Object Oriented Programming using C#.NET

- Object-oriented programming is a way of developing software applications using real-world terminologies to create entities (classes) that interact with one another using objects.

- Object-oriented programming makes applications flexible (easy to change or add new features), reusable, well-structured, and easy to debug and test.

- Terms

    - Object

    - Class

    - Abstraction

    - Encapsulation

    - Inheritance

- Object-oriented Design Principles

- There are various object-oriented principles and techniques using which you can develop applications that are maintainable and extendable.

- The followings are four main principles of object-oriented programming:

- Abstraction

- Encapsulation

- Inheritance

- Polymorphism

- A thing in a real world that can be either physical or conceptual. An object in object-oriented programming can be physical or conceptual.

- Object has state and behavior.

- State represents physical looking of an object.

- The object's state is determined by the value of its properties or attributes.

- Properties or attributes →member **variables** or data members

- The object's behaviour is determined by the operations that it provides.

- Operations → member functions or **methods**

Example: Marker

State – solid, colors, length, price etc

Behavior – to write something

object

A bulb:

1. It's a real-world thing.

methods

2. Can be switched on to generate light and switched off.

3. It has real features like the glass covering, filament and holder.

4. It also has conceptual features like power.

member variables

5. A bulb manufacturing factory produces many bulbs based on a basic description / pattern of what a bulb is.

class

- Abstraction is the process of taking only a set of essential characteristics from something(object).

- Example

- For a Doctor→you are a Patient

- Name, Age, Old medical records

- For a Teacher→you are a Student

- Name, Roll Number/RegNo, Education background

- For HR Staff→you are _____

- A class is a construct created in object-oriented programming languages that enables creation of objects.

- Also, a class is sometimes called blueprint or template or prototype from which objects are created.

- It defines members (variables and methods).

- A class is an abstraction.

- Class is a collection of class members.

- Class in C# is created using the class keyword.

```
class YourClassNameHere
{
    //your class members
}
```

➢ The variables and functions defined inside a class are called members of the class.

Types of members

➢ Data members/ Fields

➢ Methods

➢ Constructors

➢ Destructors

➢ Properties

➢ Indexer

```
class YourClassNameHere
{
    //your data members here
    //your methods
}
```

```
class Customer{

public string Name;

public uint CustId;

public string Address;

public void Display(){

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

System.Console.WriteLine("ID:"+CustId);

System.Console.WriteLine("Address:"+Address);

}}
```

public makes the member accessible from everywhere

Data members

→ Method or Member function

Accessing members within the method of the same class

Responsibility of this class is the integrity of details of customer that it encompasses.

Creating objects and accessing members

- To create a Customer object, the **new** keyword is used.

> Class_Name Object_Name=new Class_Name();

```
class Test{

public static void Main(string[] a)      {

    Customer C1= new Customer();──→Customer object created

    C1.custId=10;

    C1.name="John";

    C1.address="H-234, Jayadev Apts, BTM, Banglore";

    C1.Display(); ──→Accessing member using . operator

}}
```
Note:Member of a class are accessed using . operator.
Convention: The method names are verbs. Field names are nouns. Field names must use Pascal naming convention

- Scope of variable can be largely divided into 2 categories

- Local

  - Declarations are made inside a method; accessible only inside the method

  - Local scoped variables must be initialized before use

- Class

  - Declarations are made outside a method and inside a class; accessible only inside the class

  - Class scoped variables are automatically initialized to their default values.

  - 2 types

    - Instance variables(default)

    - Static variables

Member variables are automatically assigned a default value.

```
bool→ false

Integer types→0

Floating point types→0.0

char →'\0'

string and references→ null
```

Member Visibility

- **public**
  - Accessible from anywhere
- **private**
  - Accessible only from within a class
  - Unmarked members are private by default
- **internal**
  - Accessible only within classes in the same assembly
- **protected**
- **protected internal**

Applicable when inheriting.

Top-level Class Visibility

(including interfaces, structures, enumerations and delegates which form namespace members
public
internal → default

- It is a function defined in the class.

- It is used to do some work.

- It is also called as Function / Member Function / Operation.

- It contains some executable code.

- It is a collection of statements (known as instructions).

- A method should have a name and a return type.

  - Return type specifies what type of data that the method returns. If you don't want to return any value, specify "void".

- Methods can access the data members, present with in the same class; and also manipulate them.

Syntax:

```
class YourClassNameHere
{

    YourReturnTypeHere  YourMethodNameHere( )
    {

        //your code here

    }

}
```

Example:

```
class Employee
{

    void Transfer( )
    {

            //your code for trasfering the employee here

    }

}
```

- The members that we have seen so far are instance members.

- Members cannot be accessed using instance.

- **`WriteLine()`** is a static method of the **`System.Console`** class.

- **`Main`** is declared as a static method.

- Static data is shared by all the instances of that class.

- Static member functions can access only static data members.

- The **`static`** keyword cannot be used for local variables.

Syntax:

```
class YourClassNameHere
{
Static datatype var_name;
Static void call()
{
}
}
```

Example:

```
class Employee
{
    static void Transfer( )
    {
        //your code for trasfering the employee here
    }
}
```

- It is the built-in feature of OOPS.
- The concept of grouping-up the data members and methods together, is called as "data encapsulation".
- Encapsulation is defined 'as the process of enclosing one or more items within a logical package'. Encapsulation, in object oriented programming methodology, prevents access to implementation details.
- Abstraction and encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible
- Class is the real implementation of "Data Encapsulation".

```
class Contact
{
    //some data members
        public  string  ContactName;
        public  string  Email;

    //some function members
        public  void  SendEmail()
        {

        }
}
```

Method Parameter modifiers

- By default, the parameters are passed by value.

- Parameter modifiers can be used to alter this behaviour.

- Parameter modifiers that alter the default behaviour :

  - **ref**

  - **out**

- These modifiers do not create any storage location in the method call.

- Properties, indexers (*we will see later*) or dynamic member cannot be passed as an **out** or **ref** parameter.

- Another modifier that could be used with the parameters is **params** that is used to pass any number of arguments.

- The **ref** keyword makes the parameter passing to be done by reference.

- The **ref** keyword is specified both in the method definition and while calling.

- **ref** requires that variable to be initialized before the call.

```
using System;
class OutParam{
public static void cal(int i,int j,   ref int k){
k=i+j;
}
public static void Main(){
int i=10,j=20,k=0;
cal(i,j,ref k);
Console.WriteLine(k);// prints 30
}}
```

- **out** is similar to **ref**, except that the initial value of the argument provided by the calling function is not important. Values of it cannot be requested in the method unless its value is assigned in the method.

- Also the out parameter must have a valid value before the function exits.

```
using System;
class OutParam{
public static void cal(int i,int j,   out int k){
k=i+j;
}
public static void Main(){
int i=10,j=20,k;
cal(i,j,out k);
Console.WriteLine(k);// prints 30
}}
```

- Sending any number of arguments of a particular type.

- There can be only one **params** for any method.

- The **params** argument must be the last parameter specified.

- **params** should be a single dimensional or a jagged array.

```csharp
using System;
class Params{
static int sum(params int[] i){
int sum=0;
for(int k=0;k<i.Length;k++) sum+=i[k];
return sum; }
public static void Main(){
int s=sum(1,2,3,4);
Console.WriteLine(s);
s=sum(11,22);
Console.WriteLine(s); }}
```

- Methods with the same name but different signature are called overloaded methods.

- Methods may differ in terms of

  - Number of parameters

  - Types of parameter

  - Order of parameter

- Note that if two methods are identical except for their return types or access specifiers, then the methods are not overloaded.

- Overloading works same for `class` and `struct`

```
using System;
class Overload1{
static void add(int a, int b){
int c=0;
c=a+b;
Console.WriteLine(c);}
static void add(int a, int b, int c){
    int x=0;
    x=a+b+c;
    Console.WriteLine(x);
}
static void add(double a, double b){
    double c=0;
    c=a+b;
    Console.WriteLine(c);
}
public static void Main(){
    add(1,2);
    add(1,2,3);
    add(1.2,1.3);
}
}
```

Output:

3
6
2.5

- A special method used to initialize members when object is constructed.

- Constructor is called automatically on creation of object using `new` .

- Name of the constructor is same as the name of the class.

- A constructor does not have return type. It can have any access specifers.

- Types:

  - A constructor that takes no parameters is called a default constructor

  - Parameterized constructor

- A class can have any number of constructors.

- Classes (non-static) without constructors are given a public default constructor by the C# compiler in order to enable class instantiation.

```
using System;
class Point{
private int x,y;
Point(int x1,int y1){
x=x1;
y=y1;
}
Point(){
x=0;
y=0;
}
static void Main(){
Point p1= new Point(10,20);
Point p2= new Point();
}}
```

➢ The this keyword refers to the current instance of the class.

➢ It can be used to access members from within constructors, instance methods, and instance accessors.

➢ Point(int x,int y){

➢ this.x=x;

➢ this.y=y;}

➢ It can be used to pass the current object as a parameter to a method.

➢ call(this);

➢ this is also used for constructor chaining and to declare indexers.

```
public void MyMethod()
{
      this.DataMember1
      this.DataMember2
      this.fun1();
}
```

- Like constructors are used to initialize instance fields, static constructors are created to initialize static fields.

- But unlike regular constructors, static constructor

  - cannot have arguments

  - cannot have any access modifier

  - can be single only

- A static constructor executes before any other constructor.

- It gets called just before any of the class member (static or instance or constructor) is invoked.

- It gets called only once.

- Constant members can be created using the `const` keyword.

- Members of the `const` type must be initialized during compile time.

- Constants are implicitly `static`. Therefore, they are accessed using class name only.

- **class Circle{**

  public const double PI=3.14;

  …

  }

- Accessing outside the class: **Circl.PI**

- Read-only fields are assigned value only once either during compile time or runtime.

- They must be initialized either with the declaration or in the constructor.

- The keyword `readonly` is used for this.

- Unlike constants, read-only fields are instance members and not static members.

- Note that the `readonly` keyword is a modifier that can be used on fields.

- Properties are named members of classes (structs, and interfaces) that provide a flexible mechanism to read, write, or compute and validate the values of private/public fields through accessors.

- Syntax:

```
Access-specifier type name {

get{}

set{}

}
```

- The properties are accessed using their names.

```
using System;
public class Employee{
        private uint empID=111111;
        private string empName;
    public string Name{
        get{return empName;}
        set{ if(value!=null) empName=value;
            else Console.WriteLine("invalid name");}
        }
    public uint ID {
            get { return empID; }
            set { if (value != 0) empID = value;
        else Console.WriteLine("invalid ID"); }
    }
      public static void Main(string[] a)    {
       Employee e=new Employee();
       e.Name = "Raj";  e.ID = 0;
       Console.WriteLine("{0:d} {1:s}",e.empID, e.Name);
}}
```

```csharp
class Circle{

private static double PI;

public static double pi{

get{return PI;}

set{ PI=value;}

}

static void Main(){

Circle c= new Circle(4);

Circle.pi=3;

System.Console.WriteLine(Circle.pi);

}
…}
```

- The struct in C# is closer to C++ struct.

- Like a C# class, struct can also have members which are constructors, constants, fields, methods, properties, indexers, operators, events, and nested types.

- Recommended to be used for smaller data structure

```csharp
using System;
struct Box{
  public int width;
  public int  height;
  public Box(int w, int h) {
       width = w;
       height = h;    }
static void Main() {
       Box b = new Box(12, 13);
       Console.WriteLine(b.height);
       Console.WriteLine(b.width); }}
```

```
using System;
struct Box {
  /* public int width=10;
     public int  height=10; */
   public static int width=10;
   public static int  height=10;

   public Box(int w, int h)     {
       width = w;
       height = h;
   }

  /* public Box()     {
       width = 1;
       height = 1;
   } */
```

Error

Error

```
static void Main()
    {
        Box b = new Box(12, 13);
        Console.WriteLine(b.height);
        Console.WriteLine(b.width);
        Box b1; //constructor is not invoked
        b1.width = 10;
        Console.WriteLine(b1.width);
}
}
```

`struct`

- Are value types
- No default constructors
- No initializers
- No destructors
- Can be instantiated with or without the **new** operator
- Cannot be involved in inheritance

`class`

- Are Reference types.
- Can have default constructors
- Can have initializers
- Can have dsetructors
- Cannot be instantiated without the **new** operator
- Can be involved in inheritance

All Primitive types are predefined structures
Ex: int,float,double,char etc.