OBJECT ORIENTED PROGRAMMING

OOP IN C#

JANUARY 1, 2025

YOUSEF IBRAHEM EISSA

Content

Object-Oriented Programming (OOP)

OOP Definition:-

Object-Oriented Programming (OOP) Is a Programming Paradigm That Allows You To Package Together Data States And Functionality.

البرمجة الموجهة للكائنات هي توجه برمجي يقوم على تجميع الخصائص و الأفعال المترابطة ضمن قالب واحد.

Class&Object

Class Syntax :-

```
// <ClassModifier> class <ClassName>
// {
//     Class Block
// }
// <ClassModifier>  → public , internal (default)
```

EX:

```
Internal class Employee
{
        public const var Tax = 0.03d;
        public string fName;
        public string lName;
}
```

Object Syntax :-

- **Declaration**
  ```
  // <Type> <ObjectName> ;
  ```

- **Assignment**

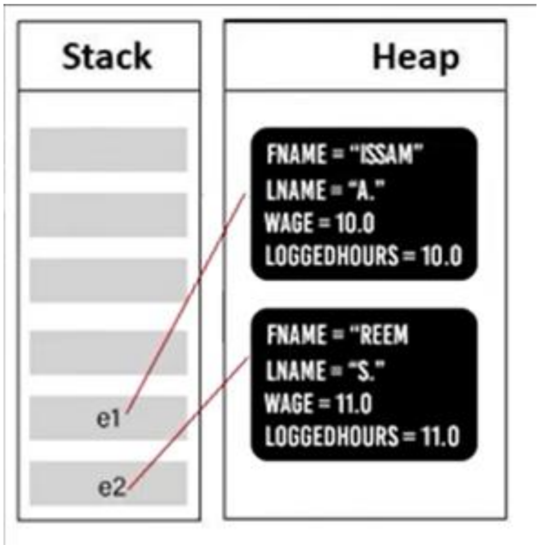
// <ObjectName> = new <Type> () ;

- **Initialization**

  // <Type> <ObjectName> = new <Type> () ;

كيف يخزن الـ Object في الـ Memory ؟

- **The Object Name → Stack**
- **The Object Value → Heep**



Class Members

1- **Fields**

Field Syntax :-

// <AccessModifier> <DataType> <FieldName> = <InitialValue>;
// <AccessModifier> → public , private , protected

EX:

public string fName = "Ali";

Notes :-

- اذا كان الـ Field من نوع public يبدء الاسم بحرف Capital .
- اذ كان الـ Field من نوع private يبدء الاسم بحرف Small أو _ .
- يفضل دائما ان يكون الـ field من نوع private و التعامل معه يكون من خلال الـ Constractor .

EX:

```
public int DayInMonth ;
private int dayInMonth ;
private int _dayInMonth;
```

2- Constants

Constant Syntax :-

```
// <AccessModifier>  const  <DataType>  <ConstantName>  =  <Value>;
```

EX:

```
public const var Tax = 0.03d;
```

Q:- What is Different Between Constant Mamber And Static Member ?

- Constant Mamber Is A Promise That Can Not Be Changed After It Has Been Initialized.
- Static Member Is A Shared Variable That Can Be Changed After It Has Been Initialized.

Static Syntax:-

```
// <AccessModifier>  static  <DataType>  <StaticName>  =  <Value>;
```

EX:

```
public static var Tax = 0.03d;
```

Note:-

Constant And Static Calling By Class Name Not Object Name.

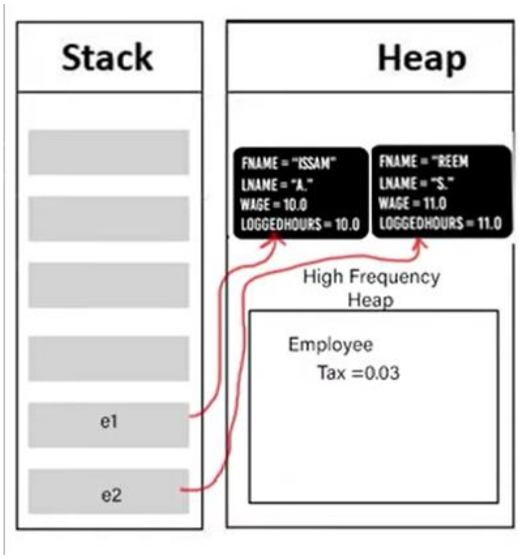
```
// <ClassName>.<ConstantName>
```

But Field Calling By Object Name

```
// <ObjectName>.<FieldName>
```

أين يتم تخزين كلا من الـ Constant والـ Static داخل الذاكرة ؟

- **يتم تخزين كلا من الـ Constant والـ Static داخل منطقة تقع في الـ Heap تسمى High Frequence Heap و يرتبط باسم الـ Class مباشرة لانها قيمة مشتركة بين الجميع كما في المخطط التالى.**



3- Methods

Method Syntax :-

```
// <AccessModifier>  <DataType>/void  <MethodName> (<Prameter List>)
// {
//       Series of Statement
// }
```

EX:

```
public void DeSomething(int age)
{
        age = age + 3 ;
}
```

Method Calling :-

```
// <ClassName> <ObjectName> = new <ClassName> () ;
// <ObjectName>.<MethodName> (<The Arguments>);
```

EX:

```
Int age = 10;
Demo d1 = new Demo();
d1. DeSomething(age);
```

Notes:

- **Instance Method Is Called By Object.**
- **Static Method Is Called By TypeName (ClassName).**
- **يبدء اسم الـ Parameter دائما بـ Small Letter .**

Q:- What Is Different Between The Parameter And The Argument ?

- A parameter is a variable defined in the function or method's declaration.
- An argument is the actual value or data passed to the function when it is called.

Q:- What Is Different Between Ref And Out Keyword ?

The ref and out keywords in C# are used to pass arguments by reference to a method. While they share similarities, there are important differences between the two:

- The ref keyword allows a method to modify the value of an argument and requires that the variable be initialized before passing it to the method.

EX:

```
class Program

{

        static void Main (string[] args)

        {

                var age = 18;

                Demo d1 = new Demo ();

                d1.Dosomething (ref age);

                Console.WriteLine (age);     //21

        }

}

public class Demo {

        public void Dosomething (ref int age) {

                age = age + 3;

        }
}
```

- The out keyword is used to return multiple values from a method. It does not require the variable to be initialized before being passed, but the method must assign a value to the out parameter before it returns.

EX:

```
class Program
{
    static void Main (string[] args)
    {
        var age;
        Demo d1 = new Demo ();
        d1.Dosomething (out age);
        Console.WriteLine (age);    //21

    }
}

public class Demo {
    public void Dosomething (out int age) {
        age = 18;
        age = age + 3;
    }
}
```

Method Signature :-

```
// Method Signature ( neme + parameter type + parameter order )
```

EX1:

```
public void Dosomething ( int x , double y )
{
}
public void Dosomething ( double y , int x )
{
}
```

This Two Methods Are Different Methods Because The Parameters Order Are Different .

EX2:

```
public void Dosomething ( int x , double y )
{
}
public void Dosomething ( string x , double y )
{
}
```

This Two Methods Are Different Methods Because The Parameters Type Are Different .

Note:-

This Action Are Called Method Overloading .

Q:- What Is Method Overloading ?

Method overloading is a concept in programming where multiple methods in the same class share the same name but differ in the number or type or order of their parameters. It allows a class to provide multiple ways to perform a similar operation based on different inputs.

Note:-

Method Overloading Is a Common Way Of Implementing Polymorphism .

Expression-Bodied Methods :-

Expression-bodied methods are a shorthand syntax used to define methods that have a single expression. This style is concise and improves readability for simple methods, especially when the method logic is straightforward.

Expression-Bodied Methods Syntax :-

```
// <AccessModifier>  <Datatype>/void <MethodName> (<Parameters>)
                                                    => <Returned Value> ;
```

Ex:

- **The Usual Method :-**

```
public bool IsEven (int number) {
        return  number % 2 == 0;
}
```

- **The Expression-Bodied Methods :-**

```
public bool IsEven (int number) => return  number % 2 == 0;
```

Local Method :-

Local methods are methods declared inside the body of another method, constructor, or property. They are scoped to the enclosing method or block, making them accessible only within that context.

EX:

```
public void PrintEvens (int[] original) {
        foreach ( var n in original ) {
                if ( IsEven (n) ) {
                        Console.Writeline (n) ;
                }
        }
        bool Is IsEven (int number) => number % 2 == 0 ;
}
// IsEven Is Local Method To PrintEvens.
```

Static Method :-

A static method in programming is a method that belongs to a class rather than any specific instance of the class. It can be called on the class itself without creating an object (instance) of the class.

Static Method Syntax :-

```
// <AccessModifier> static  <Datatype>/void <MethodName>
(<Parameters>)  {
//      Series of Statement
// }
```

EX:

```
Public static void PrintEvens (int[] original) {
        foreach ( var n in original ) {
                if ( IsEven (n) ) {
                        Console.Writeline (n) ;
                }
        }
        bool Is IsEven (int number) => number % 2 == 0 ;
}
```

4- Constructors

A constructor is a special method in object-oriented programming that is used to initialize objects of a class. It is automatically invoked when an object of the class is created.

Note:-

The Purpose Of The Constructor Is Seting The Initial Values For Object Attributes.

Constructor Syntax :-

```
// <AccessModifier>  <TypeName> (<Parameters List>)
// {
//        series of statement
// }
```

EX:-

```
public class Date
{
    private int Day;
    private int Month;
    private int Year;

    public Date (int day, int month, int year)
    {
        Day = day;
        Month = month;
        Year = year;
    }
}
```

Notes:-

- عند انشاء class جديد تقوم الدوت نت بانشاء constructor تلقائي لهذا الـ class و هو ما يعرف باسم الـ Implicit Constructor .
- اذا كان الـ constructor من نوع private لا يمكن التعامل مع الـ class الا من خلال static method .
- اسناد القيم الى الـ field التابع للـ class يتم بطريقتين :-
  1- عن طريق اسناد القيمة للـ field مباشرة

  ```
  d1.Day = 10;
  ```

  2- عن طريق الـ constructor

  ```
  Date d1 = new Day (10, 04, 2002);
  ```

- ReadOnly Field Can Be Changed Inside The Constructor Only.

Constructor OverLoading :-

EX:-

```
public class Data
{
    private readonly int day;
    private readonly int month;
    private readonly int year;
```

```
        public Date (int day, int month, int year)
        {
                this.day = day;
                this.month = month;
                this.year = year;
        }
        public Date (int month, int year) : this (01, month, year)
        {
        }
        public Date (int year) : this (01, 01, year)
        {
        }
}
```

Object Initializer :-

Object Initializer Syntax :-

```
// <ClassName>  <ObjectName> = new <ClassName> {
//      <prop1> = <value>,
//      <prop2> = <value>,
//      ....
//      <propN> = <value>
//   };
```

EX:-

```
public class Employee {
        public int Id;
        public string Name;
        public string Phone;
}
class Program
{
    static void Main (string[] args)
    {
        Employee e1 = new Employee {
            Id = 100,
            Name = “Ali”,
            Phone = “01095388071”
        };
    }
}
```

5- Properties

Property Is a Public Way To Access Private Field .

Note :-

Properties Promote Encapsulation.

Property Syntax :-

```
// <AccessModifier> <FieldDataType> <PropertyName>
// {
//     get {
//         series of statement
//     }
//     set {
//         series of statement
//      }
// }
```

EX :-

```
public class Dollar
{
        private decimal  _amount ;
        public decimal  Amount
        {
                get {
                        return this._amount;
                }
                set {
                        this._amount = value;
                }
        }
}
class Program
{
        static void Main (string[] args)
        {
                Dollar dollar = new Dollar ();
                dollar.Amount = 1.99m;      // set
                Console.WriteLine(dollar.Amount);     // get
        }
}
```

Note :-

_amount is called Backing Field to Amount.

Read-Only Property :-

EX:-

```
public class Dollar
{
    private decimal  _amount ;
    public decimal  Amount
    {
        get {
            return this._amount;
        }
    }
}
```

Write-Only Property :-

EX :-

```
public class Dollar
{
    private decimal  _amount ;
    public decimal  Amount
    {
        set {
            this._amount = value;
        }
    }
}
```

Expression-Bodied Property :-

EX :-

```
public class Dollar
{
        private decimal  _amount ;
        public decimal  Amount  =>  this._amount;    // readonly Property
}
```

Auto-Implemented Property (Simplified Syntax) :-

EX :-

```
public class Dollar
{
        private decimal  _amount ;
        public decimal  Amount  { get; set; }
}
```

6- Indexers

An indexer is a special property that allows an object to be indexed like an array. It provides a way to access elements within a class or structure using square brackets ([]).

Indexer Syntax :-

```
// <AccessModifier> <DataType> this [<DataType> index]
// {
//       get {
//       }
//      set {
//      }
// {
```

EX :-

```
public class Ip {
        private int[] segments = new int[4];
        public int this [int index]
        {
                get {
                        return segments[index];
                }
                set {
                        segments[index] = value;
                }
        }
}
class Program {
        public void Main (string[] args) {
                var ip = new Ip (114, 117, 55, 33);
                var fristSegment = ip[0];
                Console.WriteLine(fristSegment);        // 114
        }
}
```

EX2 :-

```
public class Suduko {
        private int[,] _matrix ;
        public int this [int row, int col]
```

```
    {
        get {
            return _matrix [row, col];
        }
        set {
            _matrix [row, col] = value;
        }
    }
    public Suduko (int[,] matrix) {
        _matrix = matrix;
    }
}
class Program {
    public void Main (string[] args) {
        int [,] inputs = new int[,] {
            {8, 3, 5, 4, 1, 6, 9, 2, 7},
            {2, 9, 6, 8, 5, 7, 4, 3, 1},
            {4, 1, 7, 2, 9, 3, 6, 5, 8},
            {5, 6, 9, 1, 3, 4, 7, 8, 2},
            {1, 2, 3, 6, 7, 8, 5, 4, 9},
            {7, 4, 8, 5, 2, 9, 1, 6, 3},
            {6, 5, 2, 7, 8, 1, 3, 9, 4},
            {9, 8, 1, 3, 4, 5, 2, 7, 6},
            {3, 7, 4, 9, 6, 2, 8, 1, 5}
        }
        var suduko = new Suduko(inputs);
        Console.WriteLine(suduko[5, 5]);        // 9
    }
```

```
}
```

7- **Delegate**

- **Delegate Object that Points to Method AI Changed at Run Time.**
- **Delegate is a Special Method Without Body.**
- **Delegate is used to sent condition or function as a argument.**

Delegate Syntax :-

```
// <AccessModifier> delegate <DataType> <Name> (<Parameters>);
```

Note :-

Delegate is a Reference Type.

EX Without Delegate :-

```
public class Employee {
        public int Id { get; set; };
        public string Name { get; set; };
        public decimal TotalSales { get; set; };
        public string Gender { get; set; };
}

public class Report {
public viod ProcessEmployeeWith60000PlusSales(Employee[] employees) {
                Console.WriteLine(“Employees With $60,000+ Sales”);
                Console.WriteLine(“~~~~~~~~~~~~~~~~~~~~~~~~~~~~~”);
                Foreach(var e in employees){
                        If(e. TotalSales >= 60000m)
```

```
                Console.WriteLine(e.Name);
        }
    }
public viod
ProcessEmployeeWithSalesBetween30000and59999(Employee[]
employees) {
Console.WriteLine(“Employees With Sales Between 30000 and 59999”);
        Console.WriteLine(“~~~~~~~~~~~~~~~~~~~~~~~~~~~~”);
        Foreach(var e in employees){
            If(e. TotalSales < 60000m && e. TotalSales >= 30000m)
                Console.WriteLine(e.Name);
        }
    }
public viod ProcessEmployeeWithSalesLessThan30000(Employee[]
employees) {
        Console.WriteLine(“Employees With Sales Less Than 30000”);
        Console.WriteLine(“~~~~~~~~~~~~~~~~~~~~~~~~~~~~”);
        Foreach(var e in employees){
            If(e. TotalSales < 30000m)
                Console.WriteLine(e.Name);
        }
    }
}

public class Program{
    public void Main (string[] args) {
        var emps = new Employee[]
        {
```

```
                new Employee { Id = 1, Name = “Issam A”,
                                Gender = “M”, TotalSales = 65000m },
                new Employee { Id = 2, Name = “Reem S”,
                                Gender = “F”, TotalSales = 50000m },
                new Employee { Id = 3, Name = “Suzan B”,
                                Gender = “F”, TotalSales = 65000m },
                new Employee { Id = 4, Name = “Sara A”,
                                Gender = “F”, TotalSales = 40000m },
                new Employee { Id = 5, Name = “Salah C”,
                                Gender = “M”, TotalSales = 42000m },
                new Employee { Id = 6, Name = “Rateb A”,
                                Gender = “M”, TotalSales = 30000m },
                new Employee { Id = 7, Name = “Abeer C”,
                                Gender = “F”, TotalSales = 16000m },
                new Employee { Id = 8, Name = “Marwan M”,
                                Gender = “M”, TotalSales = 15000m },
            }
        var report = new Report();
        report.ProcessEmployeeWith60000PlusSales(emps);
        report.ProcessEmployeeWithSalesBetween30000and59999(emps);
        report.ProcessEmployeeWithSalesLessThan30000 (emps);
    }
}
```

The Same EX With Delegate :-

```
public class Employee {
    public int Id { get; set; };
    public string Name { get; set; };
    public decimal TotalSales { get; set; };
    public string Gender { get; set; };
}

public class Report {
    public delegate bool IllegibleSales (Employee e);
    public viod ProcessEmployee (Employee[] employees, string title ,
    IllegibleSales isIllegible)
    {
        Console.WriteLine(title);
        Console.WriteLine("~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        Foreach( isIllegible(e) )
        {
            If(e. TotalSales >= 60000m)
                Console.WriteLine(e.Name);
        }
    }
}

public class Program{
    public void Main (string[] args) {
        var emps = new Employee[]
        {
            new Employee { Id = 1, Name = "Issam A",
                           Gender = "M", TotalSales = 65000m },
            new Employee { Id = 2, Name = "Reem S",
                           Gender = "F", TotalSales = 50000m },
            new Employee { Id = 3, Name = "Suzan B",
```

```
                                      Gender = “F”, TotalSales = 65000m },
                new Employee { Id = 4, Name = “Sara A”,
                                      Gender = “F”, TotalSales = 40000m },
                new Employee { Id = 5, Name = “Salah C”,
                                      Gender = “M”, TotalSales = 42000m },
                new Employee { Id = 6, Name = “Rateb A”,
                                      Gender = “M”, TotalSales = 30000m },
                new Employee { Id = 7, Name = “Abeer C”,
                                      Gender = “F”, TotalSales = 16000m },
                new Employee { Id = 8, Name = “Marwan M”,
                                      Gender = “M”, TotalSales = 15000m },
        }
 var report = new Report();
 report.ProcessEmployee(emps,
                                    “Sales >= $60,000”,
                                     IsGreaterThanOrEqual60000 );
 report.ProcessEmployee(emps,
                                    “$30,000 > Sales< $60,000”,
                                     IsBetween30000And59999 );
 report.ProcessEmployee(emps,
                                    “Sales < $30,000”,
                                    IsLessThan30000 );
}

static bool IsGreaterThanOrEqual60000(Employee e) {
        return e.TotalSales > 60000m;
}
static bool IsBetween30000And59999(Employee e) {
```

```
                return e.TotalSales >= 30000m &&  e.TotalSales < 60000m;
        }
        static bool IsLessThan30000(Employee e) {
                return e.TotalSales < 30000m;
        }
}
```

Annonymous Delegate :-

EX :-

```
public class Program{
        public void Main (string[] args) {
                var emps = new Employee[]
                {
                        new Employee { Id = 1, Name = “Issam A”,
                                                Gender = “M”, TotalSales = 65000m },
                        new Employee { Id = 2, Name = “Reem S”,
                                                Gender = “F”, TotalSales = 50000m },
                        new Employee { Id = 3, Name = “Suzan B”,
                                                Gender = “F”, TotalSales = 65000m },
                        new Employee { Id = 4, Name = “Sara A”,
                                                Gender = “F”, TotalSales = 40000m },
                        new Employee { Id = 5, Name = “Salah C”,
                                                Gender = “M”, TotalSales = 42000m },
                        new Employee { Id = 6, Name = “Rateb A”,
                                                Gender = “M”, TotalSales = 30000m },
                        new Employee { Id = 7, Name = “Abeer C”,
                        Gender = “F”, TotalSales = 16000m },
                new Employee { Id = 8, Name = “Marwan M”,
                        Gender = “M”, TotalSales = 15000m },
        }
     var report = new Report();
     report.ProcessEmployee(emps, “Sales >= $60,000”,
                delegate (Employee e) { return e.TotalSales > 60000m; } );
     report.ProcessEmployee(emps, “$30,000 > Sales< $60,000”,
                delegate (Employee e) return e.TotalSales >= 30000m
                                   &&  e.TotalSales < 60000m; } );
     report.ProcessEmployee(emps, “Sales < $30,000”,
               delegate (Employee e) return e.TotalSales < 30000m; } );
    }
}
```

Lambda Expression :-

EX :-

```
    var report = new Report();
     report.ProcessEmployee(emps, “Sales >= $60,000”,
                e => e.TotalSales > 60000m );
     report.ProcessEmployee(emps, “$30,000 > Sales< $60,000”,
                e => e.TotalSales >= 30000m &&  e.TotalSales < 60000m );
     report.ProcessEmployee(emps, “Sales < $30,000”,
                 e => e.TotalSales < 30000m );
```

Multicast Delegate :-

EX :-

```
public class RectangeHelper {
    public void GetArea (decimal width, decimal height)
    {
        var result = width * height ;
        Console.WriteLine(result);
    }
    public void GetPerimeter (decimal width, decimal height)
    {
        var result = 2 * (width + height) ;
        Console.WriteLine(result);
    }
}
public delegate void RectDelegate (decimal width, decimal height);
public class Program {
    public void Main(string[] args) {
        var helper = new RectangeHelper();
        RectDelegate rect;
        rect = helper. GetArea;
        rect += helper. GetPerimeter;
        rect(10, 10);
    }
}
```

8- Events

Events enable a class or object to notify other classes or objects when something of interest occurs.

Note:-

The DataType of events is Delegate.

Event Syntax :-

```
// <AccessModifier> event <DelegateName> <EventName>;
```

EX :-

```
public delegate void stockHandler(Stock stock, decimal oldPrice);
public class Stock {
        private string name;
        private decimal price;
        public event stockHandler OnPriceChanged;

        public string Name => this.name;
        public decimal Price { get => this.price;  set => this.price  = value; }

        public Stock(string stockName){
                this.name = stockName;
        }
        Public void ChangeStockPriceBy(decimal percent){
                Decimal oldPrice = this.Price;
                this.Price += Math.Round(this.Price * percent, 2);
                if(OnPriceChanged != null){
                        OnPriceChanged(this, oldPrice);
                }
        }
}
class Program {
        static void Main (string[] args){
```

```
        var stock = new Stock(“Amazon”);
        stock.Price = 100;
        stock.OnPriceChanged += Stock_OnPriceChanged;
        stock.ChangeStockPriceBy(0.05m);
        stock.ChangeStockPriceBy(-0.02m);
        stock.ChangeStockPriceBy(0.00m);
    }
    private static void Stock_OnPriceChanged (Stock stock,
                                  decimal oldPrice){
        if(stock.Price > oldPrice){
            Console.ForegroundColor = ConsoleColor.Green;
        }
        else if(stock.Price < oldPrice){
            Console.ForegroundColor = ConsoleColor.Red;
        }
        else {
            Console.ForegroundColor = ConsoleColor.Gray;
        }
        Console.WriteLine($” {stock.Name} : {stock.Price} ”);
    }
}
```

9- Operators Overloading

نستخدم الـ Operator Overloading للتمكن من عمل عمليات رياضية و منطقية على الـ User Defined Types مثل الـ Classes .

Note :-

عند عمل Operator Overloading لبعض العمليات المنطقية يجب عمل Operator Overloading للعمليات المعاكسة لها مثل : (<= , >=) , (< , >) , (== , !=).

EX :-

```
class Money {
        private decimal amount;
        public decimal Amount => amount;
        public Money {
                this.amount = Math.Round(value, 2);
        }
        public static Money operator +(Money m1, Money m2) =>
                                        new Money (m1.amount + m2.amount);
        public static Money operator -(Money m1, Money m2) =>
                                        new Money (m1.amount - m2.amount);
        public static Money operator *(Money m1, Money m2) =>
                                        new Money (m1.amount * m2.amount);
        public static Money operator /(Money m1, Money m2) =>
                                        new Money (m1.amount / m2.amount);
        public static Money operator ++(Money m) =>
                                                  new Money (++ m.amount);
        public static Money operator --(Money m) =>
                                                  new Money (-- m.amount);
        public static bool operator >(Money m1, Money m2) =>
                                                    m1.amount > m2.amount
        public static bool operator <(Money m1, Money m2) =>
                                                    m1.amount  < m2.amount
        public static bool operator >=(Money m1, Money m2) =>
                                                    m1.amount >= m2.amount
        public static bool operator <=(Money m1, Money m2) =>
```

```
                                                    m1.amount <= m2.amount
    public static bool operator ==(Money m1, Money m2) =>
                                                    m1.amount == m2.amount
    public static bool operator !=(Money m1, Money m2) =>
                                                    m1.amount != m2.amount
}
```

10- Finalizers (Desctructor)

Methods Inside The Class Used To Destroy Instances of that class .

Q :- When is the Desctructor called?

- **The destructor is called when an object is destroyed or goes out of scope.**

Desctructor Syntax :-

```
// ~<className> ( )
// {
//          Series of Statement
// }
```

EX :-

```
class Person {
    public string Name { get; set; }
    public Person () {
        Console.WriteLine(“This is Person Constructor”);
    }
    ~Person () {
        Console.WriteLine(“This is Person Desctructor”);
    }
}
```

Note :-

In C# We Use Garbage Collection Library To Reclaim Memory That Is No Longer In Use.

EX :-

```
class Program {
        static void Main (string[] args) {
                MakeSomeGarabge();
                Console.WriteLine($”Memory Used Before Collection
                                {GC.GetTotalMemory(false):N0}”);   // 95,264
                GC.Collect();
                Console.WriteLine($”Memory Used After Collection
                                {GC.GetTotalMemory(true):N0}”);   // 73,336
        }
        static void MakeSomeGarabge() {
                Version v;
                For(int I = 0; I < 1000; i++){
                        V = new Version();
                }
        }
}
```

11- Nested Types

Type Defined Within a Type or Class Defined Within a Class.

EX :-

```
class A {
        class B {
        }
}
```

Access Modifiers

1- Public
   - Accessible from any other code in the same assembly or another assembly.
   - No restrictions on accessibility.

2- Private
   - Accessible only within the class or struct where it is declared.
   - Most restrictive access modifier.

3- Protected
   - Accessible within the same class and by derived classes (even if in another assembly).

4- Internal
   - Accessible only within the same assembly.
   - Not accessible from another assembly.

5- Protected Internal
   - Accessible within the same assembly or by derived classes in another assembly.

6- Private Protected *(C# 7.2 and later)*
   - Accessible within the same class and by derived classes in the same assembly.
   - Combines the restrictions of protected and internal.

الخلاصة :-

الـ public في الـ solution كامل حتى لو مكون من اكثر من project ، الـ Internal في الـ project الواحد (كل عناصر الـ project) ، الـ Private فقد داخل الـ member المعرف به.

Note :-

In Nested Types: The Methods of Nested Class Implicitly Have Access To Private Members in The Container Class.

Error Types

1- Syntax Error :-
   - We can follow syntax error using error list window.

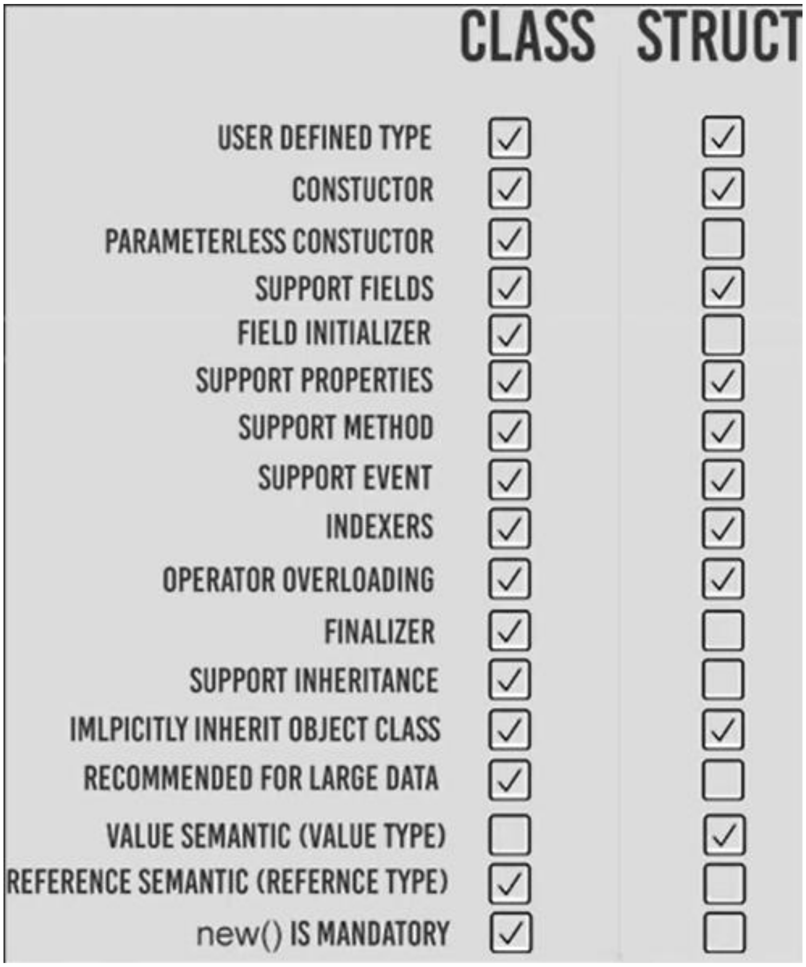
2- Runtime Error :-
   - We can be avoided runtime error by exceptions (Try-Catch-Finally Block).

3- Logical Error :-
   - We can follow logical error by traccing the code using debugging.

Struct

A Struct (short for "structure") is a user-defined data type that groups together variables under one name, Structs are commonly used to represent a collection of related data.

Q:- What Is Deffirent Between Class and Struct ?



	CLASS	STRUCT
USER DEFINED TYPE	☑	☑
CONSTUCTOR	☑	☑
PARAMETERLESS CONSTUCTOR	☑	□
SUPPORT FIELDS	☑	☑
FIELD INITIALIZER	☑	□
SUPPORT PROPERTIES	☑	☑
SUPPORT METHOD	☑	☑
SUPPORT EVENT	☑	☑
INDEXERS	☑	☑
OPERATOR OVERLOADING	☑	☑
FINALIZER	☑	□
SUPPORT INHERITANCE	☑	□
IMLPICITLY INHERIT OBJECT CLASS	☑	☑
RECOMMENDED FOR LARGE DATA	☑	□
VALUE SEMANTIC (VALUE TYPE)	□	☑
REFERENCE SEMANTIC (REFERNCE TYPE)	☑	□
new() IS MANDATORY	☑	□

EX :-

```
struct DigitalSize {
        private long bit;

        private const long bitsInBit = 1;
        private const long bitsInByte = 8;
        private const long bitsInKB = bitsInByte * 1024;
        private const long bitsInMB = bitsInKB * 1024;
        private const long bitsInGB = bitsInMB * 1024;
        private const long bitsInTB = bitsInGB * 1024;
```

```
    public string Bit => $” {(bit / bitsInBit):N0} Bit ”;
    public string Byte => $” {(bit / bitsInByte):N0} Byte ”;
    public string KB => $” {(bit / bitsInKB):N0} KB ”;
    public string MB => $” {(bit / bitsInMB):N0} MB ”;
    public string GB => $” {(bit / bitsInGB):N0} GB ”;
    public string TB => $” {(bit / bitsInTB):N0} TB ”;

    public DigitalSize(long initialValue) {
        this.bit = initialValue;
    }
}
class Program {
    static void Main(string[] args) {
        DigitalSize size = new DigitalSize(60000);

        Console.WriteLine(size.Bit);
        Console.WriteLine(size.Byte);
        Console.WriteLine(size.KB);
        Console.WriteLine(size.MB);
        Console.WriteLine(size.GB);
        Console.WriteLine(size.TB);
    }
}
```