



GLA
UNIVERSITY
MATHURA
Recognised by UGC Under Section 2(f)

Accredited with **A⁺** Grade by **NAAC**
3.46 Score

12-B Status from UGC

(SESSION 2023-2024)

MCA –IIIrd SEMESTER

PRACTICAL FILE – .Net Framework Lab

SUBMITTED TO:

MR. Sachendra Singh Chauhan

(ASSISTANT PROFESSOR)

DEPT. OF C.E.A.

SUBMITTED BY:

NAME – Vinay Singh

SECTION – B ROLL NO. – 76

UNIV. ROLL NO. - 2284200236

DATE :- 16-SEP-2023

Lab Assignment :- 4

Ques:-1 Create a class representing a bank account with a balance property. Implement a property validation that prevents the balance from going negative.

Ans:-

```
using System;

public class BankAccount
{
    private double balance;

    // Balance property with validation
    public double Balance
    {
        get { return balance; }

        set
        {
            if (value < 0)
            {
                Console.WriteLine("Error: Balance cannot be negative.");
            }

            else
            {

```

```
        balance = value;
    }
}

// Constructor to initialize the account with an initial balance
public BankAccount(double initialBalance)
{
    Balance = initialBalance;
}

// Method to deposit money into the account
public void Deposit(double amount)
{
    if (amount > 0)
    {
        Balance += amount;

        Console.WriteLine($"Deposited ${amount}. New balance: ${Balance}");
    }
    else
    {
        Console.WriteLine("Error: Invalid deposit amount.");
    }
}
```

```
// Method to withdraw money from the account

public void Withdraw(double amount)
{
    if (amount > 0 && amount <= Balance)
    {
        Balance -= amount;

        Console.WriteLine($"Withdrawn ${amount}. New balance: ${Balance}");
    }
    else
    {
        Console.WriteLine("Error: Invalid withdrawal amount or insufficient funds.");
    }
}

// Method to display the account balance

public void DisplayBalance()
{
    Console.WriteLine($"Account balance: ${Balance}");
}

}

class Program
{
    static void Main()
    {
        // Create a new bank account with an initial balance of $1000
    }
}
```

```
BankAccount account = new BankAccount(1000);

// Display the initial balance
account.DisplayBalance();

// Deposit $500
account.Deposit(500);

// Withdraw $200
account.Withdraw(200);

// Try to withdraw $2000 (should display an error)
account.Withdraw(2000);

// Try to deposit a negative amount (should display an error)
account.Deposit(-100);

// Display the final balance
account.DisplayBalance();
}
}
```

Ques:-2 . Write a class representing a car with properties for make, model, and year. Implement a property that returns the full car name (e.g., "Toyota Camry 2022").

Ans:- public class Car

```
{  
  
    public string Make { get; set; }  
  
    public string Model { get; set; }  
  
    public int Year { get; set; }  
  
  
    // Property that returns the full car name  
  
    public string FullCarName  
  
    {  
  
        get  
  
        {  
  
            return $"{Make} {Model} {Year}";  
  
        }  
  
    }  
  
  
    // Constructor to initialize the car properties  
  
    public Car(string make, string model, int year)  
  
    {  
  
        Make = make;  
  
        Model = model;  
  
        Year = year;  
  
    }  
  
}  
  
class Program  
  
{  
  
    static void Main()
```

```
{  
    // Create a new car instance  
    Car myCar = new Car("Toyota", "Camry", 2022);  
  
    // Access and display the full car name  
    string carName = myCar.FullCarName;  
    Console.WriteLine("My car: " + carName); // Output: My car: Toyota Camry 2022  
}  
}
```

Ques:-3 Create a class representing a person with properties for first name and last name. Implement a property that returns the full name in uppercase.

Ans:-

```
public class Person  
{  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
  
    // Property that returns the full name in uppercase  
    public string FullNameUpperCase  
    {  
        get  
        {  
            return $"{FirstName} {LastName}".ToUpper();  
        }  
    }  
}
```

```

    }

    // Constructor to initialize the person's properties
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}

class Program
{
    static void Main()
    {
        // Create a new person instance
        Person person = new Person("John", "Doe");

        // Access and display the full name in uppercase
        string fullNameUpper = person.FullNameUpperCase;

        Console.WriteLine("Full Name in Uppercase: " + fullNameUpper); // Output: Full Name in
Uppercase: JOHN DOE
    }
}

```

Ques:-4 . Implement a property for a class representing a temperature in Celsius that converts the temperature to Fahrenheit when accessed.

Ans:- public class Temperature

```
{  
    private double celsius;  
  
    // Property for temperature in Celsius  
    public double Celsius  
    {  
        get { return celsius; }  
        set { celsius = value; }  
    }  
  
    // Property for temperature in Fahrenheit (computed property)  
    public double Fahrenheit  
    {  
        get  
        {  
            return (Celsius * 9 / 5) + 32;  
        }  
    }  
  
    // Constructor to initialize the temperature in Celsius  
    public Temperature(double celsius)  
    {  
        Celsius = celsius;  
    }  
}
```

```

}

class Program
{
    static void Main()
    {
        // Create a new Temperature instance with 25 degrees Celsius

        Temperature temp = new Temperature(25);


        // Access and display the temperature in Fahrenheit

        double fahrenheit = temp.Fahrenheit;

        Console.WriteLine($"Temperature in Fahrenheit: {fahrenheit}°F"); // Output: Temperature in
Fahrenheit: 77°F
    }
}

```

Ques:-5 Build a class representing a custom list and implement an indexer to access elements by index.

Ans:-

```

public class CustomList<T>
{
    private T[] items;

    private int capacity;

    private int count;


    // Constructor to initialize the custom list with a specified capacity

```

```
public CustomList(int capacity)
{
    if (capacity <= 0)
    {
        throw new ArgumentException("Capacity must be greater than zero.");
    }

    this.capacity = capacity;
    this.items = new T[capacity];
    this.count = 0;
}

// Indexer to access elements by index
public T this[int index]
{
    get
    {
        if (index < 0 || index >= count)
        {
            throw new IndexOutOfRangeException("Index is out of range.");
        }

        return items[index];
    }
    set
```

```
{  
    if (index < 0 || index >= count)  
    {  
        throw new IndexOutOfRangeException("Index is out of range.");  
    }  
  
    items[index] = value;  
}  
}
```

// Method to add an item to the custom list

```
public void Add(T item)  
{  
    if (count == capacity)  
    {  
        // If the list is full, double the capacity  
        capacity *= 2;  
        Array.Resize(ref items, capacity);  
    }  
  
    items[count] = item;  
    count++;  
}
```

// Method to get the number of elements in the custom list

```
public int Count
{
    get { return count; }
}

class Program
{
    static void Main()
    {
        // Create a custom list of integers with an initial capacity of 3
        CustomList<int> myList = new CustomList<int>(3);

        // Add elements to the list
        myList.Add(1);
        myList.Add(2);
        myList.Add(3);

        // Access elements by index using the indexer
        Console.WriteLine("Element at index 0: " + myList[0]); // Output: Element at index 0: 1
        Console.WriteLine("Element at index 1: " + myList[1]); // Output: Element at index 1: 2
        Console.WriteLine("Element at index 2: " + myList[2]); // Output: Element at index 2: 3

        // Modify an element using the indexer
        myList[1] = 99;

        Console.WriteLine("Modified element at index 1: " + myList[1]); // Output: Modified element at
index 1: 99
    }
}
```

```
}  
}
```

Ques:-6 . How can you use an indexer to create a simple stack data structure in C#?

Ans:-

```
using System;  
  
public class SimpleStack<T>  
{  
    private T[] items;  
    private int top; // Index of the top element  
  
    public SimpleStack(int capacity)  
    {  
        if (capacity <= 0)  
        {  
            throw new ArgumentException("Capacity must be greater than zero.");  
        }  
  
        items = new T[capacity];  
        top = -1; // Initialize top to -1 to represent an empty stack  
    }  
  
    // Indexer to access the element at the top of the stack  
    public T this[int index]
```

```
{  
    get  
    {  
        if (index < 0 || index > top)  
        {  
            throw new IndexOutOfRangeException("Index is out of range.");  
        }  
  
        return items[index];  
    }  
}
```

// Push an element onto the stack

```
public void Push(T item)  
{  
    if (top == items.Length - 1)  
    {  
        // Stack is full, resize the array  
        Array.Resize(ref items, items.Length * 2);  
    }  
  
    top++;  
    items[top] = item;  
}
```

```
// Pop an element from the stack
```

```
public T Pop()
```

```
{
```

```
    if (top == -1)
```

```
    {
```

```
        throw new InvalidOperationException("Stack is empty.");
```

```
    }
```

```
    T poppedItem = items[top];
```

```
    top--;
```

```
    return poppedItem;
```

```
}
```

```
// Check if the stack is empty
```

```
public bool IsEmpty
```

```
{
```

```
    get { return top == -1; }
```

```
}
```

```
// Get the number of elements in the stack
```

```
public int Count
```

```
{
```

```
    get { return top + 1; }
```

```
}
```



```
}  
  
class Program  
{  
    static void Main()  
    {  
        // Create a stack of integers with an initial capacity of 3  
        SimpleStack<int> stack = new SimpleStack<int>(3);  
  
        // Push elements onto the stack  
        stack.Push(1);  
        stack.Push(2);  
        stack.Push(3);  
  
        // Access elements by index using the indexer  
        Console.WriteLine("Element at the top of the stack: " + stack[stack.Count - 1]); // Output: Element  
        at the top of the stack: 3  
  
        // Pop elements from the stack  
        int poppedItem = stack.Pop();  
        Console.WriteLine("Popped item: " + poppedItem); // Output: Popped item: 3  
  
        // Check if the stack is empty  
        Console.WriteLine("Is stack empty? " + stack.IsEmpty); // Output: Is stack empty? False  
  
        // Get the number of elements in the stack
```

```
Console.WriteLine("Number of elements in the stack: " + stack.Count); // Output: Number of  
elements in the stack: 2
```

```
}  
}
```

Ques:-7 . Implement an indexer in a class representing a bookshelf that allows you to access books by title.

Ans:-

```
using System;  
  
using System.Collections.Generic;
```

```
public class Book  
{  
    public string Title { get; set; }  
    public string Author { get; set; }  
}
```

```
public class Bookshelf  
{  
    private List<Book> books;  
  
    public Bookshelf()  
    {  
        books = new List<Book>();  
    }
```

```
// Indexer to access books by title
```

```
public Book this[string title]
```

```
{
```

```
    get
```

```
    {
```

```
        return books.Find(book => book.Title == title);
```

```
    }
```

```
}
```

```
// Method to add a book to the bookshelf
```

```
public void AddBook(Book book)
```

```
{
```

```
    books.Add(book);
```

```
}
```

```
}
```

```
class Program
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        // Create a bookshelf
```

```
        Bookshelf bookshelf = new Bookshelf();
```

```
        // Add books to the bookshelf
```

```
bookshelf.AddBook(new Book { Title = "Book 1", Author = "Author 1" });

bookshelf.AddBook(new Book { Title = "Book 2", Author = "Author 2" });

bookshelf.AddBook(new Book { Title = "Book 3", Author = "Author 3" });


// Access books by title using the indexer

Book book1 = bookshelf["Book 1"];

Book book2 = bookshelf["Book 2"];

Book book4 = bookshelf["Book 4"]; // This will be null if the book is not found


if (book1 != null)
{
    Console.WriteLine($"Book 1 author: {book1.Author}"); // Output: Book 1 author: Author 1
}


if (book2 != null)
{
    Console.WriteLine($"Book 2 author: {book2.Author}"); // Output: Book 2 author: Author 2
}

else
{
    Console.WriteLine("Book 2 not found.");
}


if (book4 != null)
{
```

```
        Console.WriteLine($"Book 4 author: {book4.Author}");  
    }  
    else  
    {  
        Console.WriteLine("Book 4 not found."); // Output: Book 4 not found.  
    }  
}  
}
```

Ques:-8 Create an enum representing the seasons and write a switch statement that prints a message based on the current season.

Ans:- using System;

```
public enum Season  
{  
    Spring,  
    Summer,  
    Autumn,  
    Winter  
}
```

```
class Program  
{
```

```
static void Main()
{
    Season currentSeason = Season.Autumn;

    switch (currentSeason)
    {
        case Season.Spring:
            Console.WriteLine("It's spring! Flowers are blooming.");
            break;

        case Season.Summer:
            Console.WriteLine("It's summer! Enjoy the sunshine.");
            break;

        case Season.Autumn:
            Console.WriteLine("It's autumn! Leaves are falling.");
            break;

        case Season.Winter:
            Console.WriteLine("It's winter! Bundle up for the cold.");
            break;

        default:
            Console.WriteLine("Unknown season.");
            break;
    }
}
```

Ques:-9 . Implement an enum to represent different geometric shapes (e.g., Circle, Square, Triangle) and use it to calculate the area of a specific shape.

Ans:- using System;

```
public enum ShapeType
{
    Circle,
    Square,
    Triangle
}

public class Program
{
    public static void Main()
    {
        // Specify the shape type you want to calculate the area for
        ShapeType selectedShape = ShapeType.Circle;

        switch (selectedShape)
        {
            case ShapeType.Circle:
```

```
double circleRadius = 5.0;

double circleArea = CalculateCircleArea(circleRadius);

Console.WriteLine($"Circle Area: {circleArea}");

break;
```

```
case ShapeType.Square:
```

```
double squareSideLength = 4.0;

double squareArea = CalculateSquareArea(squareSideLength);

Console.WriteLine($"Square Area: {squareArea}");

break;
```

```
case ShapeType.Triangle:
```

```
double triangleBase = 6.0;

double triangleHeight = 8.0;

double triangleArea = CalculateTriangleArea(triangleBase, triangleHeight);

Console.WriteLine($"Triangle Area: {triangleArea}");

break;
```

```
default:
```

```
Console.WriteLine("Unknown shape type.");

break;
```

```
}
```

```
}
```

```
// Calculate the area of a circle
```



```
public static double CalculateCircleArea(double radius)
{
    return Math.PI * radius * radius;
}

// Calculate the area of a square
public static double CalculateSquareArea(double sideLength)
{
    return sideLength * sideLength;
}

// Calculate the area of a triangle
public static double CalculateTriangleArea(double @base, double height)
{
    return 0.5 * @base * height;
}
}
```

Ques:-10 Create an enum with flags to represent the permission levels (Read, Write, Execute) of a file, and demonstrate how to combine these permissions for a user.

Ans:- using System;

[Flags]

```
public enum FileAccessPermission
```

```
{
```

```
    None = 0,    // No permissions
```

```
    Read = 1 << 0, // Read permission
```

```
    Write = 1 << 1, // Write permission
```

```
    Execute = 1 << 2 // Execute permission
```

```
}
```

```
class Program
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        // Demonstrate combining permissions for a user
```

```
        FileAccessPermission userPermissions = FileAccessPermission.Read | FileAccessPermission.Write;
```

```
        // Check if the user has Read permission
```

```
        if ((userPermissions & FileAccessPermission.Read) == FileAccessPermission.Read)
```

```
        {
```

```
            Console.WriteLine("User has Read permission.");
```

```
        }
```

```
        else
```

```
        {
```

```
            Console.WriteLine("User does not have Read permission.");
```

```
        }
```

```
// Check if the user has Write permission

if ((userPermissions & FileAccessPermission.Write) == FileAccessPermission.Write)
{
    Console.WriteLine("User has Write permission.");
}

else
{
    Console.WriteLine("User does not have Write permission.");
}


// Check if the user has Execute permission

if ((userPermissions & FileAccessPermission.Execute) == FileAccessPermission.Execute)
{
    Console.WriteLine("User has Execute permission.");
}

else
{
    Console.WriteLine("User does not have Execute permission.");
}

}
```

