalities, which means that the same text box to enter the SSNumber or IDNumber for the search, and also return the resultant ID number to the same textbox, etc.

      - For example, in my ***Small Business Application*** Example, I use the following management Form style:



      - As you can see, the same ***ID number*** text box I use to enter the data, is the same text box I used to receive the data for ID Number being returned.
      - I also include other textboxes to receive the remaining appropriate properties of the Customer Object.
      - The same set of text boxes will also be used to receive input for ADD, Edit etc.

  - Therefore the number of controls you will need can be designed in different ways, it is up to you to decide how you want to do this. You can follow my example above or come up with your own!

Mandatory Controls (ComboBox)

- Some controls such as ***ComboBox*** in some forms such as ***DVD Management*** & ***VideoGame Management*** are MANDATORY AND YOU MUST IMPLEMENT THESE AS REQUIRED for the following features:

  - **Rating:** ***G, PG, PG-13, NC-17, R , None***
  - **Movie Category:** ***Action_Adventure, Drama, Famil_Kids, Horror, Sci-Fi_Fantasy, Music, Sports, Romance, Comedy, Western, None***
  - **DVD Format:** ***DVD, HD-DVD, BLU-RAY DISC, None***
  - **Video Game Category:** ***Action, Role-playing, Shooting, Fighting, Racing, Sports, Strategy, Horror, Flight Simulators, Online, Rhythm, None***
  - **Video Game Format:** ***X-Box, X-Box 360, PS3, PS2,GameCube, DS, Wii, PC, None***

Other Controls

- For the Management Forms LIST feature, you are given the option to choose from the following:

  - **ListBox:** Use a list box as my example above
  - **DataViewGrid:** You also have the option of using a Data Grid Control, which shows results as a table. This is a little more complex, but if you know how to do it and want to leverage this you are free to do so.

Final Words on Controls

- You have flexibility as to how you want to design your Form and what controls to use with the exception of the ***ComboBoxes*** which are mandatory.

Displaying & Working Multiple Forms

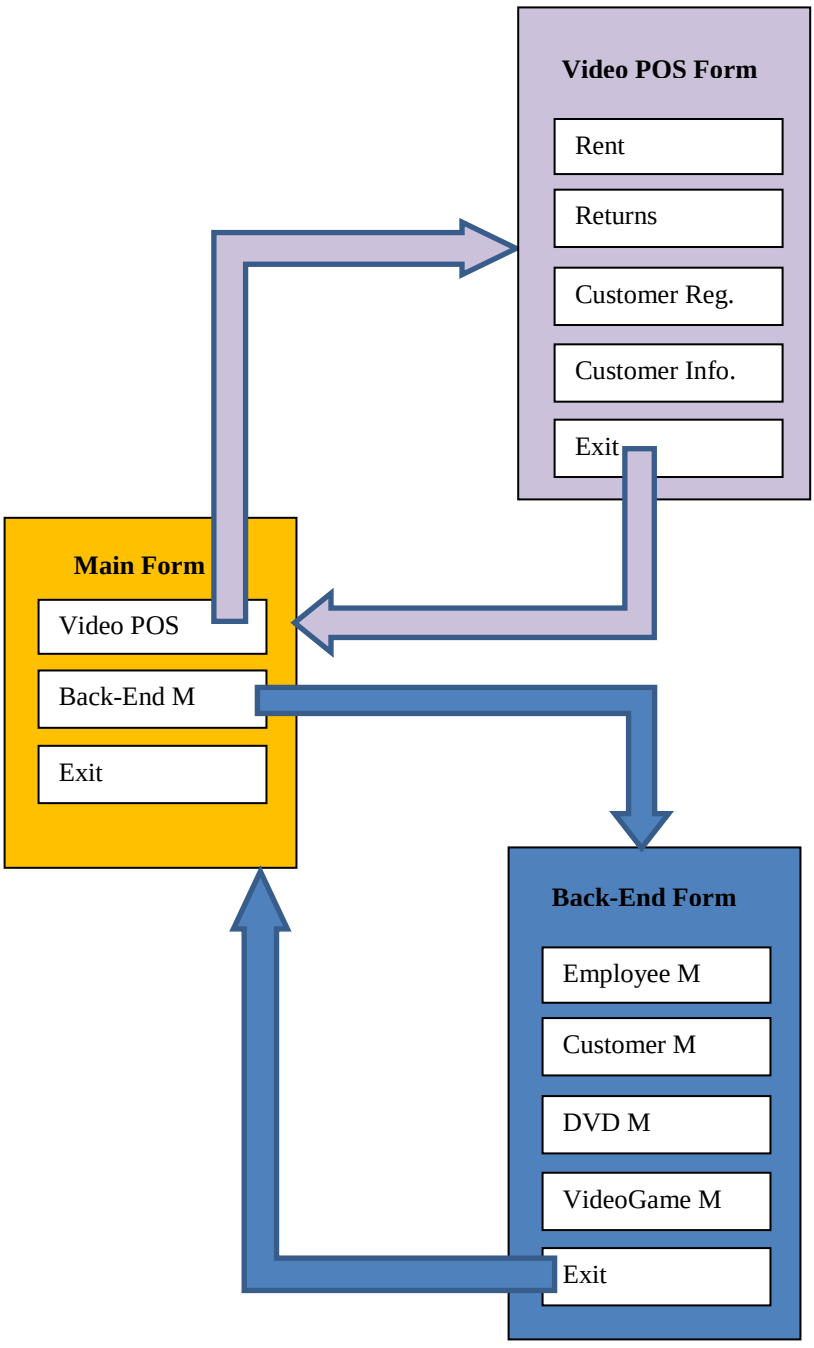
- In this project you will be working and displaying multiple Forms
- In other words PARENT CHILD RELATIONSHIPS BETWEEN FORMS.
- For example, the USER NAVIGATION FLOW between the ***Main Welcome Form*** & ***Video POS Form*** is as follows:

1. ***Main Welcome Form*** **Video POS Button Event-Handler Code** *INVOKES* the ***Video POS Form***
2. ***Video POS Form*** **Exit Button Event-Handler Code** will return the user back to the ***Main Welcome Form***

- And the USER NAVIGATION FLOW between the ***Main Welcome Form*** & ***Back-End Form*** is as follows:

1. ***Main Welcome Form*** **Back-End Management Button Event-Handler Code** *INVOKES* the ***Back-End Management Form***
2. ***Back-End Management Form*** **Exit Button Event-Handler Code** will return the user back to the ***Main Welcome Form***

- Diagram below illustrates this NAVIGATION FLOW:



Designing the Navigation for Multiple Forms (Parent & Child Forms)

- In order to implement a navigation strategy for an application you need to keep in mind the following basic concepts:

  - **STEP 1 –** Forms are **CLASSES**.
    - When you are designing your FORM you are building a **CLASS**.

  - **STEP 2 –** To run or execute your Form, you must CREATE an **OBJECT** of the **Form CLASS** you just created.
    - When the program is running it is the FORM OBJECT that you are manipulating or using.

  - **STEP 3 –** A RUNNING FORM is an **OBJECT** of the **Form CLASS** you created being **USED**

    - **SET/GET Properties of the FORM OBJECT:**
      - Form Name, Label etc:
        - **Name Property – To SET the name of Form OBJECT to manipulate**.
        - **Text Property – To SET the Name displayed on the Form**.

      - Color, design.
        - **BackColor Property – To SET the name of Form OBJECT to manipulate**.
        - **BackGroundImage Property – To SET the Name displayed on the Form**.

    - **CALL Methods of the FORM OBJECT:**
      - Displaying Forms:**,** etc.
        - **ShowDialog() – Displays Form as MODAL FORM**.
          - Form displays:
            - But you cannot continue with program UNTIL FORM IS **CLOSED** (A**ll resource CREATED WITHIN THE OBJECT ARE CLOSED AND NO LONGER AVAILABLE, YOU WOULD NEED TO CALL ShowDialog() AGAIN**)
            - OR **HIDDEN** (**OBJECT IS NOT VISIBLE, BUT STILL EXIST AND ACCESSIBLE, all resources are still being used by OBJECT, is JUST HIDDEN**).

          - As form DISPLAYS via **ShowDialog()**, ACCESS TO OTHER FORMS or the FORM THAT INVOKED THE FORM DISPLAYED IS NOT AVAILABLE, YOU MUST CLOSED OR HIDE THE FORM FOR THE PROGRAM TO CONTINUE.
          - **IMPORTANT!** If a FORM is display as NODAL or via **ShowDialog()** and you HIDE IT AGAIN, you CANNOT DISPLAY AS **ShowDialog()** YOU MUST CALL THE **Show()** METHOD INSTEAD.

        - **Show() – Displays FORM as NON-MODAL**.

          - Form displays, AND YOU HAVE ACCESS TO OTHER FORMS, ETC.
          - PROGRAM FLOW DOES NOT STOP.

      - Closing Forms: **Close()**

    - **TRIGGER EVENTS – Execute Event-Handlers:**
      - Form Loading & Closing**:**
        - **Load Event – Occurs when Form is LOADING or Displaying**.
        - **FormClosing Event – Occurs when Clicking the X button or FORM is Closing**.

      - User interaction**:**
        - **Click Event – Occurs when user CLICKS a BUTTON CONTROL**.
        - **MouseClick Event**, **MouseDoubleClick Event** etc **– Occurs during Mouse interactions, etc.**

    - **Object-To-Object Interactions – Parent/Child Objects.**
      - Multiple Forms**:**

- **Form invokes another Form to DISPLAY – Calling or INVOKING FORM is the PARENT, FORM BEING DISPLAY IS THE CHILD**.

- From the above concept we derive the following RULES of FORM NAVIGATION:

1. A PARENT FORM is a FORM that INVOKES OR DISPLAYS ANOTHER FORM. THE FORM DISPLAYED IS THE CHILD FORM.

2. In order to display THE CHILD FORM, you need to create an OBJECT of the FORM, from within THE PARENT FORM and call either **ShowDialog()** or **Show()** Methods to DISPLAY THE CHILD FORM.

3. HOW TO RETURN BACK TO PARENT FORM FROM A CHILD FORM DISPLAYED VIA **ShowDialog()** **WITH THE PARENT FORM STILL DISPLAYING IN THE BACKGROUND**:

   - If you CREATE A CHILD FORM OBJECT, from within a PARENT FORM OBJECT and call the CHILD OBJECT'S **ShowDialog()** method AND YOU WANT THE PARETN FORM TO STILL DISPLAYING IN THE BACKGROUND,.

     1) THEN YOU SIMPLY CALL **ShowDialog()**.
     - THE PROGRAM EXECUTION STOPS RIGHT AT THE LOCATION IN THE PARENT FORM WHERE **ShowDialog()** WAS CALLED
     - The PARENT FORM STILL SHOWS IN THE BACKGROUND, YOU HAVE NO ACCESS TO IT OR ANY OTHER CODE IN THE BACKGROUND

   - Once the CHILD FORM is DISPLAY and the user is finished using the FORM, you need to RETURN BACK TO THE PARENT FORM.
   - At this point you have TWO OPTIONS, either CLOSING or HIDDING THE CHILD FORM. Each option has its; pros and cons.

     a) CLOSING THE CHILD FORM (**BEST OPTION FOR MANAGEMENT FORMS**):

     - If the CHILD FORM CLOSES ITSELF, control is passed back to the PARENT FORM, since it was the PARENT FORM THAT DISPLAYED THE FORM.
     - IMPORTANT! The point here is the CLOSING THE FORM **CLOSES ALL RESOUCES TO THE FORM THAT WAS JUST DISPLAYED**.
     - **A BENEFIT OF CLOSING CHILD FORM is that you can leverage the FORM_CLOSE or FORM_CLOSING Event-Handlers to add code to perform CLEANUP OR ANY OTHER PROCESS YOU WANT WHEN THE FORM CLOSES**.
     - Inside the PARENT FORM, THE CODE AFTER THE CALL TO **ShowDialog()** will now continue to execute.

     b) HIDING THE CHILD FORM:

     - If you HIDE THE CHILD FORM, control is passed back to PARENT FORM as previous option .
     - IMPORTANT! The point here is that HIDDING THE FORM **ALL RESOUCES TO THE FORM are STILL AVAILABLE, FOR IS JUST HIDDING.**
     - **DEPENDING ON THE SCENARIO, THIS MAY BE A BAD OPTION BECAUSE THE FORM_CLOSE or FORM_CLOSING EVENT DO NOT EXECUTE, SO IF YOU HAVE CLEANUP CODE THERE THEY WILL NOT EXECUTE, THUS THE PROGRAM MAY NOT FUNCTION PROPERTLY**.
     - Inside the PARENT FORM, THE CODE AFTER THE CALL TO **ShowDialog()** will now continue to execute.

4. HOW TO RETURN BACK TO PARENT FORM FROM A CHILD FORM DISPLAYED VIA **ShowDialog()** **WITH THE PARENT FORM HIDDING IN THE BACKGROUND**:

- If you CREATE A CHILD FORM OBJECT, from within a PARENT FORM OBJECT and call the CHILD OBJECT'S **ShowDialog()** method **AND YOU WANT THE PARENT FORM NOT TO DISPLAY IN THE BACKGROUND, THEN YOU NEED TO HIDE THE PARENT FORM FIRST:**

  1) PARENT FORM must HIDE itself using **Me.Hide()**.

  - PARETN FORM IS HIDDEN IN THE BACKGROUND, THE PROGRAM EXECUTION STOPS RIGHT at the call to **ShowDialog()**

  2) PARENT FORM CALLS **ShowDialog()** method

- Once the CHILD FORM is DISPLAY and the user is finished using the FORM, you need to RETURN BACK TO THE PARENT FORM.
- At this point you have TWO OPTIONS, either CLOSING or HIDDING THE CHILD FORM. Each option has its; pros and cons:

  a) CLOSING THE CHILD FORM (**BEST OPTION FOR MANAGEMENT FORMS**):

  - If the CHILD FORM CLOSES ITSELF, control is passed back to the PARENT FORM, since it was the PARENT FORM THAT DISPLAYED THE FORM.
  - IMPORTANT! The point here is the CLOSING THE FORM **CLOSES ALL RESOUCES TO THE FORM THAT WAS JUST DISPLAYED**
  - **A BENEFIT OF CLOSING CHILD FORM is that you can leverage the FORM_CLOSE or FORM_CLOSING Event-Handlers to add code to perform CLEANUP OR ANY OTHER PROCESS YOU WANT WHEN THE FORM CLOSES**.
  - Inside the PARENT FORM, THE CODE AFTER THE CALL TO **ShowDialog()** will now continue to execute.

  b) HIDING THE FORM:

  - If you HIDE THE CHILD FORM, control is passed back to PARENT FORM as previous option.
  - IMPORTANT! The point here is that HIDDING THE FORM **ALL RESOUCES TO THE FORM are STILL AVAILABLE, FOR IS JUST HIDDING.**
  - **DEPENDING ON THE SCENARIO, THIS MAY BE A BAD OPTION BECAUSE THE FORM_CLOSE or FORM_CLOSING EVENT DO NOT EXECUTE, SO IF YOU HAVE CLEANUP CODE THERE THEY WILL NOT EXECUTE, THUS THE PROGRAM MAY NOT FUNCTION PROPERTLY**.
  - Inside the PARENT FORM, THE CODE AFTER THE CALL TO **ShowDialog()** will now continue to execute.

  3) PARENT FORM CALLS **Show()** method to SHOW ITSELF AGAIN, since THE PARENT FORM IS ALREADY BEING SHOWN AS A NODAL FORM WHEN IT WAS CALLED BY IT'S PARENT AS **ShowDialog()**, YOU CANNOT CALL **ShowDialog()** AGAIN.

Algorithm for Main Welcome Form & Back-End Management form Example

- Based on the rules listed above, I will show the algorithm for the navigation flow between the ***Main Welcome Form*** and the ***Back-End Management Form***.

- The USER NAVIGATION FLOW between the ***Main Welcome Form*** & ***Back-End Form*** is as follows:

  1. ***Main Welcome Form*** **Back-End Management Button Event-Handler Code** *INVOKES* the ***Back-End Management Form***
  2. ***Back-End Management Form*** **Exit Button Event-Handler Code** will return the user back to the ***Main Welcome Form***

- Below we will look at the two options, displaying the PARENT FORM in the BACKGROUND or HIDDING THE PARENT FORM IN THE BACKGROUND.

Option 1 – PARENT FORM Main Welcome Form Displays in the BACKGROUND While CHILD FORM Back-End Management Form Displays in the FOREFRONT

- **STEP 0 - This algorithm assumes that you have already CREATED OR DESIGNED YOUR FORM CLASS**.

- PARENT FORM ***Main Welcome Form*** **Back-End Management Button Event-Handler Code** Algorithm:

1. **CREATE OBJECT** of ***Back-End Management Form***
2. CALL **ShowDialog()** **METHOD OF THE CHILD OBJECT TO DISPLAY IT AS A NODAL FORM**
3. (Do nothing after this, let the Event-Handler End)

- **NOTE, nothing else needs to be done inside the Event-Handler.**
- **ONCE THE CHILD FORM CLOSES, CONTROL IS PASSED BACK TO THE CODE AFTER THE CALL TO ShowDialog() OR STEP 3.**
- **AT THIS POINT NOTHING NEEDS TO BE DONE, LET THE EVENT-HANDLER END.**

- CHILD FORM ***Back-End Management Form*** **Exit Button Event-Handler Code** Algorithm:

1. CHILD FORM CLOSES ITSELF by calling the ***METHOD*** **Me.Close()**

- **AT THIS POINT, CONTROL IS PASSED BACK TO THE PARENT FORM *Main Welcome Form* Back-End Management Button Event-Handler Code TO THE INSTRUCTION AFTER THE CALL TO ShowDialog() WAS MADE**
- **NOTE, THAT YOU CAN LEVERAGE the FORM_CLOSE or FORM_CLOSING Event-Handlers to add code to perform CLEANUP OR ANY OTHER PROCESS YOU WANT WHEN THE FORM CLOSES, SUCH AS SAVING & CLEARING THE OBJECTS IN THE ARRAY, ETC.**

Option 2 – PARENT FORM Main Welcome Form HIDES in the BACKGROUND While CHILD FORM Back-End Management Form Displays in the FOREFRONT

- **STEP 0 - This algorithm assumes that you have already CREATED OR DESIGNED YOUR FORM CLASS**.

- PARENT FORM ***Main Welcome Form*** **Back-End Management Button Event-Handler Code** Algorithm:

1. **CREATE OBJECT** of ***Back-End Management Form***
2. PARENT FORM **HIDES** ITSELF by calling the METHOD **Me.Hide()**
2. CALL **ShowDialog() METHOD OF THE CHILD OBJECT TO DISPLAY IT AS A NODAL FORM**
3. PARENT FORM **SHOWS** ITSELF by calling the METHOD **Me.Show()**
4. (Do nothing after this, let the Event-Handler End)

- **NOTE, nothing else needs to be done inside the Event-Handler.**
- **ONCE THE CHILD FORM CLOSES, CONTROL IS PASSED BACK TO THE CODE AFTER THE CALL TO ShowDialog() OR STEP 3, IN THIS CASE WHERE THE PARENT FORM HAS TO SHOW ITSELF AGAIN USING Me.Show().**
- **AT THIS POINT NOTHING NEEDS TO BE DONE, LET THE EVENT-HANDLER END.**

- CHILD FORM ***Back-End Management Form*** **Exit Button Event-Handler Code** Algorithm:

3. CHILD FORM CLOSES ITSELF by calling the ***METHOD*** **Me.Close()**

- **AT THIS POINT, CONTROL IS PASSED BACK TO THE PARENT FORM *Main Welcome Form* Back-End Management Button Event-Handler Code TO THE INSTRUCTION AFTER THE CALL TO ShowDialog() WAS MADE**
- **NOTE, THAT YOU CAN LEVERAGE the FORM_CLOSE or FORM_CLOSING Event-Handlers to add code to perform CLEANUP OR ANY OTHER PROCESS YOU WANT WHEN THE FORM CLOSES, SUCH AS SAVING & CLEARING THE OBJECTS IN THE ARRAY, ETC.**

Conclusion

- All the other navigation flow uses the same algorithm as shown above. With the exception of the LOGIN FORM which is being displayed and managed by the MAIN PROGRAM IN THE MODULE.

Detailed Form Descriptions:

Login Form Screen:

- This is the screen the employee will see after the main screen if they wish to use the Video Management System:

  - ***Provides controls for user to enter Username***
  - ***Provides controls for user to enter Username***
  - ***Provides opton to initiate the login (OK)***
  - ***Provides option to clear all text boxes (Cancel)***

- Implementation details:

  - Design as you like.
  - No special requirements
  - Recommendation it to use ***TEXTBOXES***, ***LABELS*** and ***BUTTON CONTROLS***

Main Screen:

- This is the main screen to the program, with options for:

  - ***Video POS System Option:*** Selecting this option will take you to the Point-of-Sales Screen
  - ***Back-End Management Options:*** Selecting this option will take you to the Back-End Management screens (described below)
  - ***Exit Option:*** User is ready to exit and return to Login Screen

- Implementation details:

  - Design as you like.
  - Recommendations is to use ***LABELS*** and ***BUTTON CONTROLS*** for the selections, or some other method you feel would work.

Video POS Screen:

- Main ***Video Point-Of-Sales*** screen for future implementation with option for:

  - ***Rent:*** Selecting this option will allow employee to perform rental transaction (Note, this may require other user interfaces to interact with user to handle the request).
  - ***Return:*** Selecting this option will allow employee to perform rental transaction. (Note, this may require other user interfaces to interact with user to handle the request)
  - ***Customer Information:*** Selecting this option will display a customer record (Note, this may require other user interfaces to interact with user to handle the request).
  - ***Customer Registration:*** Selecting this option will register a new customer (Note, this may require other user interfaces to interact with user to handle the request).
  - ***Exit Option:*** User is ready to exit and return to the previous screen.

- Implementation details:

  - Design as you like.
  - Recommendations is to use ***LABELS*** and ***BUTTON CONTROLS*** for the selections, or some other method you feel would work.
  - Do not implement features, simply user interface.

Employee Management Screen:

- The screen to manage the employee data via the ***objEmployeeList*** *object*. The screen has the following requirements:

  - The ***Employee Management screens*** should allow for backend database support, where employees can manage the employees individual records by adding, removing, editing, listing employee records etc. The backend screen should contain the necessary UI controls, text & code to implement all the options shown in table below.

GENERAL FORM Information	Description
▪ **Management Form** ▪ **Created from Form Class** ▪ **You will need to create an OBJECT of this FORM & call the OBJECT.`ShowDialog()` method to display the Form as a Modal dialog box.**	▪ Use Windows Form to implement. ▪ Choose any of the desired Form, standard, MDI, etc., but keep functionality shown below. ▪ Choose your own colors etc. ▪ You are the visual designer. ▪ Some functionality will require special CONTROLS. See table below for description and requirements.

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
Loading or Displaying Form: ▪ As form is displaying, the COLLECTION CLASS OBJECT **objEmployeeList** should be POPULATED with RECORDS from permanent storage or DATABASE, in this case the ***EmployeeData.txt*** file. ▪ The LOADING is done by the ***Business Object Layer*** via ***objEmployeeList*** Collection Class Object	1. Call the **objEmployeeList.Load()** method to load the data from the ***EmployeeData.txt*** file.	▪ Done from ***Form_Load Event-Handler***.	
SEARCH: ▪ Search the database for the employee record who's KEY or ***SSNumber*** is entered on the Form Textbox control. ▪ The SEARCH is done by the ***Business Object Layer*** via ***objEmployeeList*** Collection Class Object	**1.** Grabs the ***SSNumber*** from the Form's ***TextBox***. 2. **Calls objEmployeeList .Item(SSNUMBER)** property to retrieve a POINTER to object stored in Collection. 3. Displays content of OBJECT'S PROPERTIES being pointed by POINTER returned from Search on ***TextBoxes*** in the Form. 4. If object is NOT FOUND, displays a message indicating "***Employee*** not found"	▪ **Button Control** – for user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***SSNumber*** for SEARCH. ▪ **TextBoxes Controls** – to display the properties of the Object pointed by the POINTER returned from SEARCH() call ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	
ADD: ▪ ADDS the NEW RECORD entered & displayed on the FORM. ▪ Selecting the ADD button will trigger process and ADD the OBJECT in the COLLECTION based on its KEY or ***SSNumber***. ▪ The ADD is done by the ***Business Object Layer*** via ***objEmployeeList*** Collection	**1.** Grabs the ***all values*** from the Form's ***TextBoxes*** that contain the record**.** 2. Calls **objEmployeeList .Add()** method to do the work.	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBoxes Controls** – to accept data that make up the RECORD to be ADD to COLLECTION. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ In the ***EmployeeList*** Class, there are two ADD methods to choose from. ▪ Select the appropriate ***ADD()*** method**.**

Class Object.			

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
EDIT: ▪ EDITS the current RECORD being displayed on the FORM. ▪ Selecting the EDIT button will trigger process and modify the OBJECT in the COLLECTION based on its KEY or ***SSNumber***. ▪ The EDIT is done by the ***Business Object Layer*** via ***objEmployeeList*** Collection Class Object.	1. **Grabs the** *SSNumber* **from the Form's** *TextBox*. 2. Calls **objEmployeeList.Edit()** method in module to do the work 3. If object is NOT FOUND, displays a message indicating "***Employee*** not found"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBoxes Controls** – to accept data that make up the RECORD to be EDITED in the COLLECTION. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ In the ***EmployeeList*** Class, there are two EDIT methods to choose from. ▪ Select the appropriate ***EDIT()*** method**.**
REMOVE: ▪ DELETES the Object in the COLLECTION, who's KEY or ***SSNumber*** is entered on the Form Textbox control. ▪ The DELETE is done by the ***Business Object Layer*** via ***objEmployeeList*** Collection Class Object	1. **Grabs the** *SSNumber* **from the Form's** *TextBox*. 2. Calls **objEmployeeList .Remove()** method to do the work 3. If object is NOT FOUND, displays a message indicating "***Employee*** not found"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***SSNumber*** of OBJECT to be deleted. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪
PRINT: ▪ PRINTS the Object in the COLLECTION, who's KEY or ***SSNumber*** is entered on the Form Textbox control to the ***Network_Printer.txt*** file. ▪ The PRINT is done by the ***Business Object Layer*** via ***objEmployeeList*** Collection Class Object	1. **Grabs the** *SSNumber* **from the Form's** *TextBox*. 2. Calls **objEmployeeList.Print()** method to do the work 3. If object is NOT FOUND, displays a message indicating "***Employee*** not found"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***SSNumber*** of OBJECT to be deleted. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪
PRINT ALL: ▪ PRINTS ALL Objects in the COLLECTION to the ***Network_Printer.txt*** file. ▪ The PRINT is done by the ***Business Object Layer*** via ***objEmployeeList*** Collection Class Object	1. Calls **objEmployeeList.PrintAllEmployee()** method to do the work	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
LIST ALL: ▪ LIST of all ***employee*** records ON THE FORM to a ***ListBox*** Control as a comma-delimited string. ▪ The LISTING is done by the FORM getting the data from ***Business Object Layer*** via ***objEmployeeList*** Collection Class Object	1. Clears the ***ListBox*** Control. 2. Loops through the COLLECTION in the **objEmployeeList)** OBJECT 3. During each LOOP, extract all the PROPERTIES of the current OBJECT in the COLLECTION. 4. Creates a comma-delimited string from ALL THE PROPERTEIS. 5. ADD & displays the string on the ListBox. 6. Repeats the process for ALL OBJECTS in the COLLECTION.	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **ListBox Control** – to store and display as a ***comma-delimited*** line the content of each of the OBJECTS in the COLLECTION. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ If you feel you can use a DataGridView Control instead of ListBox, you can do so. ▪ Future implementation will use the DataGridView.
Exiting or Closing Form (SAVING THE DATA): ▪ As form is closing, the COLLECTION CLASS OBJECT **objEmployeeList** should be SAVED to permanent storage or DATABASE, in this case the ***EmployeeData.txt*** file. ▪ The SAVING is done by the ***Business Object Layer*** via ***objEmployeeList*** Collection Class Object	1. Call the **objEmployeeList .Saves()** method to do the work of Saving the content of the COLLECTION to database, in this case the ***EmployeeData.txt*** File.	▪ Done from ***Form_Load Event-Handler***.	
Exiting or Closing Form (CLEARING THE COLLECTION): ▪ As form is closing, the COLLECTION CLASS OBJECT **objEmployeeList** should be CLEARD of all OBJECTS, so is ready to be loaded the next time around. ▪ The CLEARING is done by the ***Business Object Layer*** via ***objEmployeeList*** Collection Class Object	1. Call the **objEmployeeList .Clear()** method to remove all objects from the COLLECTION.	▪ Done from ***Form_Load Event-Handler***.	

Customer Management Screen:

- The ***Customer Management Screen*** Form has the following requirements:

  - Customers can be managed by employees via the ***Customer Management Screen***, which provides backend database support to manage customer records and perform operations such as adding, removing, editing, listing customer records etc. The backend screen should contain the necessary UI controls & code to implement all management options.

GENERAL FORM Information	Description
▪ **Management Form** ▪ **Created from Form Class** ▪ **You will need to create an OBJECT of this FORM & call the OBJECT.`ShowDialog()` method to display the Form as a Modal dialog box.**	▪ Use Windows Form to implement. ▪ Choose any of the desired Form, standard, MDI, etc., but keep functionality shown below. ▪ Choose your own colors etc. ▪ You are the visual designer. ▪ Some functionality will require special CONTROLS. See table below for description and requirements.

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
Loading or Displaying Form: ▪ As form is displaying, the COLLECTION CLASS OBJECT **objCustomerList** should be POPULATED with RECORDS from permanent storage or DATABASE, in this case the ***CustomerData.txt*** file. ▪ The LOADING is done by the ***Business Object Layer*** via ***objCustomerList*** Collection Class Object	1. Call the **objCustomerList.Load()** method to load the data from the ***CustomerData.txt*** file.	▪ Done from ***Form_Load Event-Handler***.	
SEARCH: ▪ Search the database for the ***CUSTOMER*** record who's KEY or ***IDNumber*** is entered on the Form Textbox control. ▪ The SEARCH is done by the ***Business Object Layer*** via ***objCustomerList*** Collection Class Object	**1.** Grabs the ***IDNumber*** from the Form's ***TextBox***. 2. **Calls objCustomerList .Item(SSNUMBER)** Property to retrieve a POINTER to object stored in Collection. 3. Displays content of OBJECT'S PROPERTIES being pointed by POINTER returned from Search on ***TextBoxes*** in the Form. 4. If object is NOT FOUND, displays a message indicating "**Customer not found**"	▪ **Button Control** – for user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***IDNumber*** for SEARCH. ▪ **TextBoxes Controls** – to display the properties of the Object pointed by the POINTER returned from ***SEARCH()*** call. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	
ADD: ▪ ADDS the NEW RECORD entered & displayed on the FORM. ▪ Selecting the ADD button will trigger process and ADD the OBJECT in the COLLECTION based on its KEY or ***IDNumber***. ▪ The ADD is done by the ***Business Object Layer*** via ***objCustomerList*** Collection	**1.** Grabs the ***all values*** from the Form's ***TextBoxes*** that contain the record**.** 2. Calls **objCustomerList .Add()** method to do the work.	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBoxes Controls** – to accept data that make up the RECORD to be ADD to COLLECTION. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ In the ***CustomerList*** Class, there are two ADD methods to choose from. ▪ Select the appropriate ***ADD()*** method**.**

Class Object.			

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
EDIT: ▪ EDITS the current RECORD being displayed on the FORM. ▪ Selecting the EDIT button will trigger process and modify the OBJECT in the COLLECTION based on its KEY or ***IDNumber***. ▪ The EDIT is done by the ***Business Object Layer*** via ***objCustomerList*** Collection Class Object.	1. **Grabs the *IDNumber* from the Form's** *TextBox*. 2. Calls **objCustomerList.Edit()** method in module to do the work 3. If object is NOT FOUND, displays a message indicating "***Customer not found***"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBoxes Controls** – to accept data that make up the RECORD to be EDITED in the COLLECTION. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ In the ***CustomerList*** Class, there are two EDIT methods to choose from. ▪ Select the appropriate ***EDIT()*** method**.**
REMOVE: ▪ DELETES the Object in the COLLECTION, who's KEY or ***IDNumber*** is entered on the Form Textbox control. ▪ The DELETE is done by the ***Business Object Layer*** via ***objCustomerList*** Collection Class Object	1. **Grabs the *IDNumber* from the Form's** *TextBox*. 2. Calls **objCustomerList .Remove()** method to do the work 3. If object is NOT FOUND, displays a message indicating "***Customer not found***"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***IDNumber*** of OBJECT to be deleted. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪
PRINT: ▪ PRINTS the Object in the COLLECTION, who's KEY or ***IDNumber*** is entered on the Form Textbox control to the ***Network_Printer.txt*** file. ▪ The PRINT is done by the ***Business Object Layer*** via ***objCustomerList*** Collection Class Object	1. **Grabs the *IDNumber* from the Form's** *TextBox*. 2. Calls **objCustomerList.Print()** method to do the work 3. If object is NOT FOUND, displays a message indicating "***Customer not found***"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***IDNumber*** of OBJECT to be deleted. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪
PRINT ALL: ▪ PRINTS ALL Objects in the COLLECTION to the ***Network_Printer.txt*** file. ▪ The PRINT is done by the ***Business Object Layer*** via ***objCustomerList*** Collection Class Object	1. Calls **objCustomerList.PrintAllEmployee()** method to do the work	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
LIST ALL: ▪ LIST of all ***CUSTOMER*** records ON THE FORM to a ***ListBox*** Control as a comma-delimited string. ▪ The LISTING is done by the FORM getting the data from ***Business Object Layer*** via ***objCustomerList*** Collection Class Object	1. Clears the ***ListBox*** Control. 2. Loops through the COLLECTION in the **objCustomerList)** OBJECT 3. During each LOOP, extract all the PROPERTIES of the current OBJECT in the COLLECTION. 4. Creates a comma-delimited string from ALL THE PROPERTEIS. 5. ADD & displays the string on the ***ListBox***. 6. Repeats the process for ALL OBJECTS in the COLLECTION.	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **ListBox Control** – to store and display as a ***comma-delimited*** line the content of each of the OBJECTS in the COLLECTION. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ If you feel you can use a ***DataGridView*** Control instead of ***ListBox***, you can do so. ▪ Future implementation will use the ***DataGridView***.
Exiting or Closing Form (SAVING THE DATA): ▪ As form is closing, the COLLECTION CLASS OBJECT **objCustomerList** should be SAVED to permanent storage or DATABASE, in this case the ***CustomerData.txt*** file. ▪ The SAVING is done by the ***Business Object Layer*** via ***objCustomerList*** Collection Class Object	1. Call the **objCustomerList .Saves()** method to do the work of Saving the content of the COLLECTION to database, in this case the ***CustomerData.txt*** File.	▪ Done from ***Form_Load Event-Handler***.	
Exiting or Closing Form (CLEARING THE COLLECTION): ▪ As form is closing, the COLLECTION CLASS OBJECT **objCustomerList** should be CLEARD of all OBJECTS, so is ready to be loaded the next time around. ▪ The CLEARING is done by the ***Business Object Layer*** via ***objCustomerList*** Collection Class Object	1. Call the **objCustomerList .Clear()** method to remove all objects from the COLLECTION.	▪ Done from ***Form_Load Event-Handler***.	

DVD Management Screen:

- The screen to manage the account data via the ***objDVDList*** *object* with the following requirements:

  - The ***DVD Management screens*** should allow for backend database support, where employees can manage the individual *DVD* records. The backend features include adding, removing, editing, listing DVD records etc. The backend screen should contain the necessary UI controls & code to implement the management options.

GENERAL FORM Information	Description
▪ **Management Form** ▪ **Created from Form Class** ▪ **You will need to create an OBJECT of this FORM & call the OBJECT.`ShowDialog()` method to display the Form as a Modal dialog box.**	▪ Use Windows Form to implement. ▪ Choose any of the desired Form, standard, MDI, etc., but keep functionality shown below. ▪ Choose your own colors etc. ▪ You are the visual designer. ▪ Some functionality will require special CONTROLS. See table below for description and requirements.

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
Loading or Displaying Form: ▪ As form is displaying, the COLLECTION CLASS OBJECT **objDVDList** should be POPULATED with RECORDS from permanent storage or DATABASE, in this case the ***DVDData.txt*** file. ▪ The LOADING is done by the ***Business Object Layer*** via ***objDVDList*** Collection Class Object	1. Call the **objDVDList.Load()** method to load the data from the ***DVDData.txt*** file.	▪ Done from ***Form_Load Event-Handler***.	
SEARCH: ▪ Search the database for the ***DVD*** record who's KEY or ***IDNumber*** is entered on the Form Textbox control. ▪ The SEARCH is done by the ***Business Object Layer*** via ***objDVDList*** Collection Class Object	**1.** Grabs the ***IDNumber*** from the Form's ***TextBox***. 2. **Calls objDVDList .Item(SSNUMBER)** property to retrieve a POINTER to object stored in Collection. 3. Displays content of OBJECT'S PROPERTIES being pointed by POINTER returned from Search on ***TextBoxes*** in the Form. 4. If object is NOT FOUND, displays a message indicating "**DVD not found**"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***IDNumber*** for SEARCH. ▪ **TextBoxes Controls** – to display the properties of the Object pointed by the POINTER returned from ***SEARCH()*** call. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	

Functionality	Action Taken	Graphical Control/Event-Handler	Graphical Control/Event-Handler
ADD: ▪ ADDS the NEW RECORD entered & displayed on the FORM. ▪ Selecting the ADD button will trigger process and ADD the OBJECT in the COLLECTION based on its KEY or ***IDNumber***. ▪ The ADD is done by the ***Business Object Layer*** via ***objDVDList*** Collection Class Object. ▪ In the ***DVDList*** Class, there are two ADD methods to choose from. ▪ **Select the appropriate *ADD()* method.**	1. Grabs the ***all values*** from the Form's ***TextBoxes*** that contain the record**.** 2. Calls **objDVDList .Add()** method to do the work.	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBoxes Controls** – to accept data that make up the RECORD to be ADD to COLLECTION. ▪ **(*MANDATORY CONTROL*) One COMBOBOX Control** – to display & allow user to select the various **Movie Category** options: - ***Action_Adventure*** - ***Drama,*** - ***Family_Kids*** - ***Horror*** - ***Sci-Fi_Fantasy*** - ***Music*** - ***Sports*** - ***Romance*** - ***Western*** - ***Comedy*** - ***None*** ▪ **Note:** The functionality of the **COMBOBOX** is not only to allow users to select the **CATEGORY** options shown, but to RETURN its equivalent as the appropriate ***Enumerated Data Type* enumMovieCategory** options: `MovieCategory`***.Action_Adventure*** `MovieCategory`***.Drama*** `MovieCategory`***.Family_Kids*** `MovieCategory`***.Horror*** `MovieCategory`***.Sci-Fi_Fantasy*** `MovieCategory`***.Music*** `MovieCategory`***.Sports*** `MovieCategory`***.Romance*** `MovieCategory`***.Western*** `MovieCategory`***.Comedy*** `MovieCategory`***.None*** ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ **(*MANDATORY CONTROL*) One COMBOBOX Control** – to display & allow user to select the various **DVD Format** options: - ***DVD*** - ***HD-DVD,*** - ***BLU-RAY DISC*** - ***None*** ▪ **Note:** The functionality of the **COMBOBOX** is not only to allow users to select the **FORMAT** options shown, but to RETURN its equivalent as the appropriate ***Enumerated Data Type* enumDVDFormat** options: `DVDFormat`***.DVD*** `DVDFormat`***.HD-DVD*** `DVDFormat`***.BLU-RAY DISC*** `DVDFormat`***.None***

Functionality	Action Taken	Graphical Control/Event-Handler	Graphical Control/Event-Handler
EDIT: ▪ EDITS the current RECORD being displayed on the FORM. ▪ Selecting the EDIT button will trigger process and modify the OBJECT in the COLLECTION based on its KEY or ***IDNumber***. ▪ The EDIT is done by the ***Business Object Layer*** via ***objDVDList*** Collection Class Object. ▪ In the ***DVDList*** Class, there are two EDIT methods to choose from. ▪ **Select the appropriate *EDIT()* method.**	1. **Grabs the *IDNumber* from the Form's** *TextBox*. 2. Calls **objDVDList.Edit()** method in module to do the work 3. If object is NOT FOUND, displays a message indicating "**DVD not found**"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBoxes Controls** – to accept data that make up the RECORD to be EDITED in the COLLECTION. ▪ **(*MANDATORY CONTROL*) One COMBOBOX Control** – to display & allow user to select the various **Movie Category** options: - ***Action_Adventure*** - ***Drama,*** - ***Family_Kids*** - ***Horror*** - ***Sci-Fi_Fantasy*** - ***Music*** - ***Sports*** - ***Romance*** - ***Western*** - ***Comedy*** - ***None*** ▪ **Note:** The functionality of the **COMBOBOX** is not only to allow users to select the **CATEGORY** options shown, but to RETURN its equivalent as the appropriate ***Enumerated Data Type*** **enumMovieCategory** options: `MovieCategory`***.Action_Adventure*** `MovieCategory`***.Drama*** `MovieCategory`***.Family_Kids*** `MovieCategory`***.Horror*** `MovieCategory`***.Sci-Fi_Fantasy*** `MovieCategory`***.Music*** `MovieCategory`***.Sports*** `MovieCategory`***.Romance*** `MovieCategory`***.Western*** `MovieCategory`***.Comedy*** `MovieCategory`***.None*** ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design	▪ **(*MANDATORY CONTROL*) One COMBOBOX Control** – to display & allow user to select the various **DVD Format** options: - ***DVD*** - ***HD-DVD,*** - ***BLU-RAY DISC*** - ***None*** ▪ **Note:** The functionality of the **COMBOBOX** is not only to allow users to select the **FORMAT** options shown, but to RETURN its equivalent as the appropriate ***Enumerated Data Type*** **enumDVDFormat** options: `DVDFormat`***.DVD*** `DVDFormat`***.HD-DVD*** `DVDFormat`***.BLU-RAY DISC*** `DVDFormat`***.None***

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
REMOVE: ▪ DELETES the Object in the COLLECTION, who's KEY or ***IDNumber*** is entered on the Form Textbox control. ▪ The DELETE is done by the ***Business Object Layer*** via ***objDVDList*** Collection Class Object	1. **Grabs the *IDNumber* from the Form's** *TextBox*. 2. Calls **objDVDList .Remove()** method to do the work 3. If object is NOT FOUND, displays a message indicating "**DVD not found**"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***IDNumber*** of OBJECT to be deleted. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪
PRINT: ▪ PRINTS the Object in the COLLECTION, who's KEY or ***IDNumber*** is entered on the Form Textbox control to the ***Network_Printer.txt*** file. ▪ The PRINT is done by the ***Business Object Layer*** via ***objDVDList*** Collection Class Object	1. **Grabs the *IDNumber* from the Form's** *TextBox*. 2. Calls **objDVDList.Print()** method to do the work 3. If object is NOT FOUND, displays a message indicating "**DVD not found**"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***IDNumber*** of OBJECT to be deleted. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪
PRINT ALL: ▪ PRINTS ALL Objects in the COLLECTION to the ***Network_Printer.txt*** file. ▪ The PRINT is done by the ***Business Object Layer*** via ***objDVDList*** Collection Class Object	1. Calls **objDVDList.PrintAllEmployee()** method to do the work	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
LIST ALL: ▪ LIST of all ***DVD*** records ON THE FORM to a ***ListBox*** Control as a comma-delimited string. ▪ The LISTING is done by the FORM getting the data from ***Business Object Layer*** via ***objDVDList*** Collection Class Object	1. Clears the ***ListBox*** Control. 2. Loops through the COLLECTION in the **objDVDList)** OBJECT 3. During each LOOP, extract all the PROPERTIES of the current OBJECT in the COLLECTION. 4. Creates a comma-delimited string from ALL THE PROPERTEIS. 5. ADD & displays the string on the ***ListBox***. 6. Repeats the process for ALL OBJECTS in the COLLECTION.	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **LISTBOX Control** – to store and display as a ***comma-delimited*** line the content of each of the OBJECTS in the COLLECTION. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ If you feel you can use a ***DataGridView*** Control instead of ***ListBox***, you can do so. ▪ Future implementation will use the ***DataGridView***.
Exiting or Closing Form (SAVING THE DATA): ▪ As form is closing, the COLLECTION CLASS OBJECT **objDVDList** should be SAVED to permanent storage or DATABASE, in this case the ***DVDData.txt*** file. ▪ The SAVING is done by the ***Business Object Layer*** via ***objDVDList*** Collection Class Object	1. Call the **objDVDList .Saves()** method to do the work of Saving the content of the COLLECTION to database, in this case the ***DVDData.txt*** File.	▪ Done from ***Form_Load Event-Handler***.	
Exiting or Closing Form (CLEARING THE COLLECTION): ▪ As form is closing, the COLLECTION CLASS OBJECT **objDVDList** should be CLEARD of all OBJECTS, so is ready to be loaded the next time around. ▪ The CLEARING is done by the ***Business Object Layer*** via ***objDVDList*** Collection Class Object	1. Call the **objDVDList .Clear()** method to remove all objects from the COLLECTION.	▪ Done from ***Form_Load Event-Handler***.	

VideoGame Management Screen:

- The screen to manage the videoGame data via the ***objVideoGameList*** *object* with the following requirements:

  - The ***Video Game Management screens*** should allow for backend database support, where employees can manage the individual ***VideoGame*** *Account* records. The backend features include adding, removing, editing, listing ***VideoGame*** account records etc. The backend screen should contain the necessary UI controls & code to implement the management options.

GENERAL FORM Information	Description
▪ **Management Form** ▪ **Created from Form Class** ▪ **You will need to create an OBJECT of this FORM & call the OBJECT.`ShowDialog()` method to display the Form as a Modal dialog box.**	▪ Use Windows Form to implement. ▪ Choose any of the desired Form, standard, MDI, etc., but keep functionality shown below. ▪ Choose your own colors etc. ▪ You are the visual designer. ▪ Some functionality will require special CONTROLS. See table below for description and requirements.

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
Loading or Displaying Form: ▪ As form is displaying, the COLLECTION CLASS OBJECT **objVideoGameList** should be POPULATED with RECORDS from permanent storage or DATABASE, in this case the ***VideoGameData.txt*** file. ▪ The LOADING is done by the ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object	1. Call the **objVideoGameList.Load()** method to load the data from the ***VideoGameData.txt*** file.	▪ Done from ***Form_Load Event-Handler***.	
SEARCH: ▪ Search the database for the ***VIDEOGAME*** record who's KEY or ***IDNumber*** is entered on the Form Textbox control. ▪ The SEARCH is done by the ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object	**1.** Grabs the ***IDNumber*** from the Form's ***TextBox***. 2. **Calls objVideoGameList .Item(SSNUMBER)** property to retrieve a POINTER to object stored in Collection. 3. Displays content of OBJECT'S PROPERTIES being pointed by POINTER returned from Search on ***TextBoxes*** in the Form. 4. If object is NOT FOUND, displays a message indicating "**VideoGame not found**"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***IDNumber*** for SEARCH. ▪ **TextBoxes Controls** – to display the properties of the Object pointed by the POINTER returned from ***SEARCH()*** call. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	

Functionality	Action Taken	Graphical Control/Event-Handler	Graphical Control/Event-Handler
ADD: ▪ ADDS the NEW RECORD entered & displayed on the FORM. ▪ Selecting the ADD button will trigger process and ADD the OBJECT in the COLLECTION based on its KEY or ***IDNumber***. ▪ The ADD is done by the ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object. ▪ In the ***VideoGameList*** Class, there are two ADD methods to choose from. ▪ **Select the appropriate *ADD()* method.**	**1.** Grabs the ***all values*** from the Form's ***TextBoxes*** that contain the record**.** 2. Calls **objVideoGameList .Add()** method to do the work.	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBoxes Controls** – to accept data that make up the RECORD to be ADD to COLLECTION. ▪ **(*MANDATORY CONTROL*) One COMBOBOX Control** – to display & allow user to select the various **VideoGame Category** options: - ***Action*** - ***Role-playing,*** - ***Shooting*** - ***Fighting*** - ***Racing*** - ***Sports*** - ***Strategy*** - ***Horror*** - ***Flight Simulators*** - ***Online*** - ***Rhythm*** - ***None*** ▪ **Note:** The functionality of the **COMBOBOX** is not only to allow users to select the **CATEGORY** options shown, but to RETURN its equivalent as the appropriate ***Enumerated Data Type* enumVideoGameCategory** options: `VideoGameCategory`***.Action*** `VideoGameCategory`***.Role-playing*** `VideoGameCategory`***.Shooting*** `VideoGameCategory`***.Fighting*** `VideoGameCategory`***.Racing*** `VideoGameCategory`***.Sports*** `VideoGameCategory`***.Strategy*** `VideoGameCategory`***.Horror*** `VideoGameCategory`***.Flight Simulators*** `VideoGameCategory`***.Online*** `VideoGameCategory`***.Rhythm*** `VideoGameCategory`***.None*** ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ **(*MANDATORY CONTROL*) One COMBOBOX Control** – to display & allow user to select the various **VideoGame Format** options: - ***X-Box*** - ***X-Box 360*** - ***PS3*** - ***PS2*** - ***GameCube*** - ***DS*** - ***Wii*** - ***PC*** - ***None*** ▪ **Note:** The functionality of the **COMBOBOX** is not only to allow users to select the **FORMAT** options shown, but to RETURN its equivalent as the appropriate ***Enumerated Data Type* enumVideoGameFormat** options: `VideoGameFormat`***.X-Box*** `VideoGameFormat`***.X-Box 360*** `VideoGameFormat`***.PS3*** `VideoGameFormat`***.PS2*** `VideoGameFormat`***.GameCube*** `VideoGameFormat`***.DS*** `VideoGameFormat`***.Wii*** `VideoGameFormat`***.PC*** `VideoGameFormat`***.None***

Functionality	Action Taken	Graphical Control/Event-Handler	Graphical Control/Event-Handler
EDIT: ▪ EDITS the current RECORD being displayed on the FORM. ▪ Selecting the EDIT button will trigger process and modify the OBJECT in the COLLECTION based on its KEY or ***IDNumber***. ▪ The EDIT is done by the ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object. ▪ In the ***VideoGameList*** Class, there are two EDIT methods to choose from. ▪ **Select the appropriate *EDIT()* method.**	1. **Grabs the *IDNumber* from the Form's** *TextBox*. 2. Calls **objVideoGameList.Edit()** method in module to do the work 3. If object is NOT FOUND, displays a message indicating "**VideoGame not found**"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBoxes Controls** – to accept data that make up the RECORD to be EDITED in the COLLECTION. ▪ **(*MANDATORY CONTROL*) One COMBOBOX Control** – to display & allow user to select the various **VideoGame Category** options: - ***Action*** - ***Role-playing,*** - ***Shooting*** - ***Fighting*** - ***Racing*** - ***Sports*** - ***Strategy*** - ***Horror*** - ***Flight Simulators*** - ***Online*** - ***Rhythm*** - ***None*** ▪ **Note:** The functionality of the **COMBOBOX** is not only to allow users to select the **CATEGORY** options shown, but to RETURN its equivalent as the appropriate ***Enumerated Data Type* enumVideoGameCategory** options: `VideoGameCategory`***.Action*** `VideoGameCategory`***.Role-playing*** `VideoGameCategory`***.Shooting*** `VideoGameCategory`***.Fighting*** `VideoGameCategory`***.Racing*** `VideoGameCategory`***.Sports*** `VideoGameCategory`***.Strategy*** `VideoGameCategory`***.Horror*** `VideoGameCategory`***.Flight Simulators*** `VideoGameCategory`***.Online*** `VideoGameCategory`***.Rhythm*** `VideoGameCategory`***.None*** ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ **(*MANDATORY CONTROL*) One COMBOBOX Control** – to display & allow user to select the various **VideoGame Format** options: - ***X-Box*** - ***X-Box 360*** - ***PS3*** - ***PS2*** - ***GameCube*** - ***DS*** - ***Wii*** - ***PC*** - ***None*** ▪ **Note:** The functionality of the **COMBOBOX** is not only to allow users to select the **FORMAT** options shown, but to RETURN its equivalent as the appropriate ***Enumerated Data Type* enumVideoGameFormat** options: `VideoGameFormat`***.X-Box*** `VideoGameFormat`***.X-Box 360*** `VideoGameFormat`***.PS3*** `VideoGameFormat`***.PS2*** `VideoGameFormat`***.GameCube*** `VideoGameFormat`***.DS*** `VideoGameFormat`***.Wii*** `VideoGameFormat`***.PC*** `VideoGameFormat`***.None***

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
REMOVE: ▪ DELETES the Object in the COLLECTION, who's KEY or ***IDNumber*** is entered on the Form Textbox control. ▪ The DELETE is done by the ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object	1. **Grabs the *IDNumber* from the Form's** *TextBox*. 2. Calls **objVideoGameList .Remove()** method to do the work 3. If object is NOT FOUND, displays a message indicating "**VideoGame not found**"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***IDNumber*** of OBJECT to be deleted. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪
PRINT: ▪ PRINTS the Object in the COLLECTION, who's KEY or ***IDNumber*** is entered on the Form Textbox control to the ***Network_Printer.txt*** file. ▪ The PRINT is done by the ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object	1. **Grabs the *IDNumber* from the Form's** *TextBox*. 2. Calls **objVideoGameList.Print()** method to do the work 3. If object is NOT FOUND, displays a message indicating "**VideoGame not found**"	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **TextBox Control** – to accept ***IDNumber*** of OBJECT to be deleted. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪
PRINT ALL: ▪ PRINTS ALL Objects in the COLLECTION to the ***Network_Printer.txt*** file. ▪ The PRINT is done by the ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object	1. Calls **objVideoGameList.PrintAllEmployee()** method to do the work	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪

Functionality	Action Taken	Graphical Control/Event-Handler	Description /Comments
LIST ALL: ▪ LIST of all ***VIDEOGAME*** records ON THE FORM to a ***ListBox*** Control as a comma-delimited string. ▪ The LISTING is done by the FORM getting the data from ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object	1. Clears the ***ListBox*** Control. 2. Loops through the COLLECTION in the **objVideoGameList)** OBJECT 3. During each LOOP, extract all the PROPERTIES of the current OBJECT in the COLLECTION. 4. Creates a comma-delimited string from ALL THE PROPERTEIS. 5. ADD & displays the string on the ***ListBox***. 6. Repeats the process for ALL OBJECTS in the COLLECTION.	▪ **Button Control** – to respond to user Click & ***Event-Handler*** to execute code. ▪ **LISTBOX Control** – to store and display as a ***comma-delimited*** line the content of each of the OBJECTS in the COLLECTION. ▪ **Labels**. – To label and describe all controls. ▪ **Others** – You feel are necessary for your design.	▪ If you feel you can use a ***DataGridView*** Control instead of ***ListBox***, you can do so. ▪ Future implementation will use the ***DataGridView***.
Exiting or Closing Form (SAVING THE DATA): ▪ As form is closing, the COLLECTION CLASS OBJECT **objVideoGameList** should be SAVED to permanent storage or DATABASE, in this case the ***VideoGameData.txt*** file. ▪ The SAVING is done by the ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object	1. Call the **objVideoGameList .Saves()** method to do the work of Saving the content of the COLLECTION to database, in this case the ***VideoGameData.txt*** File.	▪ Done from ***Form_Load Event-Handler***.	
Exiting or Closing Form (CLEARING THE COLLECTION): ▪ As form is closing, the COLLECTION CLASS OBJECT **objVideoGameList** should be CLEARD of all OBJECTS, so is ready to be loaded the next time around. ▪ The CLEARING is done by the ***Business Object Layer*** via ***objVideoGameList*** Collection Class Object	2. Call the **objVideoGameList .Clear()** method to remove all objects from the COLLECTION.	▪ Done from ***Form_Load Event-Handler***.	