

Introduction to the .NET Framework

The .NET Framework is a software development platform developed by Microsoft. It provides a comprehensive and consistent programming model for building applications that can run on various platforms, including Windows, macOS, and Linux. The framework consists of several components, including a large class library called the Base Class Library (BCL) and a runtime environment called the Common Language Runtime (CLR).

Common Language Runtime (CLR)

The Common Language Runtime (CLR) is the execution environment of the .NET Framework. It provides various services, including memory management, exception handling, type safety, and garbage collection. One of the key features of the CLR is its support for multiple programming languages. Code written in languages such as C#, Visual Basic.NET, F#, and others can all be compiled into a common intermediate language (CIL or MSIL), which is then executed by the CLR.

Base Class Library (BCL)

The Base Class Library (BCL) is a collection of reusable classes, interfaces, and value types that provide the foundation for .NET development. It includes classes for working with strings, collections, input/output operations, networking, and many other common tasks. By leveraging the BCL, developers can write code more efficiently and focus on solving specific problems rather than reinventing the wheel.

Language Interoperability

One of the strengths of the .NET Framework is its support for language interoperability. This means that code written in one .NET language can seamlessly interact with code written in another .NET language. For example, a C# class can inherit from a class written in Visual Basic.NET, and objects created in one language can be passed as parameters to methods written in another language.

Development Tools

The .NET Framework comes with a set of powerful development tools that streamline the process of creating, debugging, and deploying applications. Visual Studio,

Microsoft's integrated development environment (IDE), offers a rich set of features for C# development, including code editing, debugging, profiling, and version control integration.

Cross-Platform Development

With the introduction of .NET Core (now .NET 5 and later), Microsoft has expanded the reach of the .NET Framework beyond Windows. .NET Core is a cross-platform, open-source implementation of the .NET Framework that allows developers to build and run applications on Windows, macOS, and Linux. This enables developers to target a wider range of devices and platforms with their applications.

Introduction to C#.

C# (pronounced "C sharp") is a powerful and versatile programming language developed by Microsoft. It is widely used in the development of various types of applications, including desktop, web, mobile, and gaming applications. Here are some reasons why C# is used and considered valuable for future development:

- **Versatility:** C# can be used to develop a wide range of applications, including Windows desktop applications, web applications, mobile apps (using Xamarin), games (using Unity), and even IoT (Internet of Things) applications.
- **Integration with the .NET Framework:** C# is closely integrated with the .NET Framework, which provides a vast library of classes and functions for building applications. The .NET Framework also offers features such as garbage collection, memory management, and security, which help developers build robust and secure applications more efficiently.
- **Object-Oriented Programming (OOP) Language:** C# is an object-oriented programming language, which means it allows developers to organize their code into reusable objects and classes. This makes it easier to manage large codebases and write code that is modular, scalable, and maintainable.
- **Strong Typing and Type Safety:** C# is a statically typed language, which means that variables must be explicitly declared with their data types at compile time. This helps catch errors at compile time rather than at runtime, leading to more stable and reliable code.

- **Language Features:** C# continues to evolve with new language features and improvements introduced in each new version. Some of the features introduced in recent versions include `async/await` for asynchronous programming, pattern matching, local functions, and nullable reference types.
- **Cross-Platform Development:** With the introduction of .NET Core (now known as .NET 5 and later), C# has become a cross-platform language, allowing developers to build and run applications on different operating systems, including Windows, macOS, and Linux.
- **Industry Demand:** C# is widely used in the software industry, particularly in enterprise environments and among companies that develop Microsoft-based solutions. Learning C# can open up a wide range of job opportunities for developers in various industries.

How to Run C# code in Visual Studio Code

- To Install Visual Studio Code please follow the steps provided in the link: <https://code.visualstudio.com/docs/setup/windows>
- To Install Microsoft .NET Framework please follow the steps provided in the link: <https://learn.microsoft.com/en-us/dotnet/core/install/windows?tabs=net80>

Steps:

- Open Visual Studio Code.
Create a Folder of program name.
- Create the Console Project using the cmd **"dotnet new Console"** under the Folder.
- Type your code and save.
- Compile the code using cmd **"dotnet build"**.
- Run the code with cmd **"dotnet run"**.
- Check the output in the Console.

Note: If provided with stable Internet Access follow above steps.

Or

- Open visual studio Code.
- Create a folder.
- Under the folder, create a new file ,type program and save with “.cs “ extension. Ex: program.cs
- To compile the code in Visual Studio Code ->Open Terminal-> New Terminal -> Select Command Prompt and type “csc Program name.cs”.
Ex: csc program.cs
- To run the code type only program name. Ex: Program.
- Check the output in the Console.

Note: If stable Internet Access follow above steps.

Program 1: Develop a C# program to simulate simple Arithmetic Calculator for Addition, Subtraction, Multiplication, Division and Mod operations. Read the operator and operands through console.

```
using System;
class Calculator
{
    static void Main()
    {
        Console.WriteLine("Simple Arithmetic Calculator");

        // Read operator
        Console.Write("Enter operator (+, -, *, /, %): ");
        char op = Convert.ToChar(Console.ReadLine());

        // Read operands
        Console.Write("Enter first operand: ");
        double operand1 = Convert.ToDouble(Console.ReadLine());

        Console.Write("Enter second operand: ");
        double operand2 = Convert.ToDouble(Console.ReadLine());

        // Perform calculation based on operator
        double result = 0;

        switch (op)
        {
            case '+':
                result = operand1 + operand2;
                break;
```

```
case '-':
    result = operand1 - operand2;
    break;

case '*':
    result = operand1 * operand2;
    break;

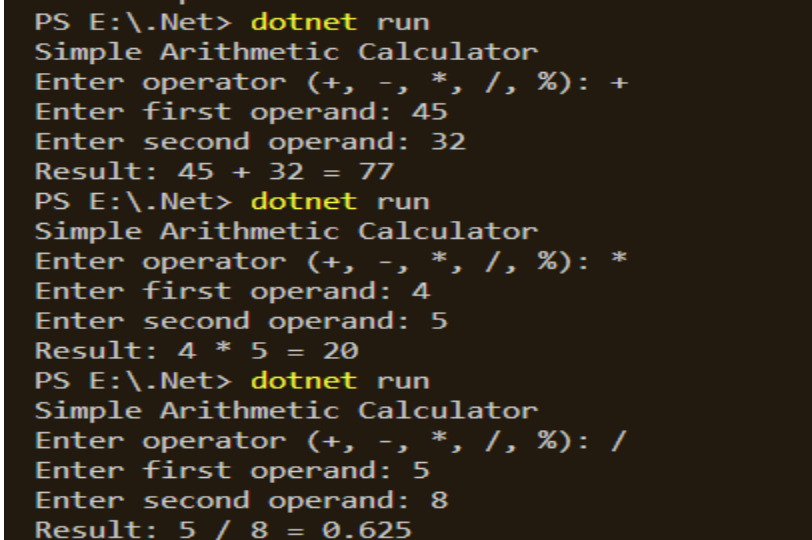
case '/':
    // Check for division by zero
    if (operand2 != 0)
    {
        result = operand1 / operand2;
    }
    else
    {
        Console.WriteLine("Error: Cannot divide by zero.");
        return;
    }
    break;

case '%':
    // Check for modulus by zero
    if (operand2 != 0)
    {
        result = operand1 % operand2;
    }
    else
    {
        Console.WriteLine("Error: Cannot find modulus with zero.");
        return;
    }
}
```

```
        }  
        break;  
  
    default:  
        Console.WriteLine("Error: Invalid operator.");  
        return;  
    }  
  
    // Display the result  
    Console.WriteLine("Result:" +result);  
}  
}
```

This program first prompts the user to enter the first operand, then the operator, and finally the second operand. It performs the selected operation and displays the result. The program includes basic input validation to handle invalid inputs and prevent division by zero or modulo by zero errors.

OUTPUT:



```
PS E:\.Net> dotnet run  
Simple Arithmetic Calculator  
Enter operator (+, -, *, /, %): +  
Enter first operand: 45  
Enter second operand: 32  
Result: 45 + 32 = 77  
PS E:\.Net> dotnet run  
Simple Arithmetic Calculator  
Enter operator (+, -, *, /, %): *  
Enter first operand: 4  
Enter second operand: 5  
Result: 4 * 5 = 20  
PS E:\.Net> dotnet run  
Simple Arithmetic Calculator  
Enter operator (+, -, *, /, %): /  
Enter first operand: 5  
Enter second operand: 8  
Result: 5 / 8 = 0.625
```

Program 2: Develop a C# program to print Armstrong Number between 1 to 1000.

```
using System;
```

```
class ArmstrongNumbers
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        Console.WriteLine("Armstrong numbers from 1 to 1000:");
```

```
        for (int i = 1; i <= 1000; i++)
```

```
        {
```

```
            if (IsArmstrongNumber(i))
```

```
            {
```

```
                Console.WriteLine(i);
```

```
            }
```

```
        }
```

```
    }
```

```
    static bool IsArmstrongNumber(int num)
```

```
    {
```

```
        int originalNum = num;
```

```
        int numDigits = CountDigits(num);
```

```
        int sum = 0;
```

```
        while (num > 0)
```

```
        {
```

```
            int digit = num % 10;
```

```
            sum += (int)Math.Pow(digit, numDigits);
```

```
            num /= 10;
```

```
        }
```

```
        return sum == originalNum;
```

```
    }
```

```
    static int CountDigits(int num)
```

```
    {
```

```
        int count = 0;
```

```
        while (num > 0)
```

```
        {
```

```
            num /= 10;
```

```
            count++;
```

```
        }
```

```
        return count;
```

```
    }
```

```
}
```


This program defines a function `IsArmstrongNumber` to check if a given number is an Armstrong number. The `Main` method then iterates through numbers from 1 to 1000 and prints those that are Armstrong numbers.

OUTPUT:

```
PS E:\.Net> dotnet run
Armstrong numbers from 1 to 1000:
1
2
3
4
5
6
7
8
9
153
370
371
407
```

Program 3: Develop a C# program to list all substrings in a given string, [Hint: use of Substring() method]

```
using System;
class Program
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string inputString = Console.ReadLine();

        Console.WriteLine("\nList of all substrings:");

        ListAllSubstrings(inputString);

        Console.ReadLine(); // To keep the console window open
    }

    static void ListAllSubstrings(string input)
    {
        for (int i = 0; i < input.Length; i++)
        {
            for (int j = i + 1; j <= input.Length; j++)
            {
                string substring = input.Substring(i, j - i);
                Console.WriteLine(substring);
            }
        }
    }
}
```

The Main method takes user input for a string. The ListAllSubstrings method is responsible for listing all substrings of the given string. Nested loops are used to iterate through all possible substrings.

OUTPUT:

```
Enter a string: abcd

List of all substrings:
a
ab
abc
abcd
b
bc
bcd
c
cd
d
```

Program 4: Develop a C# program to demonstrate Division by Zero and Index Out of Range exceptions.

```
using System;
```

```
class Program
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        // Division by Zero Exception
```

```
        Console.WriteLine("Division by Zero Exception:");
```

```
        DivisionByZeroExample();
```

```
        // Index Out of Range Exception
```

```
        Console.WriteLine("\nIndex Out of Range Exception:");
```

```
        IndexOutOfRangeExceptionExample();
```

```
    }
```

```
    static void DivisionByZeroExample()
```

```
    {
```

```
        try
```

```
        {
```

```
            int numerator = 10;
```

```
            int denominator = 0;
```

```
            int result = numerator / denominator; // Division by zero
```

```
            Console.WriteLine("Result: " + result);
```

```
        }
```

```
        catch (DivideByZeroException ex)
```

```
        {
```

```
            Console.WriteLine("Error: " + ex.Message);
```

```
        }
```

```
    }
```

```
static void IndexOutOfRangeExceptionExample()
{
    try
    {
        int[] numbers = { 1, 2, 3, 4, 5 };
        int index = 10; // Index out of range
        int value = numbers[index];
        Console.WriteLine("Value at index " + index + ": " + value);
    }
    catch (IndexOutOfRangeException ex)
    {
        Console.WriteLine("Error: " + ex.Message);
    }
}
```

OUTPUT:

In the `DivisionByZeroExample` method, we intentionally attempt to divide by zero, which will throw a `DivideByZeroException`. In the `IndexOutOfRangeExceptionExample` method, we try to access an element in an array using an index that is out of range, resulting in an `IndexOutOfRangeException`. The program catches these exceptions and prints an error message.

```
Time Elapsed 00:00:01.35
PS E:\.Net> dotnet run
Division by Zero Exception:
Error: Attempted to divide by zero.

Index Out of Range Exception:
Error: Index was outside the bounds of the array.
```

Program 5: Develop a C# program to generate and print Pascal Triangle using Two Dimensional arrays.

```
using System;
class Program
{
    static void Main()
    {
        Console.Write("Enter the number of rows for Pascal's Triangle: ");
        int numRows = int.Parse(Console.ReadLine());

        int[,] pascalsTriangle = GeneratePascalsTriangle(numRows);

        Console.WriteLine("\nPascal's Triangle:");
        PrintPascalsTriangle(pascalsTriangle);
    }

    static int[,] GeneratePascalsTriangle(int numRows)
    {
        int[,] triangle = new int[numRows, numRows];

        for (int i = 0; i < numRows; i++)
        {
            for (int j = 0; j <= i; j++)
            {
                if (j == 0 || j == i)
                    triangle[i, j] = 1;
                else
                    triangle[i, j] = triangle[i - 1, j - 1] + triangle[i - 1, j];
            }
        }
    }
}
```

```
        return triangle;
    }

    static void PrintPascalsTriangle(int[,] triangle)
    {
        int numRows = triangle.GetLength(0);

        for (int i = 0; i < numRows; i++)
        {
            for (int j = 0; j <= i; j++)
            {
                Console.Write(triangle[i, j] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

This program defines two functions: **GeneratePascalsTriangle** to generate Pascal's Triangle using a two-dimensional array, and **PrintPascalsTriangle** to print the triangle to the console. The **Main** method takes user input for the number of rows and then calls these functions to generate and print Pascal's Triangle.

OUTPUT:

```
Enter the number of rows for Pascal's Triangle: 5
Pascal's Triangle:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
PS E:\.Net> dotnet run
Enter the number of rows for Pascal's Triangle: 6
Pascal's Triangle:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
PS E:\.Net> 
```

Program 6: Develop a C# program to generate and print Floyds Triangle using Jagged arrays

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Enter the number of rows for Floyd's Triangle:");
        int numberOfRows = Convert.ToInt32(Console.ReadLine());

        int[][] floydsTriangle = GenerateFloydsTriangle(numberOfRows);

        Console.WriteLine("Floyd's Triangle:");
        PrintFloydsTriangle(floydsTriangle);
    }

    static int[][] GenerateFloydsTriangle(int rows)
    {
        int[][] triangle = new int[rows][];

        int currentValue = 1;

        for (int i = 0; i < rows; i++)
        {
            triangle[i] = new int[i + 1];

            for (int j = 0; j <= i; j++)
            {
                triangle[i][j] = currentValue;
                currentValue++;
            }
        }
    }
}
```



```
    }

    return triangle;
}

static void PrintFloydsTriangle(int[][] triangle)
{
    for (int i = 0; i < triangle.Length; i++)
    {
        for (int j = 0; j < triangle[i].Length; j++)
        {
            Console.Write(triangle[i][j] + " ");
        }
        Console.WriteLine();
    }
}
```

The **GenerateFloydsTriangle** method generates the Floyd's Triangle using a jagged array. It takes the number of rows as input and populates the array accordingly.

The **PrintFloydsTriangle** method prints the generated triangle.

In the **Main** method, the user is prompted to enter the number of rows, and then the program generates and prints Floyd's Triangle.

OUTPUT:

```
Time Elapsed 00:00:01.31
PS E:\.Net> dotnet run
Enter the number of rows for Floyd's Triangle:
5
Floyd's Triangle:
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
PS E:\.Net>
PS E:\.Net> dotnet run
Enter the number of rows for Floyd's Triangle:
12
Floyd's Triangle:
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31 32 33 34 35 36
37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 74 75 76 77 78
PS E:\.Net>
```

Program 7: Develop a C# program to read a text file and copy the file contents to another text file.

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        Console.WriteLine("Enter the path of the source text file:");
        string sourceFilePath = Console.ReadLine(); //
C:\Users\Minaz\Desktop\C#\a.txt

        Console.WriteLine("Enter the path of the destination text file:");
        string destinationFilePath = Console.ReadLine(); //
:\Users\Minaz\Desktop\C#\a.txt

        try
        {
            // Read the contents of the source file
            string fileContents = File.ReadAllText(sourceFilePath);

            // Write the contents to the destination file
            File.WriteAllText(destinationFilePath, fileContents);

            Console.WriteLine("File contents copied successfully.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("An error occurred:" + ex.Message);
        }
    }
}
```

```
}
```

This program prompts the user to enter the paths of the source and destination text files. It then reads the contents of the source file using `File.ReadAllText` and writes them to the destination file using `File.WriteAllText`. The program includes error handling to catch any exceptions that may occur during file operations.

OUTPUT:

```
PS E:\.Net> dotnet run
Enter the path of the source text file:
C:\Users\Minaz\Desktop\a.txt
Enter the path of the destination text file:
C:\Users\Minaz\Desktop\b.txt
File contents copied successfully.
PS E:\.Net> |
```

Program 8: Develop a C# Program to Implement Stack with Push and Pop Operations [Hint: Use class, get/set properties, methods for push and pop and main method]

```
using System;

class Stack
{
    private const int MaxSize = 100;
    private int[] stackArray;
    private int top;

    public Stack()
    {
        stackArray = new int[MaxSize];
        top = -1; // Stack is initially empty
    }

    static void Main()
    {
        // Creating an instance of the Stack class
        Stack myStack = new Stack();

        // Pushing elements onto the stack
        myStack.Push(10);
        myStack.Push(20);
        myStack.Push(30);

        // Displaying the elements in the stack
        Console.WriteLine("Elements in the stack:");
        myStack.DisplayStack();

        // Popping an element from the stack
    }
}
```

```
int poppedElement = myStack.Pop();
Console.WriteLine("Popped element:" +poppedElement);

// Displaying the elements in the stack after popping
Console.WriteLine("Elements in the stack after popping:");
myStack.DisplayStack();
}

public void Push(int item)
{
    if (top == MaxSize - 1)
    {
        Console.WriteLine("Stack overflow. Cannot push element onto the
stack.");
    }
    else
    {
        stackArray[++top] = item;
        Console.WriteLine("Pushed onto the stack." +item );
    }
}

public int Pop()
{
    if (top == -1)
    {
        Console.WriteLine("Stack underflow. Cannot pop element from an empty
stack.");
        return -1; // Indicates stack underflow
    }
    else
    {
        int poppedElement = stackArray[top--];
```

```
        return poppedElement;
    }
}

public void DisplayStack()
{
    if (top == -1)
    {
        Console.WriteLine("Stack is empty.");
    }
    else
    {
        for (int i = top; i >= 0; i--)
        {
            Console.WriteLine(stackArray[i]);
        }
    }
}
```

This program defines a **Stack** class with methods for push, pop, and displaying the elements in the stack. The **Main** method in the **StackExample** class demonstrates how to use the stack by pushing and popping elements

OUTPUT:

```
Time Elapsed 00:00:01.50
PS E:\.Net> dotnet run
Pushed 10 onto the stack.
Pushed 20 onto the stack.
Pushed 30 onto the stack.
Elements in the stack:
30
20
10
Popped element: 30
Elements in the stack after popping:
20
10
```

Program 9: Design a class "Complex" with data members, constructor and method for overloading a binary operator + Develop a C# program to read Two complex number and Print the results of addition.

```
using System;
class Complex
{
    private double real;
    private double imaginary;

    // Constructor
    public Complex(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Overloading the + operator
    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
    }

    // Method to display the complex number
    public void Display()
    {
        Console.WriteLine("Result:" +real +"+" +imaginary +"i");
    }

    static void Main()
    {
        Console.WriteLine("Enter the first complex number:");
```



```
Console.Write("Real part: ");
double real1 = Convert.ToDouble(Console.ReadLine());
Console.Write("Imaginary part: ");
double imaginary1 = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("\nEnter the second complex number:");
Console.Write("Real part: ");
double real2 = Convert.ToDouble(Console.ReadLine());
Console.Write("Imaginary part: ");
double imaginary2 = Convert.ToDouble(Console.ReadLine());

// Creating objects of Complex class
Complex complex1 = new Complex(real1, imaginary1);
Complex complex2 = new Complex(real2, imaginary2);

// Adding two complex numbers
Complex result = complex1 + complex2;

// Displaying the result
Console.WriteLine("\nResult of Addition:");
result.Display();
}
}
```

This program defines a **Complex** class with a constructor that initializes the real and imaginary parts of a complex number. The **+** operator is overloaded to add two complex numbers, and a **Display** method is provided to print the result. In the **Main** method, the program prompts the user to enter two complex numbers, creates instances of the **Complex** class, adds them using the overloaded **+** operator, and then displays the result.

OUTPUT:

```
Time Elapsed 00:00:01.36
PS E:\.Net> dotnet run
Enter the first complex number:
Real part: 10
Imaginary part: 23

Enter the second complex number:
Real part: 32
Imaginary part: 12

Result of Addition:
Result: 42 + 35i
```

Program 10: Develop a C# program to create a class named shape. Create three sub classes namely: circle, triangle and square, each class has two member functions named draw and erase (). Demonstrate polymorphism concepts by developing suitable methods, defining member data and main program.

```
using System;
```

```
// Base class
```

```
class Shape
```

```
{  
    public virtual void Draw()  
    {  
        Console.WriteLine("Drawing a generic shape");  
    }  
  
    public virtual void Erase()  
    {  
        Console.WriteLine("Erasing a generic shape");  
    }  
}
```

```
// Derived class Circle
```

```
class Circle : Shape
```

```
{  
    public override void Draw()  
    {  
        Console.WriteLine("Drawing a circle");  
    }  
  
    public override void Erase()  
    {  
        Console.WriteLine("Erasing a circle");  
    }  
}
```

```
    }  
}  
  
// Derived class Triangle  
class Triangle : Shape  
{  
    public override void Draw()  
    {  
        Console.WriteLine("Drawing a triangle");  
    }  
  
    public override void Erase()  
    {  
        Console.WriteLine("Erasing a triangle");  
    }  
}  
  
// Derived class Square  
class Square : Shape  
{  
    public override void Draw()  
    {  
        Console.WriteLine("Drawing a square");  
    }  
  
    public override void Erase()  
    {  
        Console.WriteLine("Erasing a square");  
    }  
}  
  
class Program
```

```
{
    static void Main()
    {
        // Creating objects of each shape
        Shape circle = new Circle();
        Shape triangle = new Triangle();
        Shape square = new Square();

        // Demonstrating polymorphism
        Console.WriteLine("Demonstrating polymorphism:");
        Console.WriteLine("-----");

        // Calling Draw and Erase methods for each shape
        DrawAndErase(circle);
        DrawAndErase(triangle);
        DrawAndErase(square);
    }

    // Polymorphic function that can accept any shape and call its Draw and Erase
    methods
    static void DrawAndErase(Shape shape)
    {
        shape.Draw();
        shape.Erase();
        Console.WriteLine(); // Adding a newline for better readability
    }
}
```

DrawAndErase function demonstrates polymorphism by accepting a parameter of type **Shape**. This function can take any shape object (whether it's a **Circle**, **Triangle**, or **Square**) and call its **Draw** and **Erase** methods. The actual implementation that gets executed is determined at runtime based on the type of the object.

OUTPUT:

```
Demonstrating polymorphism:
```

```
-----
```

```
Drawing a circle
```

```
Erasing a circle
```

```
Drawing a triangle
```

```
Erasing a triangle
```

```
Drawing a square
```

```
Erasing a square
```

Program 11: Develop a C# program to create an abstract class Shape with abstract methods calculateArea() and calculatePerimeter(). Create subclasses Circle and Triangle that extend the Shape class and implement the respective methods to calculate the area and perimeter of each shape.

```
using System;
```

```
// Abstract Shape class
```

```
public abstract class Shape
```

```
{  
    public abstract double CalculateArea();  
    public abstract double CalculatePerimeter();  
}
```

```
// Circle class, subclass of Shape
```

```
public class Circle : Shape
```

```
{  
    public double Radius { get; set; }  
  
    public Circle(double radius)  
    {  
        Radius = radius;  
    }  
  
    public override double CalculateArea()  
    {  
        return Math.PI * Math.Pow(Radius, 2);  
    }  
  
    public override double CalculatePerimeter()  
    {  
        return 2 * Math.PI * Radius;  
    }  
}
```

```
    }  
}  
  
// Triangle class, subclass of Shape  
public class Triangle : Shape  
{  
    public double Side1 { get; set; }  
    public double Side2 { get; set; }  
    public double Side3 { get; set; }  
  
    public Triangle(double side1, double side2, double side3)  
    {  
        Side1 = side1;  
        Side2 = side2;  
        Side3 = side3;  
    }  
  
    public override double CalculateArea()  
    {  
        // Using Heron's formula to calculate the area of a triangle  
        double s = (Side1 + Side2 + Side3) / 2;  
        return Math.Sqrt(s * (s - Side1) * (s - Side2) * (s - Side3));  
    }  
  
    public override double CalculatePerimeter()  
    {  
        return Side1 + Side2 + Side3;  
    }  
}  
  
// Main program  
class Program
```



```
{
    static void Main()
    {
        // Example usage
        Circle circle = new Circle(5);
        Triangle triangle = new Triangle(3, 4, 5);

        Console.WriteLine("Circle:");
        Console.WriteLine("Area: " + circle.CalculateArea());
        Console.WriteLine("Perimeter: " + circle.CalculatePerimeter());

        Console.WriteLine("\nTriangle:");
        Console.WriteLine("Area: " + triangle.CalculateArea());
        Console.WriteLine("Perimeter: " + triangle.CalculatePerimeter());
    }
}
```

Shape class declares two abstract methods **CalculateArea()** and **CalculatePerimeter()**. The **Circle** and **Triangle** classes then extend **Shape** and provide their own implementations for these methods. The **Main** method demonstrates how to create instances of these classes and calculate the area and perimeter for a circle and a triangle.

OUTPUT:

```
Time Elapsed 00:00:01.43
PS E:\.Net> dotnet run
Circle:
Area: 78.53981633974483
Perimeter: 31.41592653589793

Triangle:
Area: 6
Perimeter: 12
PS E:\.Net> 
```

Program 12: Develop a C# program to create an interface Resizable with methods `resizeWidth(int width)` and `resizeHeight(int height)` that allow an object to be resized. Create a class `Rectangle` that implements the `Resizable` interface and implements the resize methods

```
using System;
// Define the Resizable interface
public interface Resizable
{
    void ResizeWidth(int width);
    void ResizeHeight(int height);
}
// Implement the Rectangle class that implements the Resizable interface
public class Rectangle : Resizable
{
    private int width;
    private int height;

    public Rectangle(int width, int height)
    {
        this.width = width;
        this.height = height;
    }

    public void Display()
    {
        Console.WriteLine("Rectangle: Width =" +width +"Height =" +height);
    }

    public void ResizeWidth(int newWidth)
    {
        if (newWidth > 0)
        {
            width = newWidth;
            Console.WriteLine("Resized width to " +width);
        }
        else
        {
            Console.WriteLine("Width must be a positive value.");
        }
    }

    public void ResizeHeight(int newHeight)
```

```
{
    if (newHeight > 0)
    {
        height = newHeight;
        Console.WriteLine("Resized height to " +height);
    }
    else
    {
        Console.WriteLine("Height must be a positive value.");
    }
}
}
```

```
class Program
{
    static void Main()
    {
        // Create an instance of Rectangle
        Rectangle rectangle = new Rectangle(10, 5);

        // Display the initial state of the rectangle
        Console.WriteLine("Initial State:");
        rectangle.Display();

        // Resize the rectangle
        rectangle.ResizeWidth(15);
        rectangle.ResizeHeight(8);

        // Display the final state of the rectangle
        Console.WriteLine("Final State:");
        rectangle.Display();
    }
}
```

Resizable interface declares two methods, **ResizeWidth** and **ResizeHeight**. The **Rectangle** class implements this interface and provides concrete implementations for these methods. The **Main** method demonstrates how to create an instance of the **Rectangle** class, display its initial state, and then resize it using the **ResizeWidth** and **ResizeHeight** methods.

OUTPUT:

```
Time Elapsed 00:00:01.34
PS E:\.Net> dotnet run
Initial State:
Rectangle: Width = 10, Height = 5
Resized width to 15
Resized height to 8
Final State:
Rectangle: Width = 15, Height = 8
PS E:\.Net> 
```