**C# opps concept**

OOPS stands for Object-Oriented Programming (OOP). It is a programming paradigm that organizes software design around objects and their interactions. In OOP, objects are instances of classes, which encapsulate data (attributes or properties) and behaviors (methods or functions).

The core principles of OOP include:

**Encapsulation**: It refers to the bundling of data and methods within a class. The data is hidden and can only be accessed through the defined methods, ensuring data integrity and providing a clear interface to interact with the object.

*example;*

using System;

public class Person

{

// Private fields

private string name;

private int age;

// Public properties

public string Name

{

get { return name; }

set { name = value; }

}

public int Age

{

get { return age; }

set { age = value; }

}

// Public method

public void DisplayInformation()

{

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

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

}

}

public class Program

{

public static void Main(string[] args)

{

Person person = new Person();

// Accessing properties

person.Name = "John Doe";

person.Age = 30;

// Accessing method

person.DisplayInformation();

}

}

**Inheritance**: It allows creating new classes based on existing ones. Inheritance promotes code reuse by inheriting the properties and methods of a base class, known as the superclass or parent class. The new class is called the subclass or child class, which can add additional features or modify existing ones.

they have 5types of inheritance

**Single Inheritance**:

Single inheritance allows a class to inherit from a single base class. It's the most common form of inheritance. In C#, a class can only directly inherit from a single class.

using System;

public class Animal

{

public void Eat()

{

Console.WriteLine("The animal is eating.");

}

}

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine("The dog is barking.");

}

}

public class Program

{

public static void Main(string[] args)

{

Dog dog = new Dog();

dog.Eat();

dog.Bark();

}

}

**Multiple Inheritance (through interfaces**):

C# doesn't support multiple inheritance of classes directly, but it allows multiple inheritance of interfaces. By implementing multiple interfaces, a class can inherit the behaviors defined by those interfaces.

**Multilevel Inheritance:**

Multilevel inheritance is a concept where a derived class is created from another derived class. It forms a hierarchical inheritance structure with multiple levels of inheritance.

using System;

public class Animal

{

public void Eat()

{

Console.WriteLine("The animal is eating.");

}

}

public class Mammal : Animal

{

public void Sleep()

{

Console.WriteLine("The mammal is sleeping.");

}

}

public class Dog : Mammal

{

public void Bark()

{

Console.WriteLine("The dog is barking.");

}

}

public class Program

{

public static void Main(string[] args)

{

Dog dog = new Dog();

dog.Eat();

dog.Sleep();

dog.Bark();

}

}

***Hierarchical Inheritance***:

Hierarchical inheritance is a type of inheritance where multiple derived classes inherit from a single base class. Each derived class becomes a base class for subsequent derived classes.

using System;

public class Shape

{

public virtual void Draw()

{

Console.WriteLine("Drawing a shape");

}

}

public class Circle : Shape

{

public override void Draw()

{

Console.WriteLine("Drawing a circle");

}

}

public class Rectangle : Shape

{

public override void Draw()

{

Console.WriteLine("Drawing a rectangle");

}

}

public class Square : Shape

{

public override void Draw()

{

Console.WriteLine("Drawing a square");

}

}

public class Program

{

public static void Main(string[] args)

{

Shape shape1 = new Circle();

Shape shape2 = new Rectangle();

Shape shape3 = new Square();

shape1.Draw(); // Calls the Draw method of Circle class

shape2.Draw(); // Calls the Draw method of Rectangle class

shape3.Draw(); // Calls the Draw method of Square class

}

}

**Hybrid Inheritance:**

Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance. It involves the inheritance of both classes and interfaces.

using System;

public interface IAnimal

{

void Eat();

}

public interface IDog

{

void Bark();

}

public class Animal : IAnimal

{

public void Eat()

{

Console.WriteLine("The animal is eating.");

}

}

public class Dog : Animal, IDog

{

public void Bark()

{

Console.WriteLine("The dog is barking.");

}

}

public class Program

{

public static void Main(string[] args)

{

Dog dog = new Dog();

dog.Eat();

dog.Bark();

}

}

**Polymorphism**: It refers to the ability of objects of different classes to respond to the same method call. Polymorphism allows objects to be treated as instances of their parent class or as instances of their specific class, providing flexibility and extensibility in code.

*Method Overloading*

Method Overloading is a type of polymorphism. It has several names like “Compile Time Polymorphism” or “Static Polymorphism,” and sometimes it is called “Early Binding”.

Method Overloading means creating multiple methods in a class with the same names but different signatures (Parameters). It permits a class, struct, or interface to declare multiple methods with the same name with unique signatures.

The compiler automatically calls the required method to check the number of parameters and their type passed into that method.

using System;

namespace DemoCsharp

{

class Program

{

public int Add(int num1, int num2)

{

return (num1 + num2);

}

public int Add(int num1, int num2, int num3)

{

return (num1 + num2 + num3);

}

public float Add(float num1, float num2)

{

return (num1 + num2);

}

public string Add(string value1, string value2)

{

return (value1 + " " + value2);

}

static void Main(string[] args)

{

Program objProgram = new Program();

Console.WriteLine("Add with two int parameter :" + objProgram.Add(3, 2));

Console.WriteLine("Add with three int parameter :" + objProgram.Add(3, 2, 8));

Console.WriteLine("Add with two float parameter :" + objProgram.Add(3 f, 22 f));

Console.WriteLine("Add with two string parameter :" + objProgram.Add("hello", "world"));

Console.ReadLine();

}

Method Overriding

Method Overriding is a type of polymorphism. It has several names like “Run Time Polymorphism” or “Dynamic Polymorphism,” and sometimes it is called “Late Binding”.

Method Overriding means having two methods with the same name and same signatures [parameters]; one should be in the base class, and another method should be in a derived class [child class]. You can override the functionality of a base class method to create the same name method with the same signature in a derived class. You can achieve method overriding using inheritance. Virtual and Override keywords are used to achieve method overriding.

using System;

namespace DemoCsharp

{

class BaseClass

{

public virtual int Add(int num1, int num2)

{

return (num1 + num2);

}

}

class ChildClass: BaseClass

{

public override int Add(int num1, int num2)

{

if (num1 <= 0 || num2 <= 0)

{

Console.WriteLine("Values could not be less than zero or equals to zero");

Console.WriteLine("Enter First value : ");

num1 = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter First value : ");

num2 = Convert.ToInt32(Console.ReadLine());

}

return (num1 + num2);

}

}

class Program

{

static void Main(string[] args)

{

BaseClass baseClassObj;

baseClassObj = new BaseClass();

Console.WriteLine("Base class method Add :" + baseClassObj.Add(-3, 8));

baseClassObj = new ChildClass();

Console.WriteLine("Child class method Add :" + baseClassObj.Add(-2, 2));

Console.ReadLine();

}

}

}

Abstraction: It involves representing complex real-world entities as simplified models within the software. Abstraction focuses on essential characteristics and behaviors, while hiding unnecessary details. It helps manage complexity and allows developers to work with high-level concepts.

using System;

public abstract class Animal

{

public abstract void MakeSound();

}

public class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Woof!");

}

}

public class Cat : Animal

{

public override void MakeSound()

{

Console.WriteLine("Meow!");

}

}

public class Program

{

public static void Main(string[] args)

{

Animal dog = new Dog();

Animal cat = new Cat();

dog.MakeSound(); // Calls the MakeSound method of Dog class

cat.MakeSound(); // Calls the MakeSound method of Cat class

}

}

These principles facilitate modular, reusable, and maintainable code, promoting code organization, flexibility, and scalability. Object-oriented programming is widely used in various programming languages such as Java, C++, Python, and C#.

Constructors

A constructor is a special method that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

DEFAULT CONSTRUCTORS

A default constructor is a parameterless constructor that is automatically provided by the compiler if no other constructors are defined.

It initializes the object's fields to their default values (e.g., numeric types to 0, reference types to null, etc.).

Example: public MyClass() { }

public class Customer

{

public string firstName;

public string lastName;

public Customer()

{

}

}

class Program

{

static void Main(string[] args)

{

Customer custormer = new Customer();

custormer.firstName = "Farhan";

custormer.lastName = "Ahmed";

Console.WriteLine("Full Name:" + custormer.firstName + " " + custormer.lastName);

Console.ReadLine();

}

}

}

PARAMETERIZED

A parameterized constructor takes one or more parameters to initialize the object with specific values.

It allows you to provide custom initialization logic based on the passed arguments.

Example: public MyClass(int value) { }

class ParameterConstructor

{

public int FirstNumber;

public int SecondNumber;

public ParameterConstructor(int firstNumber, int secondNumber)

{

FirstNumber = firstNumber;

SecondNumber = secondNumber;

}

}

class Program

{

static void Main(string[] args)

{

ParameterConstructor p = new ParameterConstructor(10, 20);

int Result = p.FirstNumber + p.SecondNumber;

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

Console.ReadLine();

}

}

}

private

A private constructor is a constructor with private access modifiers.

It restricts the creation of objects of the class from outside the class itself.

It is commonly used in singleton design patterns or utility classes that should not be instantiated.

Example: private MyClass() { }

using System;

namespace Constructor {

class Car {

// private constructor

private Car () {

Console.WriteLine("Private Constructor");

}

}

class CarDrive {

static void Main(string[] args) {

// call private constructor

Car car1 = new Car();

Console.ReadLine();

}

}

}

COPY CONSTRUCTORS

A copy constructor creates a new object by copying the values from an existing object of the same type.

It is used to create a copy of an object to ensure that modifications to one object do not affect the other.

Example: public MyClass(MyClass other) { }

public class Employee

{

public string firstName;

public string lastName;

public string position;

public int salary;

public Employee()

{

}

// Copy constructor.

public Employee(Employee employee)

{

firstName = employee.firstName;

lastName = employee.lastName;

position = employee.position;

salary = employee.salary;

}

}

class Program

{

static void Main(string[] args)

{

Employee emp = new Employee();

Employee emp1 = new Employee(emp);

Console.WriteLine("Enter your first name:");

emp1.firstName = Convert.ToString(Console.ReadLine());

Console.WriteLine("Enter your last name:");

emp1.lastName = Convert.ToString(Console.ReadLine());

Console.WriteLine("Enter your position:");

emp1.position = Convert.ToString(Console.ReadLine());

Console.WriteLine("Enter your salary:");

emp1.salary = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("First Name:" + emp1.firstName);

Console.WriteLine("Last Name:" + emp1.lastName);

Console.WriteLine("Position:" + emp1.position);

Console.WriteLine("Salary:" + emp1.salary);

}

}

STATIC CONSTRUCTOR

A static constructor is used to initialize the static members of a class.

It is called only once, before the first instance of the class is created or any static members are accessed.

It is specified using the static keyword and does not take any parameters.

Example: static MyClass() { }

using System;

namespace Constructor {

class Car {

// static constructor

static Car () {

Console.WriteLine("Static Constructor");

}

// parameterless constructor

Car() {

Console.WriteLine("Default Constructor");

}

static void Main(string[] args) {

// call parameterless constructor

Car car1 = new Car();

// call parameterless constructor again

Car car2 = new Car();

Console.ReadLine();

}

}

}