LIBRARY MANGMENT SYSTEM

Muhammed Sherief ,231027188

Karim Hatem,231027652

Karim ahmed,231027807

Mahmoud Ahmed,231007714

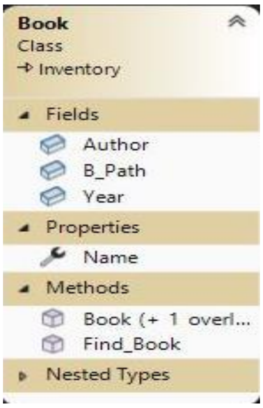
Saeed Muhammed,231017671

Table of contents

Project description

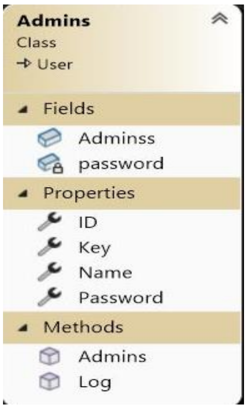
The Library Management System is a software application designed to automate and streamline the daily operations of a library. It handles user registration, cataloguing and tracking of books and DVDs, issuing and returning items, calculating fines, and managing purchase transactions. The system supports multiple user roles, such as administrators, librarians, and members, each with specific access and responsibilities. Data is stored using CSV files to ensure persistence and ease of access. Developed using object-oriented programming, the system applies concepts like encapsulation, inheritance, polymorphism, static members, and structured exception handling to ensure flexibility, maintainability, and error resilience. It also includes data collection and analysis features, measuring system performance and memory usage with and without exception handling. The project's goal is to reduce manual work, minimize human error, and provide efficient, user-friendly access to library resources including books and DVDs.

CLASS UML DIAGRAM



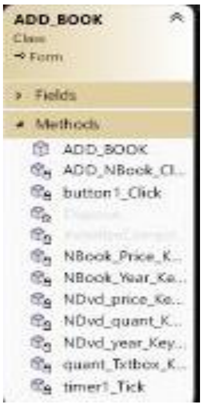
Book
Class
→ Inventory

- Fields
  - Author
  - B_Path
  - Year
- Properties
  - Name
- Methods
  - Book (+ 1 overl...
  - Find_Book
- Nested Types



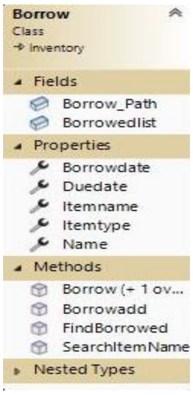
Admins
Class
→ User

- Fields
  - Adminss
  - password
- Properties
  - ID
  - Key
  - Name
  - Password
- Methods
  - Admins
  - Log



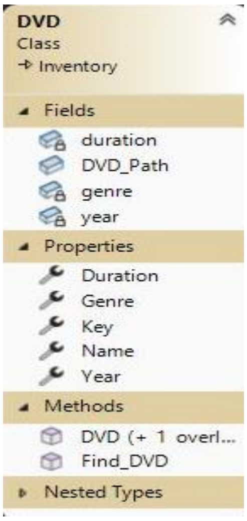
ADD_BOOK
Class
→ Form

- Fields
- Methods
  - ADD_BOOK
  - ADD_NBook_Cl...
  - button1_Click
  - Dispose
  - InitializeComponent
  - NBook_Price_K...
  - NBook_Year_Ke...
  - NDvd_price_Ke...
  - NDvd_quant_K...
  - NDvd_year_Key...
  - quant_Txtbox_K...
  - timer1_Tick



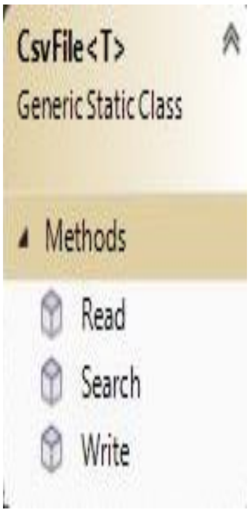
Borrow
Class
→ Inventory

- Fields
  - Borrow_Path
  - Borrowedlist
- Properties
  - Borrowdate
  - Duedate
  - Itemname
  - Itemtype
  - Name
- Methods
  - Borrow (+ 1 ov...
  - Borrowadd
  - FindBorrowed
  - SearchItemName
- Nested Types



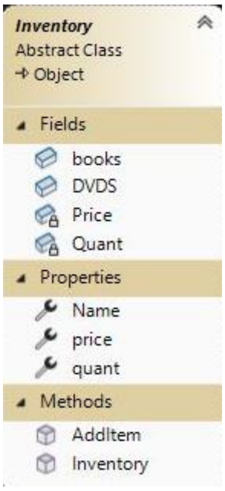
DVD
Class
→ Inventory

- Fields
  - duration
  - DVD_Path
  - genre
  - year
- Properties
  - Duration
  - Genre
  - Key
  - Name
  - Year
- Methods
  - DVD (+ 1 overl...
  - Find_DVD
- Nested Types



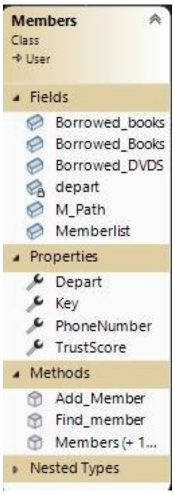
CsvFile<T>
Generic Static Class

- Methods
  - Read
  - Search
  - Write



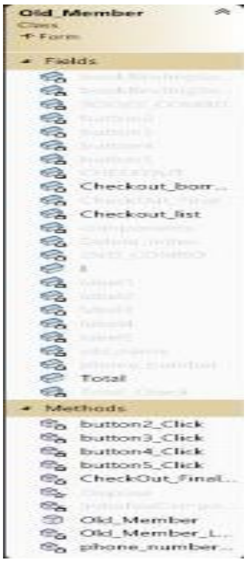
Inventory
Abstract Class
→ Object

- Fields
  - books
  - DVDS
  - Price
  - Quant
- Properties
  - Name
  - price
  - quant
- Methods
  - AddItem
  - Inventory



Members
Class
→ User

- Fields
  - Borrowed_books
  - Borrowed_Books
  - Borrowed_DVDS
  - depart
  - M_Path
  - Memberlist
- Properties
  - Depart
  - Key
  - PhoneNumber
  - TrustScore
- Methods
  - Add_Member
  - Find_member
  - Members (+ 1...
- Nested Types



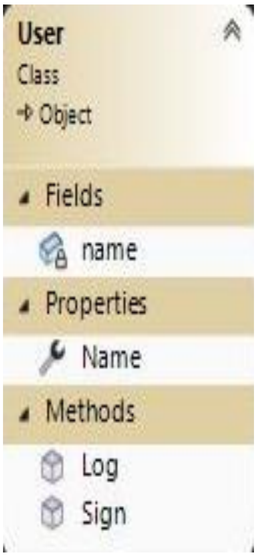
Old_Member
Class
Form

Fields

- [illegible]
- [illegible]
- BOOKS_COMBO
- button2
- button3
- button4
- button5
- CHECKOUT
- Checkout_borr...
- CheckOut_final...
- Checkout_list
- components
- Delete_mem...
- DVD_COMBO
- i
- label1
- label2
- label3
- label4
- label5
- old_name
- phone_number
- Total
- Total_Check

Methods

- button2_Click
- button3_Click
- button4_Click
- button5_Click
- CheckOut_Final...
- Dispose
- InitializeCompo...
- Old_Member
- Old_Member_L...
- phone_number...



User
Class
Object

Fields

- name

Properties

- Name

Methods

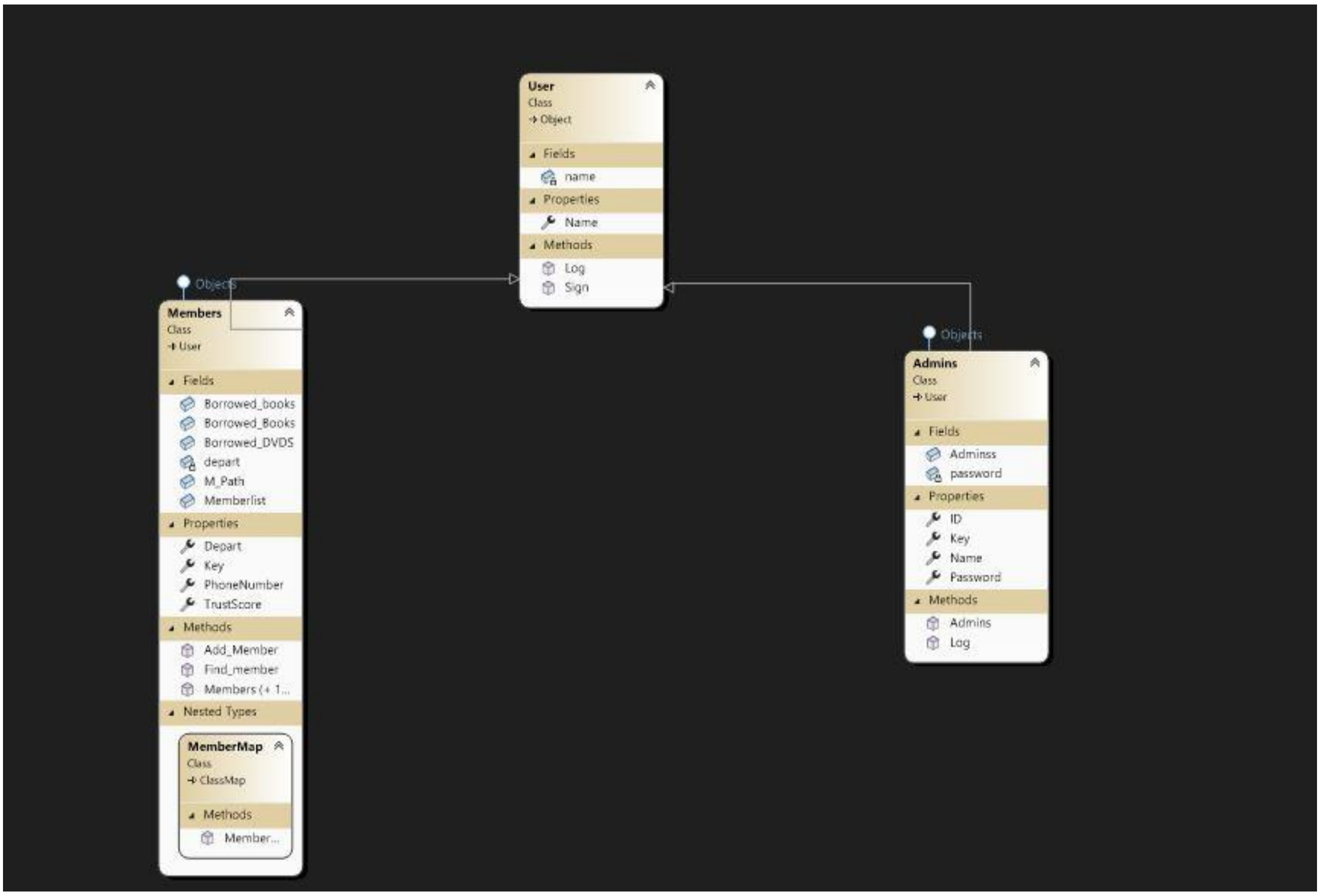
- Log
- Sign

UML DIAGRAM

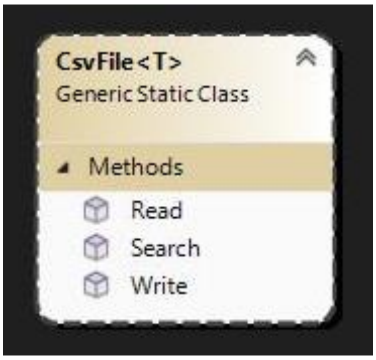
-User: Parent / Base Class

-Admins: Child / Derived Class

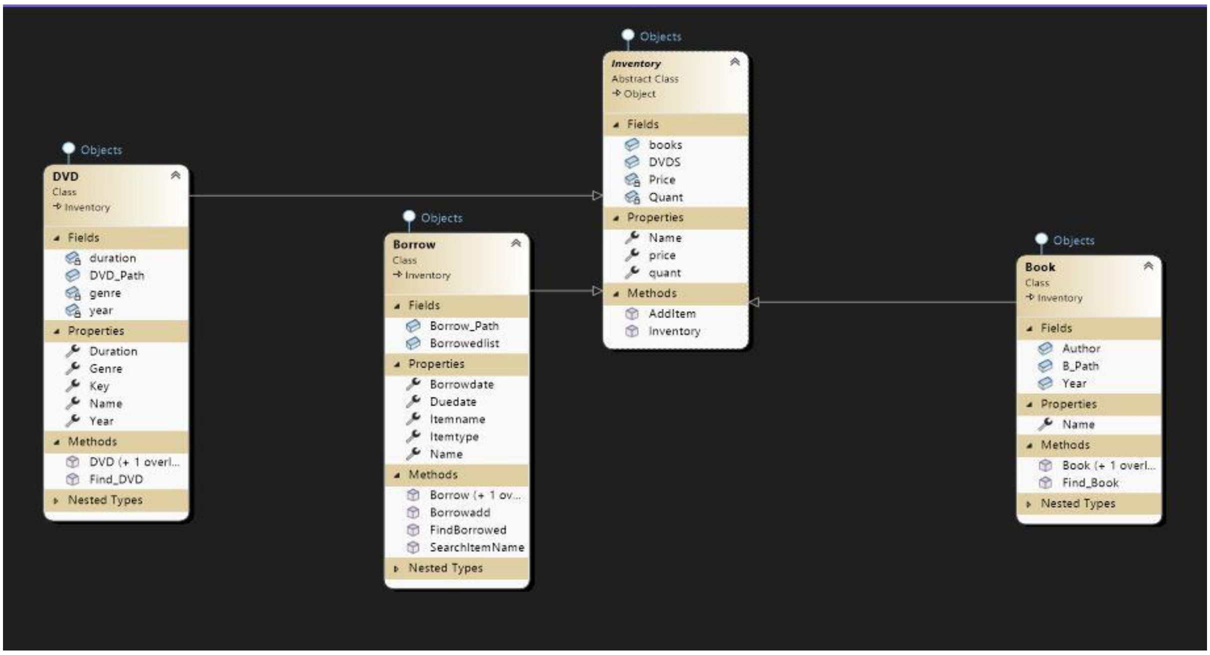
-Members: Child Derived Class



-*CSV: Generic (library Class)*



- **Inventory: Parent/Base Class**
- ***Book: Child / Derived Class***
- ***DVD: Child Derived Class***
- ***Borrow: Child / Derived Class***

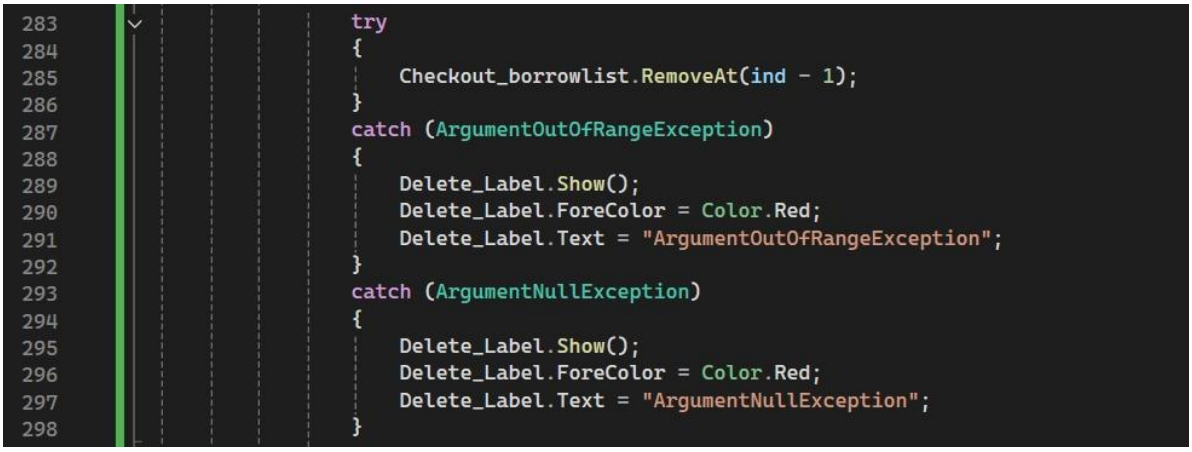


Analysis of the Exception Handling Techniques and Their Impact

3.1 Exception Handling

Exception handling allows you to manage runtime errors gracefully by using try, catch, and finally blocks. Code that might cause an error is placed inside the try block, while the catch block handles specific exceptions if they occur, preventing the program from crashing. The finally block contains code that always runs, whether an exception was thrown or not, typically used for cleanup tasks.

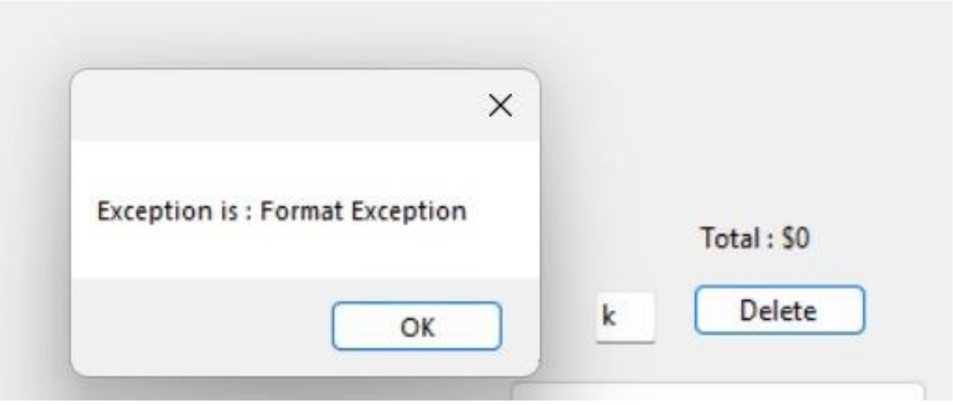
Fig 3.1.1



```
try
{
    Checkout_borrowlist.RemoveAt(ind - 1);
}
catch (ArgumentOutOfRangeException)
{
    Delete_Label.Show();
    Delete_Label.ForeColor = Color.Red;
    Delete_Label.Text = "ArgumentOutOfRangeException";
}
catch (ArgumentNullException)
{
    Delete_Label.Show();
    Delete_Label.ForeColor = Color.Red;
    Delete_Label.Text = "ArgumentNullException";
}
```

In fig 3.1.1 code attempts to remove an item from the Checkout_borrowList at the position ind - 1 and handles potential exceptions using a try-catch block. If an ArgumentOutOfRangeException or ArgumentNullException occurs, it displays a red error message on the Delete_Label indicating the specific type of exception, providing feedback to the user about the error.

Fig 3.1.2



Exception is : Format Exception

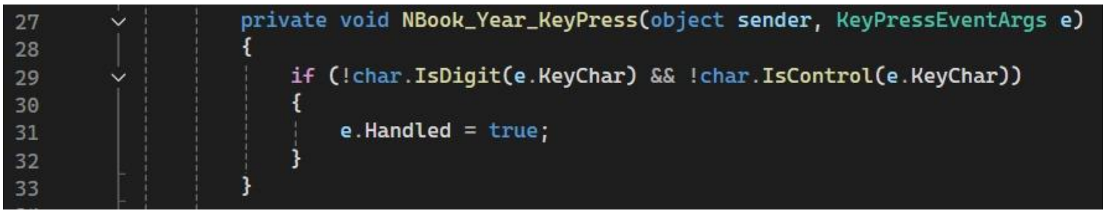
Total : $0

In fig 3.1.2 the type of exception is Format Exception as the user entered the wrong format.

3.2 validation

Validation is the process of checking input data to ensure it meets certain criteria before being processed or stored. This helps prevent errors and ensures data integrity by verifying things like required fields, data types, formats, or value ranges.

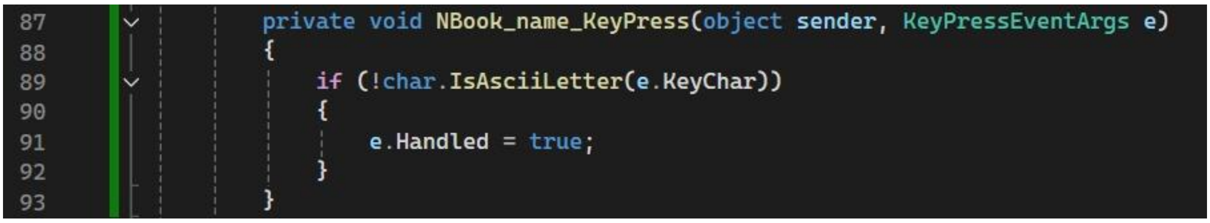
fig 3.2.1



```
private void NBook_Year_KeyPress(object sender, KeyPressEventArgs e)
{
    if (!char.IsDigit(e.KeyChar) && !char.IsControl(e.KeyChar))
    {
        e.Handled = true;
    }
}
```

In fig 3.2.1 event handler restricts user input in a text field to only digits and control characters. When a key is pressed, it checks if the character is not a digit and not a control key; if both conditions are true, it sets e.Handled = true, which cancels the key press, effectively blocking non-numeric input.

fig 3.2.2

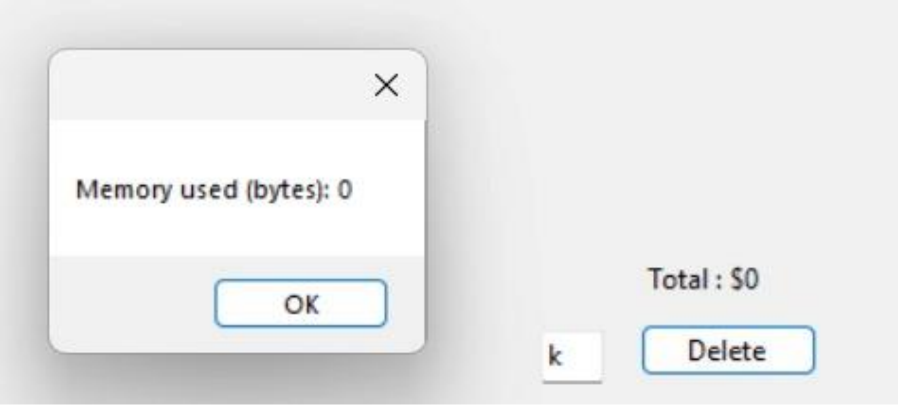


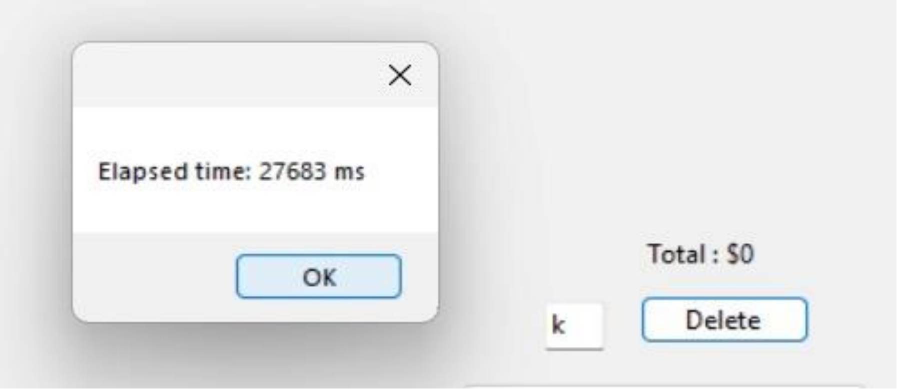
```
        private void NBook_name_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (!char.IsAsciiLetter(e.KeyChar))
            {
                e.Handled = true;
            }
        }
```

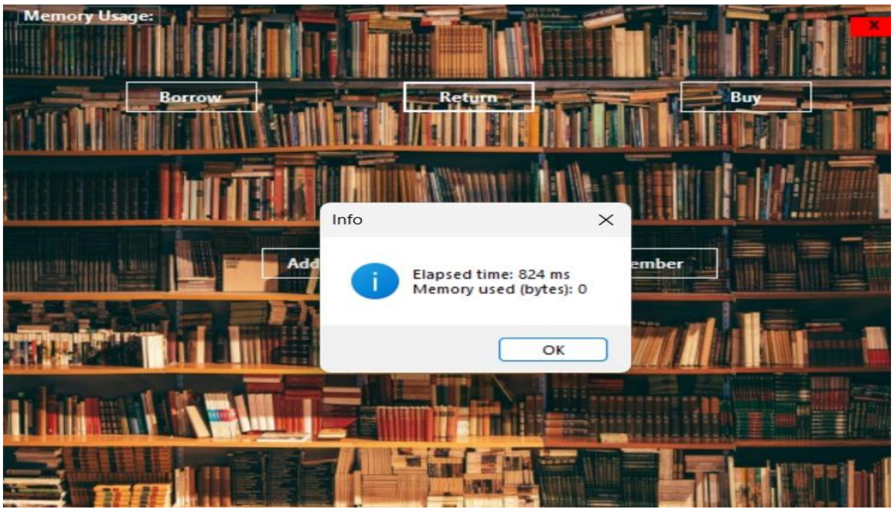
In fig 3.2.2 it checks each typed character in a text box and only allows letters. It blocks any other keys from being entered.

3.3 Elapsed time and memory usage

Elapsed time refers to the total duration a program or operation takes to complete, from start to finish, while memory usage indicates the amount of RAM consumed during its execution. Monitoring both metrics is essential for evaluating the performance and efficiency of an application, helping to identify potential bottlenecks or resource overuse.

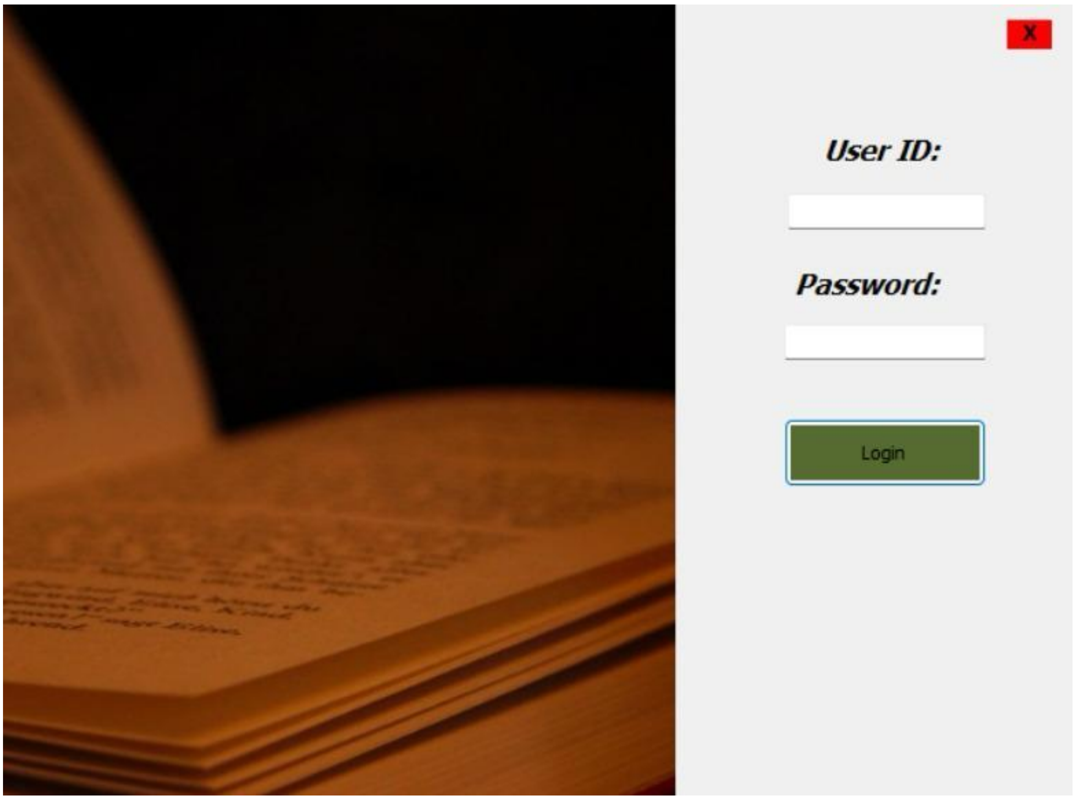

Memory used (bytes): 0

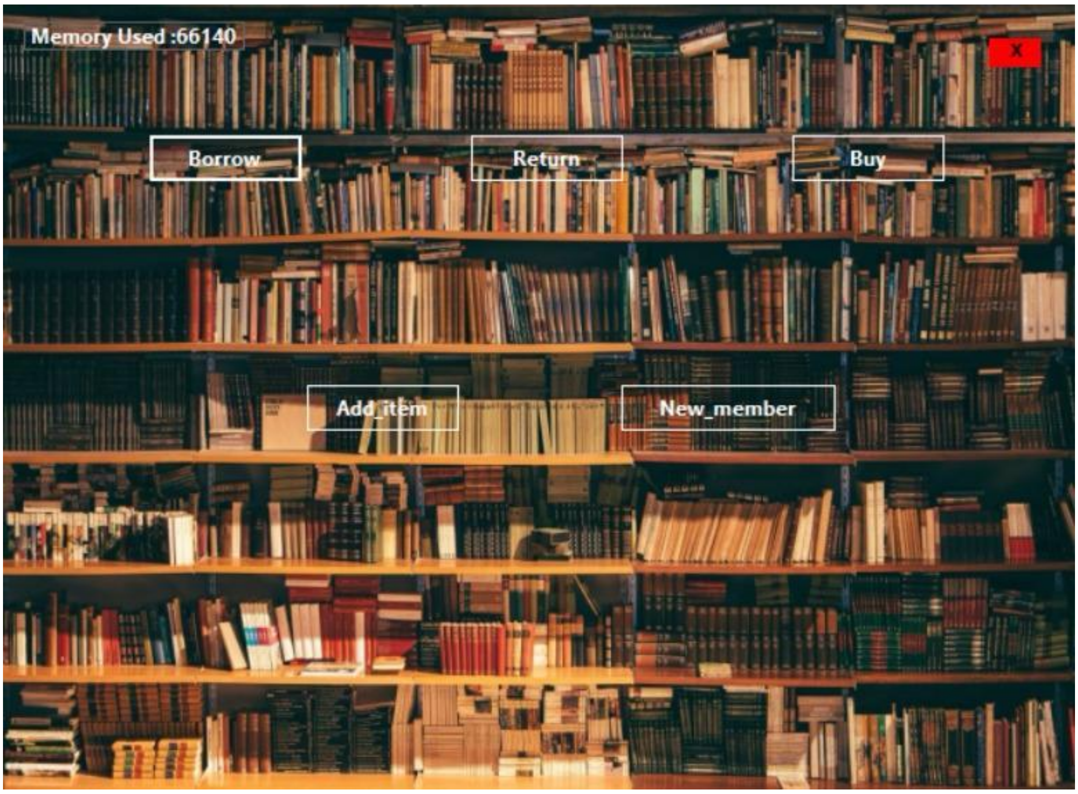

Elapsed time: 27683 ms

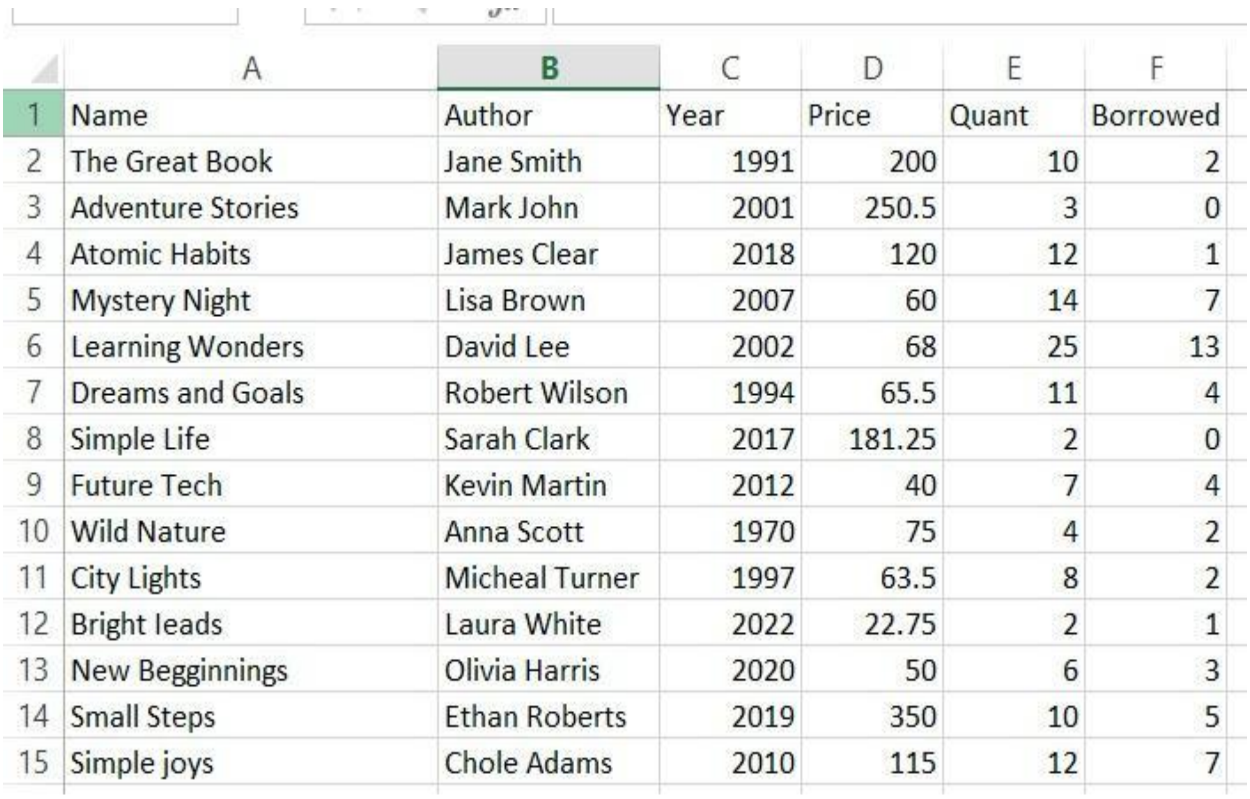

Info

Elapsed time: 824 ms
Memory used (bytes): 0

Test Results and Evaluation of System Performance

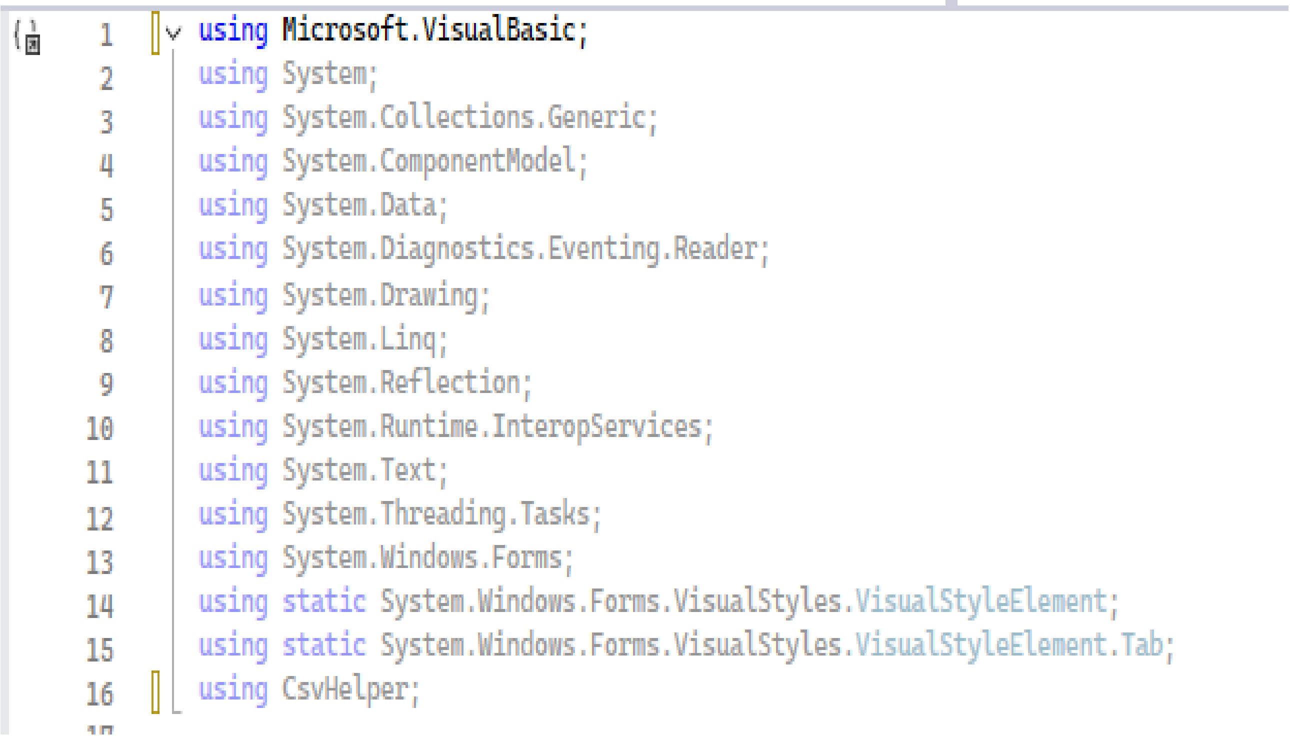






	A	B	C	D	E	F
1	Name	Author	Year	Price	Quant	Borrowed
2	The Great Book	Jane Smith	1991	200	10	2
3	Adventure Stories	Mark John	2001	250.5	3	0
4	Atomic Habits	James Clear	2018	120	12	1
5	Mystery Night	Lisa Brown	2007	60	14	7
6	Learning Wonders	David Lee	2002	68	25	13
7	Dreams and Goals	Robert Wilson	1994	65.5	11	4
8	Simple Life	Sarah Clark	2017	181.25	2	0
9	Future Tech	Kevin Martin	2012	40	7	4
10	Wild Nature	Anna Scott	1970	75	4	2
11	City Lights	Micheal Turner	1997	63.5	8	2
12	Bright leads	Laura White	2022	22.75	2	1
13	New Begginnings	Olivia Harris	2020	50	6	3
14	Small Steps	Ethan Roberts	2019	350	10	5
15	Simple joys	Chole Adams	2010	115	12	7

Documentation of Dependencies or External Libraries Used



```csharp
using Microsoft.VisualBasic;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics.Eventing.Reader;
using System.Drawing;
using System.Linq;
using System.Reflection;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using static System.Windows.Forms.VisualStyles.VisualStyleElement;
using static System.Windows.Forms.VisualStyles.VisualStyleElement.Tab;
using CsvHelper;
```

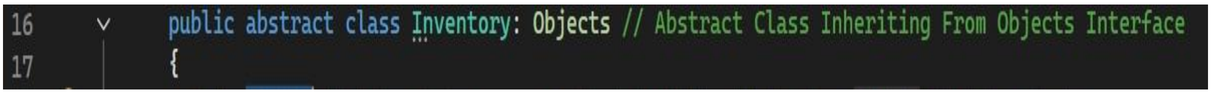
]

This file imports a wide range of namespaces for handling system functions, data processing, UI development, asynchronous tasks, and external libraries. It includes core .NET namespaces like System, System.Data, and System.Windows.Forms for basic functionality, UI, and data handling, along with Microsoft.VisualBasic for VB support and CsvHelper for working with CSV files. Static imports from VisualStyleElement allow direct access to specific UI elements, making the code more concise and organized.

Techniques Used

6.1 Inheritance

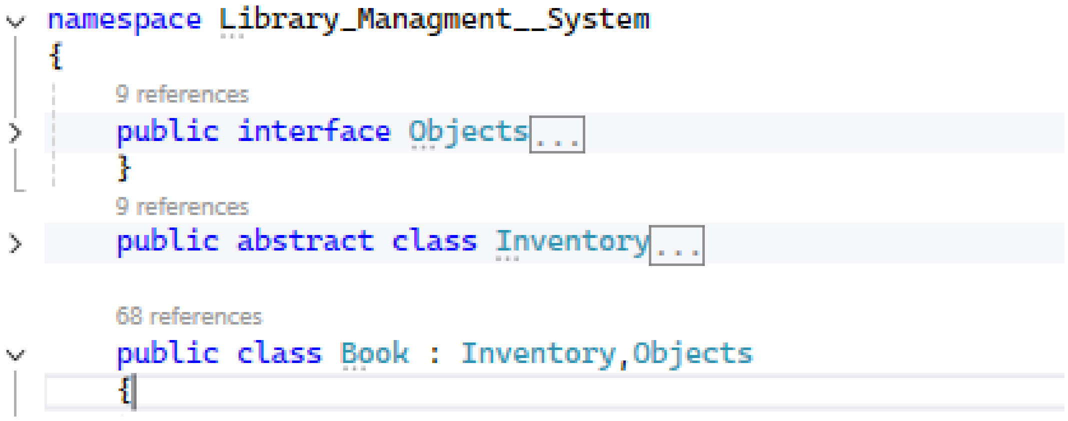
Inheritance enables a new class to reuse, extend, or modify the functionality defined in an existing class. The derived class inherits accessible members (fields, methods, properties) from the base class and can also introduce new members or override existing ones we used



```
public abstract class Inventory: Objects // Abstract Class Inheriting From Objects Interface
{
```

In C#, a class cannot inherit from more than one base class due to the complexities associated with multiple inheritance." However, C# allows a class to implement multiple interfaces, providing a way to achieve multiple inheritance of type definitions without the associated ambiguities. Here in fig 6.1.1 book extends functionality from Inventory (shared code) and commits to implement members defined by Objects.

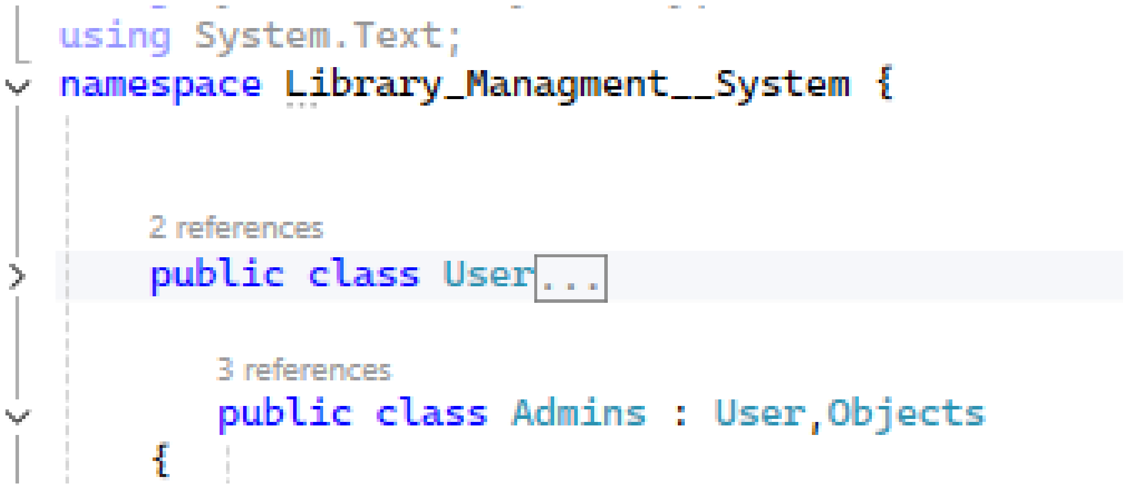
Fig 6.1.1



```
namespace Library_Managment__System
{
        9 references
        public interface Objects...
        }
        9 references
        public abstract class Inventory...

        68 references
        public class Book : Inventory,Objects
        {
```

In fig 6.1.2 public class Admins inherit from both user class and Objects.

Fig 6.1.2

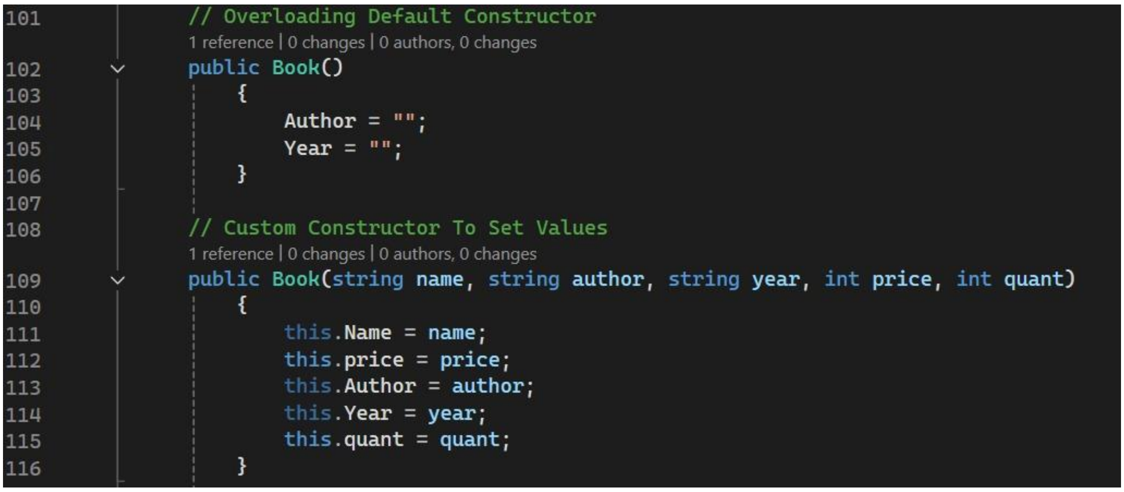


```
using System.Text;
namespace Library_Managment__System {


    2 references
    public class User...

        3 references
        public class Admins : User,Objects
    {
```

6.2 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class, enabling a single interface to represent different underlying data types. It is achieved through method overriding (runtime polymorphism) and method overloading (compile-time polymorphism), allowing the same method name to behave differently based on the object or the parameters used. This promotes flexibility, code reusability, and easier maintenance.

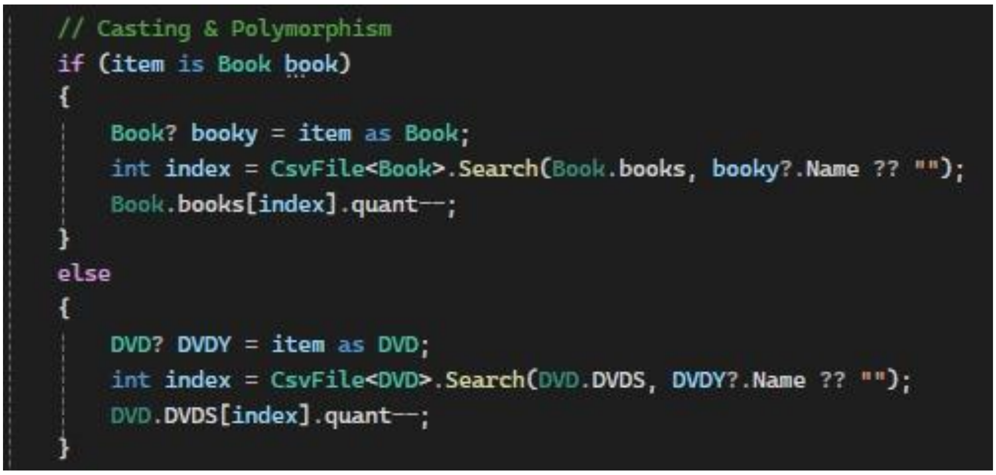
fig 6.2.1



```
        // Overloading Default Constructor
        1 reference | 0 changes | 0 authors, 0 changes
        public Book()
        {
            Author = "";
            Year = "";
        }

        // Custom Constructor To Set Values
        1 reference | 0 changes | 0 authors, 0 changes
        public Book(string name, string author, string year, int price, int quant)
        {
            this.Name = name;
            this.price = price;
            this.Author = author;
            this.Year = year;
            this.quant = quant;
        }
```

In fig 6.2.1 defines two constructors for a Book class: overloading a default one that initializes Author and Year as empty strings, and a parameterized one that sets all book properties (Name, Author, Year, Price, Quantity) when creating an instance.

fig 6.2.2



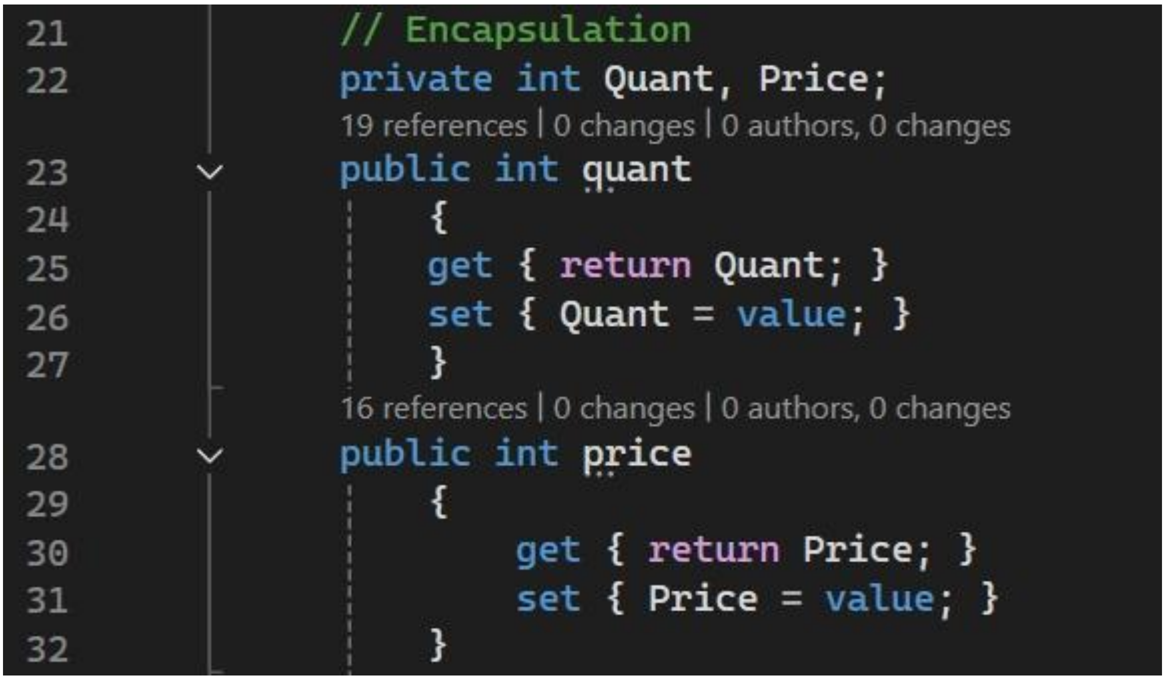
```
// Casting & Polymorphism
if (item is Book book)
{
    Book? booky = item as Book;
    int index = CsvFile<Book>.Search(Book.books, booky?.Name ?? "");
    Book.books[index].quant--;
}
else
{
    DVD? DVDY = item as DVD;
    int index = CsvFile<DVD>.Search(DVD.DVDS, DVDY?.Name ?? "");
    DVD.DVDS[index].quant--;
}
```

The code checks whether an item is a Book or DVD using type checking and casting. If the item is a Book, it locates the corresponding entry in the Book.books list by searching for a matching name, then reduces its quantity by one. Similarly, if the item is a DVD, it performs the same operation on the DVD.DVDS list. The search uses the item's name for matching, defaulting to an empty string if the name is null. This approach enables handling different inventory item types through a shared interface while maintaining separate collections for each type, with modifications applied directly to the static lists that store these items.

6.3 Encapsulation

Encapsulation involves bundling data (variables) and methods (functions) that operate on the data into a single unit, typically a class, while restricting direct access to some of the object's components. This is done by making variables private and exposing them through public methods or properties, which ensures better control, security, and maintainability of the code by hiding internal implementation details from outside interference.

fig 6.3.1



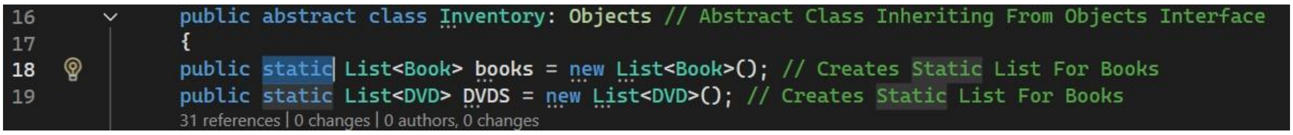
```
// Encapsulation
private int Quant, Price;
19 references | 0 changes | 0 authors, 0 changes
public int quant
    {
    get { return Quant; }
    set { Quant = value; }
    }
16 references | 0 changes | 0 authors, 0 changes
public int price
    {
        get { return Price; }
        set { Price = value; }
    }
```

In figure 6.3.1 code snippet demonstrates encapsulation by using private fields (Quant and Price) and exposing them through public properties (quant and price). The properties use get and set accessors to safely retrieve and assign values to the private fields, enabling controlled access to the data. This approach helps protect the internal state of an object and adheres to object-oriented programming principles.

6.4 Static Members

Static members are class-level variables or methods that belong to the class itself rather than to any specific object instance. This means they can be accessed without creating an object of the class. Static members are shared among all instances of the class, making them useful for defining values or behaviours that should be common across all objects.

fig 6.4.1



```
public abstract class Inventory: Objects // Abstract Class Inheriting From Objects Interface
{
public static List<Book> books = new List<Book>(); // Creates Static List For Books
public static List<DVD> DVDS = new List<DVD>(); // Creates Static List For Books
31 references | 0 changes | 0 authors, 0 changes
```

In fig 6.4.1 The static modifier ensures that the books and DVDs lists are associated with the Inventory class itself rather than individual instances. These lists exist as single, shared collections throughout the application's lifetime, allowing all parts of the program to access and modify the same set of book and DVD objects. Since they're static, these lists persist in memory from when the application starts until it terminates, providing a centralized way to manage inventory data.

Conclusion

the Library Management System successfully achieves its objective of enhancing library operations by automating core tasks such as cataloging, issuing, returning items, and managing users. By leveraging object-oriented programming principles and structured exception handling, the system ensures robustness, scalability, and maintainability. The use of CSV files for data storage offers a simple yet effective way to maintain persistent records. With support for multiple user roles and performance analysis features, the system not only improves efficiency and accuracy but also provides valuable insights into resource usage. Overall, it delivers a reliable and user-friendly solution for modern library management needs.

Special thanks to Dr. Manal Mostafa and Eng. Farah Labib for their invaluable support and guidance.