C# - What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

* OOP is faster and easier to execute
* OOP provides a clear structure for the programs
* OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Classes and objects are the two main aspects of object-oriented programming.

Everything in C# is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

An object is created from a class.

field- variable inside a class is field.

method defines how an object behaves.

public class Name

{

string name = "raj";// field

public string myname(string names="sparsh") // method

{

return names;

}

}

internal class Program

{

static void Main(string[] args)

{

// creating object

Name n = new Name();

// accessing field

n.nam = "hello";

// calling method of class

string nm = n.myname();

string nm2 = n.myname("rani");

Console.WriteLine("the value from method in class is "+nm);

Console.WriteLine("the value from method in class is " + nm2);

Console.ReadLine();

}

}

What is a Constructor?

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.

public class Fruits

{

public string fname;

public string fre2;

public Fruits(string frs, string f2) {

fname = frs;

fre2 = f2;

}

public static void Main(string[] args) {

Fruits f=new Fruits("apple","pineapple");

Console.WriteLine("fruits "+f.fname+" "+ f.fre2);

Console.ReadLine();

}

}

Access Modifiers

Access modifier is used to set the access level/visibility for classes, fields, methods and properties.

public The code is accessible for all classes

private The code is only accessible within the same class

protected The code is accessible within the same class, or in a class that is inherited from that class

internal The code is only accessible within its own assembly, but not from another assembly.

protected internal The code is accessible within its own assembly and in the derived classes.

Encapsulation

Encapsulation is to make sure that "sensitive" data is hidden from users. This can be achieved by

* declare fields/variables as private
* provide public get and set methods, through properties, to access and update the value of a private field

A property is like a combination of a variable and a method, and it has two methods: a get and a set method:

Get is to retrieve the value

Set is set the value

Why encapsulation?

* Better control of class members (reduce the possibility of yourself (or others) to mess up the code)
* Fields can be made read-only (if you only use the get method), or write-only (if you only use the set method)
* Flexible: the programmer can change one part of the code without affecting other parts
* Increased security of data.

public class Fruits

{

private string frname; // field

public string FruitName // property

{

get { return frname; }

set { frname = value; }

}

// automatic property

public string fname { get; set; }

public static void Main(string[] args)

{

Fruits f = new Fruits();

f.frname = "orange";

f.fname = "apple";

Console.WriteLine(f.frname);

Console.WriteLine(f.fname);

Console.ReadLine();

}

}

Inheritance

In C#, it is possible to inherit fields and methods from one class to another.

Derived Class (child) - the class that inherits from another class

Base Class (parent) - the class being inherited from

To inherit from a class, use the : symbol.

public class Fruit

{

public void declare()

{

Console.WriteLine("this is fruits");

}

}

public class Apple : Fruit

{

public string frt = "apple";

}

class Program

{

static void Main(string[] args)

{

Apple apple = new Apple();

apple.declare();

Console.WriteLine("this is "+apple.frt);

}

}

If you don't want other classes to inherit from a class, use the sealed keyword

sealed public class Fruit

Types of Inheritance:

1. Single Inheritance

2. Multi-Level Inheritance

3. Hierarchical Inheritance

4. Multiple Inheritance- we cannot achieve multiple inheritance through classes we use interface for this.

Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

This allows us to perform a single action in different ways.

Compile time polymorphism:

• It is method overloading technique.

• Defining multiple methods with same name and different signature(parameters).

• Parameters can be either different length or different type.

• Overloading belongs to single class(object).

Runtime polymorphism:

• Runtime Polymorphism is a Method overriding technique.

• Defining a method in the Child class with the same name and same signature of its

Parent class.

• We can implement Method overriding only in Parent-Child (Is-A) relation

The virtual keyword is used to modify a method, property, indexer, or event declaration and allow it to be overridden in derived classes.

The override keyword is used to explicitly indicate that a method, property, indexer, or event in a derived class is overriding a virtual or abstract member from a base class. It is used alongside the virtual or abstract keyword in the base class.

public class Fruit// parent class

{

public virtual void declare() // method without parameter

{

Console.WriteLine("this is fruits");

}

public virtual void declare(string name) // with parameter

{

Console.WriteLine("this is from string "+name);

}

}

public class Apple : Fruit // child class

{

public string frt = "apple";

public override void declare()

{

Console.WriteLine("this is apple");

}

}

class Program

{

static void Main(string[] args)

{

Fruit fruit = new Fruit();

Apple apple = new Apple();

// method overriding

fruit.declare();

apple.declare();

// method overloading

fruit.declare("Hello");

}

}

Abstraction

Data abstraction is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either abstract classes or interfaces.

The abstract keyword is used for classes and methods:

* Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

public abstract class Fruits // abstract class

{

public abstract void nameof(string fruit); // abstract method with no body

public void nonab() // non abstract method with body

{

Console.WriteLine("this is non abstract class");

}

public virtual void nometh()

{

Console.WriteLine("this is parent");

}

}

public class Apple : Fruits

{

public override void nameof(string fruit) { // body for parent abstract method

Console.WriteLine("the fruit is apple->"+fruit);

}

public override void nometh()

{

base.nometh();

}

}

internal class Program

{

static void Main(string[] args)

{

Apple apple = new Apple();

apple.nameof("its apple"); // calling the child override method

apple.nonab(); // calling parent non abstract method from child object

apple.nometh(); // base method assigned in child

}

}

Interface

An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies).

* Like abstract classes, interfaces cannot be used to create objects.
* Interface methods do not have a body - the body is provided by the "implement" class.
* On implementation of an interface, you must override all of its methods.
* Interfaces can contain properties and methods, but not fields/variables.
* Interface members are by default abstract and public.
* An interface cannot contain a constructor.

We use interface to achieve multiple inheritance (class name: interface1, interface2) and for security purpose (hide certain details and only show the important details of an object)

public interface IFruits

{

void nameof();

void colorof();

}

public class Apple:IFruits {

public void nameof()

{

Console.WriteLine("this is fruits interface");

}

public void colorof()

{

Console.WriteLine("this is second method of interface");

}

// need to implement all the method from the interface class

}

internal class Program

{

static void Main(string[] args)

{

IFruits fruits = new Apple(); // creating fruits object from apple we cannot create obj from interface

fruits.colorof();

fruits.nameof();

}

}

Enums

An enum is a special "class" that represents a group of constants (unchangeable/read-only variables).

internal class Program

{

enum Fruits

{

Apple,

Orange,

Banana

}

static void Main(string[] args)

{

Fruits fruits = Fruits.Apple;

Console.WriteLine(fruits.ToString());

}

}