**Interface**

// Define an interface

public interface IMyInterface

{

// Declare method signatures

int Calculate(int x, int y);

string GetString();

}

// Implement the interface in a class

public class MyClass : IMyInterface

{

public int Calculate(int x, int y)

{

// Implementation for Calculate

return x + y;

}

public string GetString()

{

// Implementation for GetString

return "Hello, World!";

}

}

class Program

{

static void Main(string[] args)

{

IMyInterface myObject = new MyClass();

int result = myObject.Calculate(10, 5);

string text = myObject.GetString();

Console.WriteLine($"Result: {result}");

Console.WriteLine($"Text: {text}");

}

}

Result 15

Helloworld

**Interface to interface**

using System;

// Define the first interface

public interface IFirstInterface

{

void FirstMethod();

}

// Define the second interface that inherits from the first interface

public interface ISecondInterface : IFirstInterface

{

void SecondMethod();

}

// Implement the second interface in a class

public class MyClass : ISecondInterface

{

public void FirstMethod()

{

Console.WriteLine("FirstMethod implementation");

}

public void SecondMethod()

{

Console.WriteLine("SecondMethod implementation");

}

}

class Program

{

static void Main(string[] args)

{

MyClass myObject = new MyClass();

myObject.FirstMethod();

myObject.SecondMethod();

}

}

Output:

FirstMethod implementation

SecondMethod implementation

**Properties:**

using System;

// Define a class with properties

public class Person

{

// Private fields to store the data

private string firstName;

private string lastName;

private int age;

// Public properties with get and set accessors

public string FirstName

{

get { return firstName; }

set

{

if (!string.IsNullOrEmpty(value))

{

firstName = value;

}

else

{

Console.WriteLine("First name cannot be empty.");

}

}

}

public string LastName

{

get { return lastName; }

set

{

if (!string.IsNullOrEmpty(value))

{

lastName = value;

}

else

{

Console.WriteLine("Last name cannot be empty.");

}

}

}

public int Age

{

get { return age; }

set

{

if (value >= 0)

{

age = value;

}

else

{

Console.WriteLine("Age cannot be negative.");

}

}

}

// Constructor

public Person(string firstName, string lastName, int age)

{

FirstName = firstName;

LastName = lastName;

Age = age;

}

// Method to display information

public void DisplayInfo()

{

Console.WriteLine($"Name: {FirstName} {LastName}, Age: {Age}");

}

}

class Program

{

static void Main(string[] args)

{

// Create a Person object and set properties

Person person = new Person("John", "Doe", 30);

// Access properties and display information

person.DisplayInfo();

// Attempt to set invalid values for properties

person.FirstName = ""; // This will print an error message.

person.Age = -5; // This will print an error message.

// Display updated information

person.DisplayInfo();

}

}

Output

Name: John Doe, Age: 30

First name cannot be empty.

Age cannot be negative.

Name: John Doe, Age: 30

**Collection**

**Queue**

using System;

using System.Collections.Generic;

class Program

{

static void Main(string[] args)

{

// Create a new queue of integers

Queue<int> myQueue = new Queue<int>();

// Enqueue (add) elements to the queue

myQueue.Enqueue(10);

myQueue.Enqueue(20);

myQueue.Enqueue(30);

myQueue.Enqueue(40);

myQueue.Enqueue(50);

// Display the elements in the queue

Console.WriteLine("Queue elements:");

foreach (int item in myQueue)

{

Console.WriteLine(item);

}

// Dequeue (remove) elements from the queue

int dequeuedItem = myQueue.Dequeue();

Console.WriteLine($"Dequeued item: {dequeuedItem}");

// Peek at the front element without removing it

int frontItem = myQueue.Peek();

Console.WriteLine($"Front item (peek): {frontItem}");

// Display the elements in the queue after dequeue

Console.WriteLine("Queue elements after dequeue:");

foreach (int item in myQueue)

{

Console.WriteLine(item);

}

}

}

Queue elements:

10

20

30

40

50

Dequeued item: 10

Front item (peek): 20

Queue elements after dequeue:

20

30

40

50

**Stack**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// Create a stack of integers

Stack<int> stack = new Stack<int>();

// Push elements onto the stack

stack.Push(1);

stack.Push(2);

stack.Push(3);

// Peek at the top element without removing it

int topElement = stack.Peek();

Console.WriteLine($"Top element: {topElement}");

// Pop elements from the stack

int poppedElement1 = stack.Pop();

int poppedElement2 = stack.Pop();

Console.WriteLine($"Popped elements: {poppedElement1}, {poppedElement2}");

// Check if the stack is empty

bool isEmpty = stack.Count == 0;

Console.WriteLine($"Is the stack empty? {isEmpty}");

// Iterate over the remaining elements in the stack

Console.WriteLine("Remaining elements in the stack:");

foreach (int element in stack)

{

Console.WriteLine(element);

}

}

}

Top element: 3

Popped elements: 3, 2

Is the stack empty? False

Remaining elements in the stack:

1

**List**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// Create a list of integers

List<int> myList = new List<int>();

// Add elements to the list

myList.Add(1);

myList.Add(2);

myList.Add(3);

// Access elements by index

int firstElement = myList[0];

Console.WriteLine($"First element: {firstElement}");

// Modify elements by index

myList[1] = 4;

Console.WriteLine($"Modified list: {string.Join(", ", myList)}");

// Remove elements by value

myList.Remove(3);

Console.WriteLine($"List after removing 3: {string.Join(", ", myList)}");

// Check if an element exists in the list

bool containsTwo = myList.Contains(2);

Console.WriteLine($"Does the list contain 2? {containsTwo}");

// Get the number of elements in the list

int count = myList.Count;

Console.WriteLine($"Number of elements in the list: {count}");

// Iterate over the elements in the list

Console.WriteLine("Elements in the list:");

foreach (int element in myList)

{

Console.WriteLine(element);

}

}

}

First element: 1

Modified list: 1, 4, 3

List after removing 3: 1, 4

Does the list contain 2? True

Number of elements in the list: 2

Elements in the list:

1

4

**Sortedlist**

using System;

using System.Collections.Generic;

class Program

{

static void Main(string[] args)

{

// Create a new sorted list with integer keys and string values

SortedList<int, string> sortedList = new SortedList<int, string>();

// Add elements to the sorted list

sortedList.Add(3, "Apple");

sortedList.Add(1, "Banana");

sortedList.Add(2, "Cherry");

sortedList.Add(4, "Date");

// Access elements by key

Console.WriteLine("Element at key 2: " + sortedList[2]);

// Display all elements in sorted order

Console.WriteLine("Sorted List Elements:");

foreach (var kvp in sortedList)

{

Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");

}

// Check if a key exists

bool containsKey = sortedList.ContainsKey(3);

Console.WriteLine("Contains key 3: " + containsKey);

// Check if a value exists

bool containsValue = sortedList.ContainsValue("Grapes");

Console.WriteLine("Contains value 'Grapes': " + containsValue);

}

}

Element at key 2: Cherry

Sorted List Elements:

Key: 1, Value: Banana

Key: 2, Value: Cherry

Key: 3, Value: Apple

Key: 4, Value: Date

Contains key 3: True

Contains value 'Grapes': False

**Generics with class:**

using System;

// Define a generic class called 'GenericClass'

public class GenericClass<T>

{

private T data;

// Constructor to initialize the data

public GenericClass(T inputData)

{

data = inputData;

}

// A method that returns the stored data

public T GetData()

{

return data;

}

}

class Program

{

static void Main(string[] args)

{

// Create an instance of GenericClass with integer data

GenericClass<int> intGeneric = new GenericClass<int>(42);

Console.WriteLine("Integer Data: " + intGeneric.GetData());

// Create an instance of GenericClass with string data

GenericClass<string> stringGeneric = new GenericClass<string>("Hello, Generics!");

Console.WriteLine("String Data: " + stringGeneric.GetData());

// Create an instance of GenericClass with double data

GenericClass<double> doubleGeneric = new GenericClass<double>(3.14159);

Console.WriteLine("Double Data: " + doubleGeneric.GetData());

// You can use the same GenericClass type with different data types.

}

}

Integer Data: 42

String Data: Hello, Generics!

Double Data: 3.14159

**Generics with method**

using System;

public class GenericMethods

{

// A generic method that swaps two values of any type

public static void Swap<T>(ref T a, ref T b)

{

T temp = a;

a = b;

b = temp;

}

// A generic method that compares two values of any type

public static bool AreEqual<T>(T value1, T value2)

{

return value1.Equals(value2);

}

}

class Program

{

static void Main(string[] args)

{

// Swap two integers

int num1 = 5;

int num2 = 10;

Console.WriteLine($"Before Swap: num1 = {num1}, num2 = {num2}");

GenericMethods.Swap(ref num1, ref num2);

Console.WriteLine($"After Swap: num1 = {num1}, num2 = {num2}");

// Compare two strings

string str1 = "Hello";

string str2 = "World";

bool areStringsEqual = GenericMethods.AreEqual(str1, str2);

Console.WriteLine($"Are Strings Equal: {areStringsEqual}");

}

}

Before Swap: num1 = 5, num2 = 10

After Swap: num1 = 10, num2 = 5

Are Strings Equal: False

**Simple delegate**

using System;

delegate int Calculator(int n);//declaring delegate

public class DelegateExample

{

    static int number = 100;

    public static int add(int n)

    {

        number = number + n;

        return number;

    }

    public static int mul(int n)

    {

        number = number \* n;

        return number;

    }

    public static int getNumber()

    {

        return number;

    }

    public static void Main(string[] args)

    {

        Calculator c1 = new Calculator(add);//instantiating delegate

        Calculator c2 = new Calculator(mul);

        c1(20);//calling method using delegate

        Console.WriteLine("After c1 delegate, Number is: " + getNumber());

        c2(3);

        Console.WriteLine("After c2 delegate, Number is: " + getNumber());

    }

}

Output:

After c1 delegate, Number is: 120

After c2 delegate, Number is: 360

**Multicast delegate**

using System;

public delegate void MyDelegate(string message);

class Program

{

static void Main(string[] args)

{

// Create an instance of the multicast delegate

MyDelegate multicastDelegate = null;

// Add methods to the delegate

multicastDelegate += DisplayMessage1;

multicastDelegate += DisplayMessage2;

multicastDelegate += DisplayMessage3;

// Invoke the multicast delegate

if (multicastDelegate != null)

{

multicastDelegate("Multicast Delegate Example");

}

}

static void DisplayMessage1(string message)

{

Console.WriteLine("Message 1: " + message);

}

static void DisplayMessage2(string message)

{

Console.WriteLine("Message 2: " + message);

}

static void DisplayMessage3(string message)

{

Console.WriteLine("Message 3: " + message);

}

}

Message 1: Multicast Delegate Example

Message 2: Multicast Delegate Example

Message 3: Multicast Delegate Example

**Deletage with return type arguments**

using System;

// Declare a delegate type with a return type and arguments

delegate int CalculatorDelegate(int a, int b);

class Program

{

static void Main(string[] args)

{

// Instantiate the delegate with a method

CalculatorDelegate addDelegate = Add;

CalculatorDelegate subtractDelegate = Subtract;

// Use the delegates to perform calculations

int result1 = addDelegate(5, 3);

int result2 = subtractDelegate(10, 4);

Console.WriteLine("Result of addition: " + result1);

Console.WriteLine("Result of subtraction: " + result2);

}

// Method that matches the delegate signature (addition)

static int Add(int a, int b)

{

return a + b;

}

// Another method that matches the delegate signature (subtraction)

static int Subtract(int a, int b)

{

return a - b;

}

}

Result of addition: 8

Result of subtraction: 6

**simple delegate with no return type**

using System;

// Define a delegate with no return type

public delegate void SimpleDelegate(string message);

class Program

{

static void Main()

{

// Create an instance of the delegate and associate it with a method

SimpleDelegate simpleDelegate = DisplayMessage;

// Invoke the delegate

simpleDelegate("Hello, world!");

}

// Method that matches the delegate signature

static void DisplayMessage(string message)

{

Console.WriteLine(message);

}

}

Hello, world!

**Delegate with return and no arguments**

using System;

// Define a delegate with a return type and no arguments

public delegate int SimpleDelegate();

class Program

{

static void Main()

{

// Create an instance of the delegate and associate it with a method

SimpleDelegate simpleDelegate = GetNumber;

// Invoke the delegate and store the result

int result = simpleDelegate();

// Display the result

Console.WriteLine("The result is: " + result);

}

// Method that matches the delegate signature (no parameters, returns an int)

static int GetNumber()

{

return 42;

}

}

The result is: 42

**No return type with arguments**

using System;

// Define a delegate with no return type and arguments

public delegate void SimpleDelegate(string message);

class Program

{

static void Main()

{

// Create an instance of the delegate and associate it with a method

SimpleDelegate simpleDelegate = DisplayMessage;

// Invoke the delegate with an argument

simpleDelegate("Hello, world!");

}

// Method that matches the delegate signature (takes a string argument, no return value)

static void DisplayMessage(string message)

{

Console.WriteLine("Message received: " + message);

}

}

Message received: Hello, world!

**Anonyms function**

// C# program to illustrate how an

// anonymous function access variable

// defined in outer method

using System;

class GFG {

// Create a delegate

public delegate void petanim(string pet);

// Main method

static public void Main()

{

string fav = "Rabbit";

// Anonymous method with one parameter

petanim p = delegate(string mypet)

{

Console.WriteLine("My favorite pet is {0}.",

mypet);

// Accessing variable defined

// outside the anonymous function

Console.WriteLine("And I like {0} also.", fav);

};

p("Dog");

}

}

o/p

My favorite pet is Dog.

And I like Rabbit also.

**Lambda**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// Example 1: Simple lambda expression

Func<int, int> square = x => x \* x;

int result = square(5);

Console.WriteLine("Square of 5 is: " + result);

// Example 2: Using lambda expression in a list

List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

List<int> squaredNumbers = numbers.Select(x => x \* x).ToList();

Console.WriteLine("Squared numbers:");

foreach (var num in squaredNumbers)

{

Console.WriteLine(num);

}

// Example 3: Lambda expression with a condition

List<int> evenNumbers = numbers.Where(x => x % 2 == 0).ToList();

Console.WriteLine("Even numbers:");

foreach (var num in evenNumbers)

{

Console.WriteLine(num);

}

}

}

Square of 5 is: 25

Squared numbers:

1

4

9

16

25

Even numbers:

2

4

**Indexer**

using System;

public class MyCollection

{

private int[] data;

// Constructor to initialize the collection

public MyCollection(int size)

{

data = new int[size];

for (int i = 0; i < size; i++)

{

data[i] = i \* 10;

}

}

// Indexer declaration

public int this[int index]

{

get

{

if (index < 0 || index >= data.Length)

{

throw new IndexOutOfRangeException();

}

return data[index];

}

set

{

if (index < 0 || index >= data.Length)

{

throw new IndexOutOfRangeException();

}

data[index] = value;

}

}

}

class Program

{

static void Main()

{

MyCollection collection = new MyCollection(5);

// Using the indexer to get values

Console.WriteLine("Value at index 0: " + collection[0]);

Console.WriteLine("Value at index 2: " + collection[2]);

// Using the indexer to set values

collection[1] = 100;

collection[3] = 300;

Console.WriteLine("Updated value at index 1: " + collection[1]);

Console.WriteLine("Updated value at index 3: " + collection[3]);

}

}

Value at index 0: 0

Value at index 2: 20

Updated value at index 1: 100

Updated value at index 3: 300

**Indexer overloading**

using System;

class MyCollection

{

private int[] data = new int[10];

// Overloaded indexer with a single integer parameter

public int this[int index]

{

get

{

if (index < 0 || index >= data.Length)

throw new IndexOutOfRangeException("Index is out of range.");

return data[index];

}

set

{

if (index < 0 || index >= data.Length)

throw new IndexOutOfRangeException("Index is out of range.");

data[index] = value;

}

}

// Overloaded indexer with a string parameter

public int this[string key]

{

get

{

if (key == "first")

return data[0];

else if (key == "last")

return data[data.Length - 1];

else

throw new ArgumentException("Invalid key.");

}

}

}

class Program

{

static void Main()

{

MyCollection collection = new MyCollection();

// Using the integer indexer

collection[0] = 42;

int valueAtIndex0 = collection[0];

Console.WriteLine("Value at index 0: " + valueAtIndex0);

// Using the string indexer

collection["first"] = 10;

collection["last"] = 99;

int firstValue = collection["first"];

int lastValue = collection["last"];

Console.WriteLine("First value: " + firstValue);

Console.WriteLine("Last value: " + lastValue);

}

}

Value at index 0: 42

First value: 10

Last value: 99

**Operator overloading**

using System;

class ComplexNumber

{

public double Real { get; set; }

public double Imaginary { get; set; }

public ComplexNumber(double real, double imaginary)

{

Real = real;

Imaginary = imaginary;

}

// Overload the + operator for ComplexNumber objects

public static ComplexNumber operator +(ComplexNumber a, ComplexNumber b)

{

return new ComplexNumber(a.Real + b.Real, a.Imaginary + b.Imaginary);

}

public override string ToString()

{

return $"{Real} + {Imaginary}i";

}

}

class Program

{

static void Main()

{

ComplexNumber num1 = new ComplexNumber(2.5, 3.0);

ComplexNumber num2 = new ComplexNumber(1.5, 2.0);

// Use the overloaded + operator to add two ComplexNumber objects

ComplexNumber sum = num1 + num2;

Console.WriteLine("num1: " + num1);

Console.WriteLine("num2: " + num2);

Console.WriteLine("Sum: " + sum);

}

}

num1: 2.5 + 3i

num2: 1.5 + 2i

Sum: 4 + 5i

**Events with delegate**

using System;

namespace SampleApp {

public delegate string MyDel(string str);

class EventProgram {

event MyDel MyEvent;

public EventProgram() {

this.MyEvent += new MyDel(this.WelcomeUser);

}

public string WelcomeUser(string username) {

return "Welcome " + username;

}

static void Main(string[] args) {

EventProgram obj1 = new EventProgram();

string result = obj1.MyEvent("Tutorials Point");

Console.WriteLine(result);

}

}

}

Welcome Tutorials Point

**Event with eventhandler**

using System;

public class EventPublisher

{

// Declare an event using the EventHandler delegate

public event EventHandler MyEvent;

public void DoSomething()

{

Console.WriteLine("Something is happening...");

// Raise the event

OnMyEvent(EventArgs.Empty);

}

protected virtual void OnMyEvent(EventArgs e)

{

EventHandler handler = MyEvent;

if (handler != null)

{

handler(this, e);

}

}

}

public class EventSubscriber

{

public void Subscribe(EventPublisher publisher)

{

// Subscribe to the event

publisher.MyEvent += HandleEvent;

}

public void Unsubscribe(EventPublisher publisher)

{

// Unsubscribe from the event

publisher.MyEvent -= HandleEvent;

}

private void HandleEvent(object sender, EventArgs e)

{

Console.WriteLine("Event handled by EventSubscriber");

}

}

public class Program

{

public static void Main()

{

EventPublisher publisher = new EventPublisher();

EventSubscriber subscriber = new EventSubscriber();

// Subscribe to the event

subscriber.Subscribe(publisher);

// Trigger the event

publisher.DoSomething();

// Unsubscribe from the event

subscriber.Unsubscribe(publisher);

// Trigger the event again, but this time, no handler will be called

publisher.DoSomething();

Console.ReadLine();

}

}

Output

Something is happening...

Event handled by EventSubscriber

**Passing custom data to event**

using System;

// Custom EventArgs class to hold custom data

public class CustomEventArgs : EventArgs

{

public string Message { get; }

public CustomEventArgs(string message)

{

Message = message;

}

}

public class EventPublisher

{

// Declare an event using a custom delegate

public event EventHandler<CustomEventArgs> MyEvent;

public void DoSomething()

{

Console.WriteLine("Something is happening...");

// Create custom event data

CustomEventArgs eventData = new CustomEventArgs("Custom data from EventPublisher");

// Raise the event with custom data

OnMyEvent(eventData);

}

protected virtual void OnMyEvent(CustomEventArgs e)

{

EventHandler<CustomEventArgs> handler = MyEvent;

if (handler != null)

{

handler(this, e);

}

}

}

public class EventSubscriber

{

public void Subscribe(EventPublisher publisher)

{

// Subscribe to the event

publisher.MyEvent += HandleEvent;

}

public void Unsubscribe(EventPublisher publisher)

{

// Unsubscribe from the event

publisher.MyEvent -= HandleEvent;

}

private void HandleEvent(object sender, CustomEventArgs e)

{

Console.WriteLine($"Event handled by EventSubscriber with message: {e.Message}");

}

}

.

public class Program

{

public static void Main()

{

EventPublisher publisher = new EventPublisher();

EventSubscriber subscriber = new EventSubscriber();

// Subscribe to the event

subscriber.Subscribe(publisher);

// Trigger the event with custom data

publisher.DoSomething();

// Unsubscribe from the event

subscriber.Unsubscribe(publisher);

// Trigger the event again, but this time, no handler will be called

publisher.DoSomething();

Console.ReadLine();

}

}

Output:

Something is happening...

Event handled by EventSubscriber with message: Custom data from EventPublisher

**try with multiple catch in c#**

using System;

public class Program

{

public static void Main()

{

try

{

// Code that may throw exceptions

int[] numbers = { 1, 2, 3 };

Console.WriteLine(numbers[5]); // Accessing an out-of-bounds array element

}

catch (IndexOutOfRangeException e)

{

// Handle an IndexOutOfRangeException

Console.WriteLine("Index out of range exception: " + e.Message);

}

catch (DivideByZeroException e)

{

// Handle a DivideByZeroException

Console.WriteLine("Divide by zero exception: " + e.Message);

}

catch (Exception e)

{

// Handle any other exceptions not caught by previous catch blocks

Console.WriteLine("General exception: " + e.Message);

}

finally

{

// Optional finally block for cleanup code

Console.WriteLine("Cleanup code in the finally block.");

}

Console.WriteLine("Program continues after exception handling.");

}

}

Output:

Index out of range exception: Index was outside the bounds of the array.

Cleanup code in the finally block.

Program continues after exception handling.

**Using throw throws**

using System;

public class Program

{

public static void Main()

{

try

{

Console.Write("Enter a number: ");

int number = int.Parse(Console.ReadLine());

if (number < 0)

{

// Manually throw an exception if the input is negative

throw new ArgumentException("Input must be a non-negative number.");

}

// Calculate the square root of the number

double result = Math.Sqrt(number);

Console.WriteLine($"Square root of {number} is {result}");

}

catch (FormatException e)

{

Console.WriteLine($"FormatException: {e.Message}");

}

catch (ArgumentException e)

{

Console.WriteLine($"ArgumentException: {e.Message}");

}

catch (Exception e)

{

Console.WriteLine($"General Exception: {e.Message}");

}

}

}

Enter a number: -5

ArgumentException: Input must be a non-negative number.

**Inner exception**

using System;

public class Program

{

public static void Main()

{

try

{

DivideByZero(); // This method will throw an exception

}

catch (Exception ex)

{

Console.WriteLine("Outer Exception:");

Console.WriteLine(ex.Message);

if (ex.InnerException != null)

{

Console.WriteLine("\nInner Exception:");

Console.WriteLine(ex.InnerException.Message);

}

}

}

public static void DivideByZero()

{

try

{

int numerator = 10;

int denominator = 0;

int result = numerator / denominator; // This will throw a DivideByZeroException

}

catch (Exception ex)

{

// Create an inner exception and rethrow it

throw new Exception("An error occurred during division.", ex);

}

}

}

Outer Exception:

An error occurred during division.

Inner Exception:

Attempted to divide by zero.

**Outer exception**

using System;

public class Program

{

public static void Main()

{

try

{

int result = Divide(10, 0); // This method will throw an exception

Console.WriteLine("Result: " + result); // This line won't be reached

}

catch (DivideByZeroException ex)

{

Console.WriteLine("Outer Exception (DivideByZeroException):");

Console.WriteLine(ex.Message);

}

catch (Exception ex)

{

Console.WriteLine("Outer Exception (General Exception):");

Console.WriteLine(ex.Message);

}

}

public static int Divide(int numerator, int denominator)

{

try

{

return numerator / denominator; // This may throw a DivideByZeroException

}

catch (Exception ex)

{

// Additional error handling or logging can be done here

throw; // Re-throw the exception

}

}

}

**Outer Exception (DivideByZeroException):**

**Attempted to divide by zero.**

**Or**

**Outer Exception (General Exception):**

**<ExceptionMessage>**

**Custom exception**

class Program

{

static void Main(string[] args)

{

try

{

// Simulate a situation where your custom exception might occur

int result = Divide(10, 0);

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

}

catch (CustomException ex)

{

Console.WriteLine("Custom Exception caught: " + ex.Message);

}

catch (Exception ex)

{

Console.WriteLine("Generic Exception caught: " + ex.Message);

}

}

static int Divide(int numerator, int denominator)

{

if (denominator == 0)

{

throw new CustomException("Division by zero is not allowed.");

}

return numerator / denominator;

}

}

Custom Exception caught: Division by zero is not allowed.

**Single inheritance**

using System;

// Base class

class Vehicle

{

public string Brand { get; set; }

public string Model { get; set; }

public void Start()

{

Console.WriteLine("The vehicle is starting.");

}

public void Stop()

{

Console.WriteLine("The vehicle is stopping.");

}

}

// Derived class inheriting from Vehicle

class Car : Vehicle

{

public void Accelerate()

{

Console.WriteLine("The car is accelerating.");

}

public void Brake()

{

Console.WriteLine("The car is braking.");

}

}

class Program

{

static void Main(string[] args)

{

Car myCar = new Car();

myCar.Brand = "Toyota";

myCar.Model = "Camry";

// Access methods and properties from the base class

Console.WriteLine($"Brand: {myCar.Brand}, Model: {myCar.Model}");

myCar.Start();

myCar.Stop();

// Access methods from the derived class

myCar.Accelerate();

myCar.Brake();

}

}

Brand: Toyota, Model: Camry

The vehicle is starting.

The vehicle is stopping.

The car is accelerating.

The car is braking.

**Multilevel inheritance**

using System;

// Base class

class Vehicle

{

public void Start()

{

Console.WriteLine("The vehicle is starting.");

}

public void Stop()

{

Console.WriteLine("The vehicle is stopping.");

}

}

// Intermediate class inheriting from Vehicle

class Car : Vehicle

{

public void Accelerate()

{

Console.WriteLine("The car is accelerating.");

}

public void Brake()

{

Console.WriteLine("The car is braking.");

}

}

// Derived class inheriting from Car

class ElectricCar : Car

{

public void Charge()

{

Console.WriteLine("The electric car is charging.");

}

}

class Program

{

static void Main(string[] args)

{

ElectricCar myElectricCar = new ElectricCar();

// Access methods from the base class (Vehicle)

myElectricCar.Start();

myElectricCar.Stop();

// Access methods from the intermediate class (Car)

myElectricCar.Accelerate();

myElectricCar.Brake();

// Access methods from the derived class (ElectricCar)

myElectricCar.Charge();

}

}

The vehicle is starting.

The vehicle is stopping.

The car is accelerating.

The car is braking.

The electric car is charging.

**Hierarchical inheritance**

using System;

// Base class

class Shape

{

public void DisplayInfo()

{

Console.WriteLine("This is a shape.");

}

}

// Derived class 1

class Circle : Shape

{

public double Radius { get; set; }

public double CalculateArea()

{

return Math.PI \* Radius \* Radius;

}

}

// Derived class 2

class Rectangle : Shape

{

public double Length { get; set; }

public double Width { get; set; }

public double CalculateArea()

{

return Length \* Width;

}

}

// Derived class 3

class Triangle : Shape

{

public double BaseLength { get; set; }

public double Height { get; set; }

public double CalculateArea()

{

return 0.5 \* BaseLength \* Height;

}

}

class Program

{

static void Main(string[] args)

{

Circle circle = new Circle();

circle.Radius = 5.0;

Rectangle rectangle = new Rectangle();

rectangle.Length = 4.0;

rectangle.Width = 3.0;

Triangle triangle = new Triangle();

triangle.BaseLength = 6.0;

triangle.Height = 8.0;

// Access the common method from the base class

circle.DisplayInfo();

rectangle.DisplayInfo();

triangle.DisplayInfo();

// Access the specific methods from each derived class

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

Console.WriteLine($"Rectangle Area: {rectangle.CalculateArea()}");

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

}

}

This is a shape.

This is a shape.

This is a shape.

Circle Area: 78.53981633974483

Rectangle Area: 12

Triangle Area: 24