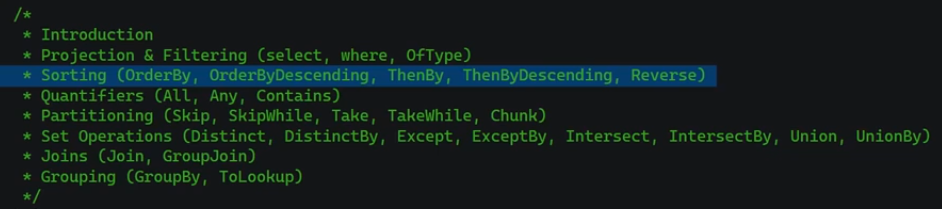
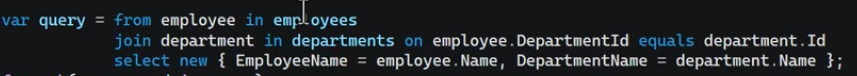
Built-in Function

* Int (Static)
* Int id = int.Parse(Console.ReadLine())
  + Convert string to integer
* Int.TryParse(Console.ReadLine(), out x)
  + Return bool
* Int
* Random
* Environment (Static)
* Environment.CurrentManagedId
* Environment.NewLine
* String (Static)
* String.IsNullOrEmpty(str)
* String
* str.ToLower()
* str.Contains(“--”)
* str.ToCharArray()
* str.Replace(oldValue, newValue)
* Array (Static)
* Array.Sort(arrName)
  + It changes in same array
  + If you want to use it with array of object. Use IComparable
* Array.Reverse(charArray)
* Array
* Int[] tmparr = (int[])arrName.Clone()
  + Clone() -> return object so, use explicit casting.
  + When you change in tmparr that does not change in arrName
* Console
* Console.Clear()
* Console.ReadLine()
* Console.WriteLine(---)
* Convert
* Int id = Convert.ToInt32(Console.ReadLine())
  + Convert string to integer
* Enumerable (Static)
* Enumerable.Range(1,100)
* HashCode (Static)
* HashCode.Combine(1,100)

Linq(Language Integrated Query)



* (Query Expression)
* From --- in --- orderby --- where --- select --- ;
  + From num
  + in nums
  + orderby num, --, --- descending
  + where num > 8 select num;
  + select
  + More readable (Syntax suger)
* (Method Exprssion)
  + .where(lambda exp)
  + arrName.where(x => x > 8)
  + .OrderByDescending(x => x)
  + .ThenBy(x => x.age).ThenBy(x => x.name)
  + .Reverse()
  + .OfType<ClassName>()
  + .Select()
* Methods on result
  + .ToList()
  + .Sum()
* Quantifier Methods
* Return true or false
  + Contains()
  + Any()
  + All()
* Partitioning
* Take some elements from list
  + mylist.Take(numOfItems)
  + mylist.TakeLast(numOfItems)
  + mylist.TakeWhile(x => x <= 7)
  + mylist.Skip(numOfItems)
  + mylist.SkipLast(numOfItems)
  + mylist.SkipWhile(x => x <= 7)
  + myList.Chunk(num)
    - divide list to num of group (chunks)
    - you can use it for multithreading
* Set Operations
* Distinct()
  + Remove repeative items
  + DistinctBy(properityInObj)
* Except(arrExcept)
  + Return all items except those in arrExcept
* Intersect(arrIntersect)
  + Return all items common to arr and arrIntersect
* Union(arrUnion)
  + Return all items in arr and arrIntersect, and remove repetive items
* Union(arrUnion)
* Join & GroupBy

Or





* + - * group.Key => id of dept
      * each group here includes (id of dept) & (list of employes)
* Interview Question
  + Immediate Execution

List<int> numbers = new List<int>{1,2,3,4,5,6,7,8,9};

var result = from num in numbers where num > 7 select num;

            foreach(var item in result)

            {

                Console.WriteLine(item);

            }

  Console.WriteLine("-------------------------------");

            numbers.AddRange(new int[] {12, 13, 14, 15});

            foreach(var item in result)

            {

                Console.WriteLine(item);

            }

Output

8

9

12

13

14

15

When Add ToList() to LINQ

List<int> numbers = new List<int>{1,2,3,4,5,6,7,8,9};

var result = (from num in numbers where num > 7 select num).ToList();

            foreach(var item in result)

            {

                Console.WriteLine(item);

            }

  Console.WriteLine("-------------------------------");

            numbers.AddRange(new int[] {12, 13, 14, 15});

            foreach(var item in result)

            {

                Console.WriteLine(item);

            }

Output

8

9

-------------------------------

8

9

* + IEnumerable VS IQuerable

Buit-in Interfaces

* IEnumerable & IEnumerator
* Used in loop
* Enable loop in class that inherits from it
* public class PayItem
* {
* public string Name { get; set; }
* public int Value { get; set; }
* }
* public class Employee : IEnumerable<PayItem>
* {
* private readonly List<PayItem> \_payItems = new();
* public string Name { get; set; }
* public void AddPayItem(string name, int value)
* {
* if (string.IsNullOrEmpty(name))
* {
* throw new ArgumentNullException("name");
* }
* \_payItems.Add(new PayItem{Name = name, Value = value});
* }
* public IEnumerator<PayItem> GetEnumerator()
* {
* foreach(var item in \_payItems)
* {
* yield return item;
* }
* }
* // Explicit Casting
* IEnumerator IEnumerable.GetEnumerator()
* {
* return GetEnumerator();
* }
* }

Call

     Employee emp = new Employee(){Name = "e1"};

     emp.AddPayItem("pay1", 2000);

     emp.AddPayItem("pay2", 100);

     emp.AddPayItem("pay3", -300);

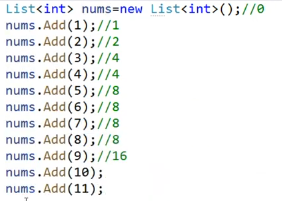
     foreach(var payItem in emp)

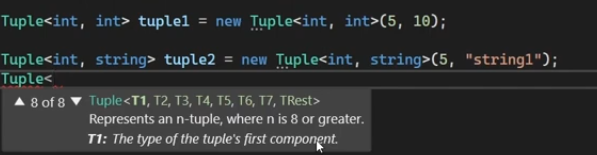
          {

        Console.WriteLine($"name: {payItem.Name}, value: {payItem.Value}");

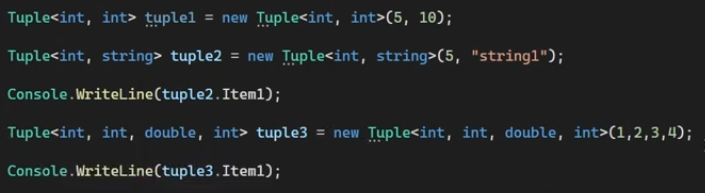
          }

Built-in class

* StringBuilder
* Sb.Append(---)
* Sb.Append(Environment.NewLine)
* ArrayList
* Arr.add(---)
* SortedList
* Arr.add(---)
* HashTable
* Arr.add(---)
* List
* mylist.add(---)
* mylist.addRange(arr)
* mylist.Capacity
  + 
  + Default => capacity = 0
  + Add (1) => capacity = 1
  + Add (2) => capacity = 2
  + Add (3) => capacity = 4 -🡪 capacity will double (2 \* 2)
  + Add (4) => capacity = 4
  + Add (5) => capacity = 8 -🡪 capacity will double (4 \* 2)
* mylist.TrimExcess()
  + remove unnecessary capacity
  + with TrimExcess => capacity = 11
  + without TrimExcess (default) => capacity = 16
* mylist.Remove(--)
* mylist.RemoveAt(indexNum)
* mylist.RemoveRange(index, range)
* mylist.FindAll(item=>item%2)
  + take condition to filter and return list
* mylist.All(item=>item%2)
  + take condition to filter and return list
* mylist.Any(item=>item%2)
  + take condition to filter and return list
* Tuple



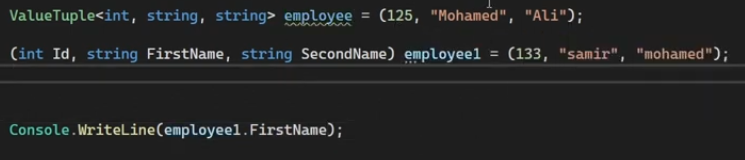
* + From 1 to 8 values with different type if you want.
  + Actually using in return method to return many values



* .Create()



* .Rest()
  + The 8th element
* ValueTuple



* + Enhanced for Tuple
  + Not limited, you can called elements.
  + Actually using in return method to return many values.
* Dictionary<key type, value type>
* dic.add(key, value)
* dic.TryAdd(key, value)
  + return bool
* Foreach(KeyValuePair item in dic)
  + Item.key , item.value

Mutable & Immutable Objects

* Mutable (can change)
* Immutable (can not change)
* String

Collection

* Generic
* List
* Dictionary<key, value>
* SortedList
* Stack
* Tuple
* Non-Generic
* ArrayList
* HashTable

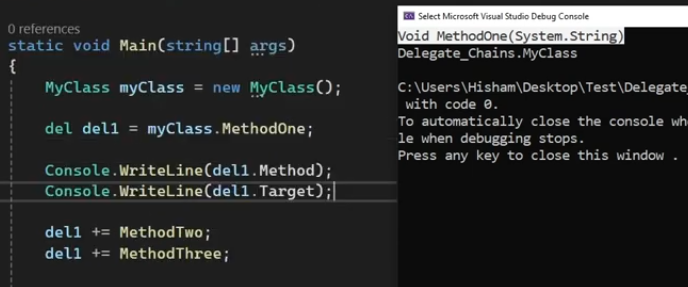
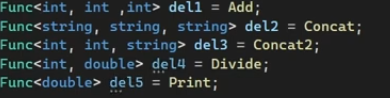
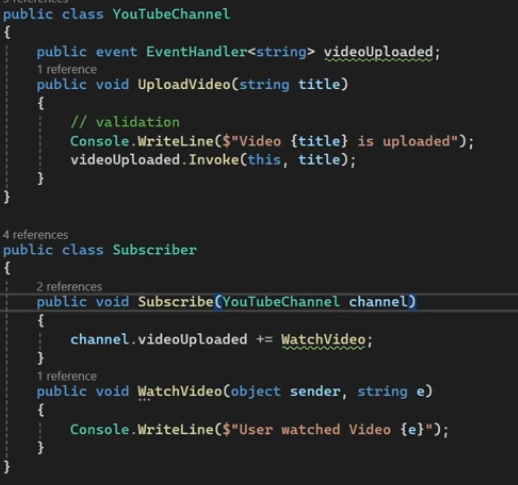
Notes

* + Each collection inherit IEnumerable Interface

Generic (Template) => (T)

* Constraint
* Valid
* Not valid

Delegate

* Syntax
* Public delegate returnType delegateName (inputParams);
* Call
* Anonymous function
* Lambda function (anonymous)
* Generic
* Built-In Function
* .Invoke(funcName)
* .GetInvocationList()
  + Delegate chain
  + Return list
* .Method()
  + Return method that it refer to.
* .Target()
  + Return class name that function belong to.
  + If function is static, will return null.
  + 
* Delegates
* Predicate
  + Take one parameter and return bool
  + Using it to check value
* Action
  + Take one or more parameters and return void
* Func
  + Take one or more parameters and return one parameter (Last input)
  + 
  + 
* Event Handler
  + Take one or more parameters and return one parameter (Last input)
  + 
  + Generic (Most used)
    - 
  + Code
    - 
* Delegate Chain
* (+)
* (-)
* Codes

public delegate bool delCheckItem(int x, int y); // <<<-------

    static List<int> GetFiltredValues(List<int> items, int num, delCheckItem del) // <<<-------

            {

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

                foreach (int item in items)

                {

                    if (del(item, num)) // <<<-------

                    {

                        result.Add(item);

                    }

                }

                return result;

            }

            static bool IsGreater(int x, int y)

            {

                return x > y;

            }

            var output = GetFiltredValues([1, 2, 4, 6, 7, 8, 9, 11], 5, IsGreater); // <<<-------

            foreach(int item in output)

            {

                Console.WriteLine(item);

            }

With Lambda expression

          public delegate bool delCheckItem(int x, int y);

            static List<int> GetFiltredValues(List<int> items, int num, delCheckItem del)

            {

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

                foreach (int item in items)

                {

                    if (del(item, num))

                    {

                        result.Add(item);

                    }

                }

                return result;

            }

            delCheckItem delIsGreater = (int x, int y) => x > y;

            delCheckItem delIsLess = (int x, int y) => x < y;

            delCheckItem delIsEqual = (int x, int y) =>  x == y;

            var output = GetFiltredValues([1, 2, 4, 6, 5, 8, 9, 11], 5, delIsEqual);

            foreach(int item in output)

            {

                Console.WriteLine(item);

            }

Delegate Chain

public delegate void delChain(string input);

delChain del = Method1;

            del += Method2;

            del += Method3;

            del("My input");

            static void Method1(string input)

            {

                Console.WriteLine($"Method1 + {input}");

            }

            static void Method2(string input)

            {

                Console.WriteLine($"Method2 + {input}");

            }

            static void Method3(string input)

            {

                Console.WriteLine($"Method3 + {input}");

            }

Event

* Syntax
* Public event delegateName eventNameInPastSimple;
* Publisher
  + create info (create event)
  + call event in setter
    - if(eventNameInPastSimple() != null) { eventNameInPastSimple();}
    - eventNameInPastSimple?.Invoke();
  + ex => Ball

    public struct Location

    {

        public int X { get; set; }

        public int Y { get; set; }

        public int Z { get; set; }

        public Location()

        {

            X = Y = Z = 0;

        }

        public Location(int \_x, int \_y, int \_z)

        {

            X = \_x;

            Y = \_y;

            Z = \_z;

        }

        public override string ToString()

        {

            return $"Location: ({X}, {Y}, {Z})";

        }

    }

    public class Football

    {

        // publisher

        public int Id { get; set; }

        public Location ballLocation;

        public Location BallLocation

        {

            set

            {

                // if(ballLocation != null)

                // {

                    ballLocation = value;

                    ballLocationChanged?.Invoke();

                // }

            }

            get

            {

                return ballLocation;

            }

        }

        public event Action ballLocationChanged;

    }

* Subscriber
* make notify
* call back method
* ex => Player

    public class Player

    {

        // Subscriber

        public string Name { get; set; }

        public string Team { get; set; }

        public void Run()

        {

            Console.WriteLine($"Player: {Name}, Clup: {Team}");

        }

    }

    public class Refers

    {

        // Subscriber

        public string Name { get; set; }

        public void Observe()

        {

            Console.WriteLine($"Observe: {Name}");

        }

    }

* Link between Publisher & Subscriber
* objName.eventNameInPastSimple += callBackMethodName
* objName.eventNameInPastSimple -= callBackMethodName

    Player p1\_1 = new Player() { Name = "Ahmed", Team = "Ahly" };

    Player p2\_1 = new Player() { Name = "Hassan", Team = "Ahly" };

    Player p1\_2 = new Player() { Name = "Mohammed", Team = "Alex" };

    Player p2\_2 = new Player() { Name = "Mostafa", Team = "Alex" };

    Refers r = new Refers() { Name = "Osama" };

    Football ball = new Football() { Id = 1, BallLocation = new Location() { X = 22, Y = 33, Z = 55 } };

    ball.ballLocationChanged += () => Console.WriteLine("Start ...");

    ball.ballLocationChanged += p1\_1.Run;

    ball.ballLocationChanged += p2\_1.Run;

    ball.ballLocationChanged += p1\_2.Run;

    ball.ballLocationChanged += p2\_2.Run;

    ball.ballLocationChanged += r.Observe;

    ball.BallLocation = new Location { X = 33, Y = 55, Z = 77 };

Output

Start ...

Player: Ahmed, Clup: Ahly

Player: Hassan, Clup: Ahly

Player: Mohammed, Clup: Alex

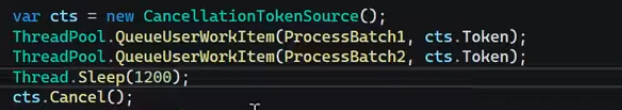
Player: Mostafa, Clup: Alex

Observe: Osama

Exception

* Throw
* ArgumentNullException(“”)

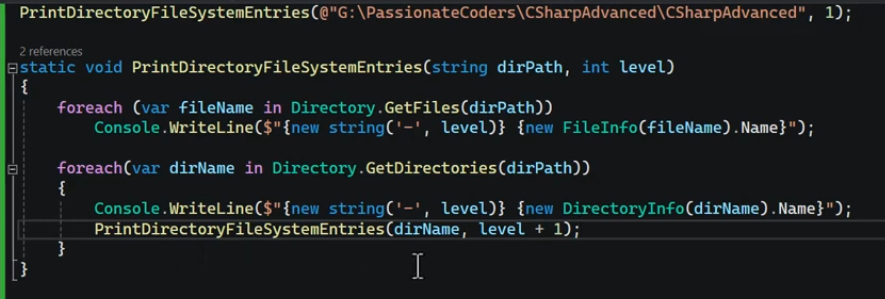
MultiThreading

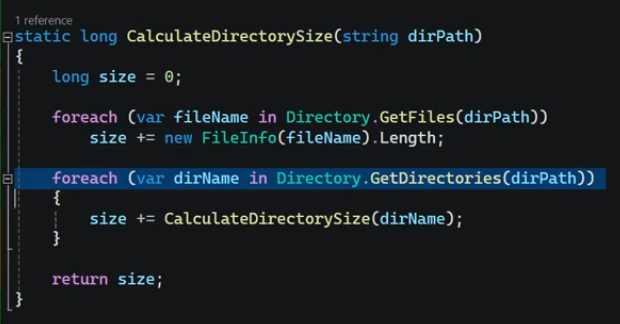
* Syntax
* Var th = new Thread(MethodName)
* ThreadPool.QueueUserWorkItem(MethodName, status)
  + 
* Built-In Function
* th.start()
* th.Priority = ThreadPriority.Highest
* th.IsBackground = true;
  + close the program even if thread not finished
  + default +> foreground
* Static Function
* Thread.Sleep(1000)
  + Executed in current thread. (used in method)
* Task
* await Task.WhenAll(task1, task2)
  + Upon completion task1 & task2, the main task will begin.
* Await TaskWhenAny(task1, task2)
  + Upon completion any task, the main task will begin. And the other task will run
* Async
* Await

Database

* Download
* Microsoft.sql.dataClient from Nuget.org
* Using Microsoft.sql.dataClient
  + Includes => sqlCommand, sqlDataReader, sqlConnection
* Connection String
* sqlConnection con = new sqlConnection(“”)
  + Windows Auth
    - (“Server=IP Add; Database= databaseName; Trusted\_Connection = true; TrustServerCertificate = true“)
  + Sql Server Auth
    - (“Server=IP Add; Database= databaseName; user id = Uname; Password = pass”)
* Built-in function
  + Con.open();
  + Con.close();
* Command
* sqlCommand command = new sqlCommand()
* Built-in function
  + Command.commandType = Data.CommandType.Text;
  + Command.commandText = “Select \* from ----”;
  + Command.connection
  + Command.ExecuteReader();
* Link command to connection object
* Command.connection = con;
* Try {con.Open();} & catch{} finally{con.close();}
* Try & Catch
* Open connection
  + Con.Open()
* Execute Query [SqlDataReader] =>
  + SqlDatatReader dr = command.ExecuteReader();
  + Sql Server Auth
* Data Bining
  + Using System.Data;
    - Include Datat Table
  + DataTable dt = new DataTable()
  + Dt.Load(dr)

File



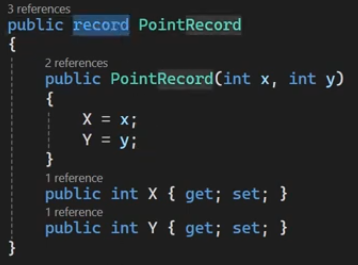


Access Modifier

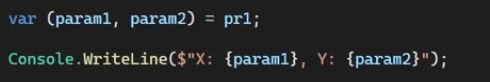
* Public
  + All
* Private
  + Class ONLY
* Protected
  + Child
* Internal (default)
  + Assembly (project)
* Protected Internal
  + Child || Assembly
* Private Protected
  + T

Record

* Put it instead of class keyword



* To make on it => instead write override
  + - ToString()
    - ==, Equals()
    - GetHashCode()
* Positional record
  + - Deconstruction
    - Immutability



* + - Copy to change the value.



Student std = new Student("Amira", 22, new Address(){City = "Cairo", StreetName = "Nasir"});

  // std.address = null; // HERE => You Can NOT Edit It

  std.address.City = "Alex"; // HERE => You Can Edit It

     Console.WriteLine(std.address.ToString());

* Immutability

OOP

* Association

class Teacher

    {

        public void Teach(Student student)

        {

            Console.WriteLine($"{student.Name} is learning.");

        }

    }

    class Student

    {

        public string Name { get; set; }

    }

Notes

* + - It is a weak relationship
* Composition

    class Room { }

class House

    {

        private Room room;

        public House()

        {

            room = new Room();

        }

        public Room GetRoom()

        {

            return room;

        }

    }

Notes

* + - It is a strong relationship
* Interface
* Generalization
* Inheritance
* Polymorphism
* Abstract

Some codes

* Jagged Array

Code

#region Jagged array -- Rarely used

            int[][] jaggedArr = new int[3][] { new int[3], new int[5] {1,2,3,4,5}, new int[2] };

            foreach(int[] rows in jaggedArr)

            {

                foreach(int col in rows)

                {

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

                }

                Console.WriteLine("Row: "+ rows);

            }

 #endregion

Output

col:0

col:0

col:0

Row: System.Int32[]

col:1

col:2

col:3

col:4

col:5

Row: System.Int32[]

col:0

col:0

Row: System.Int32[]

* properties - Setter & Getter

public class Emp

    {

        int id;

        public int Id

        {

            set

            {

                id = value;

            }

            get

            {

                return this.id;

            }

        }

    }

Call

Emp emp1 = new Emp();

            emp1.Id = 20;

            Console.WriteLine(emp1.Id);

Notes

* + - The first character in method name must be capital
* Automatic Property

public class Employee

    {

public int Id { get; set; }

    }

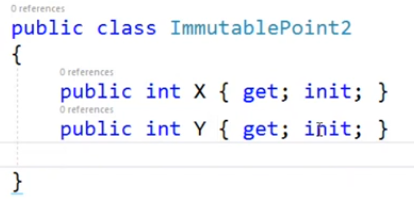
Call

Employee emp1 = new Employee();

      emp1.Id = 20;

      Console.WriteLine(emp1.Id);

* Init (Immutable)



Notes

* + - you cant set value in X or Y just set them using Constructor
    - Immutable => cannt change the value.
* Property Initializer

Call

Employee emp1 = new Employee() { Id = 10, Age = 22, Name = "mira"};

      Console.WriteLine(emp1.Age); // 22

Notes

* + - All properties must have setter & getter
* Call By Reference (ref)

    public static class MathHelper

    {

        public static void AddRef(int num1, int num2, ref int res)

        {

            res = num1 + num2;

        }

    }

Call

int x = 3, y = 5, result = 0;

    MathHelper.AddRef(x, y, ref result);

    Console.WriteLine(result); // 8

Notes

* + - Must be assigned value in main code
* Call By Reference (out)

    public static class MathHelper

    {

        public static void AddOut(int num1, int num2, out int res)

        {

            res = num1 + num2;

        }

    }

Call

int x = 3, y = 5, result;

    MathHelper.AddOut(x, y, out result);

    Console.WriteLine(result); // 8

Notes

* + - Must be initialized in function
* Static Variable

static int count;

public int Count

        {

            set

            {

                count = value;

            }

            get

            {

                return count;

            }

        }

Call

Emp emp1 = new Emp();

       emp1.Count = 10;

       Emp emp2 = new Emp();

       Console.WriteLine(emp2.Count); // 10

Notes

* + - It stored in one place
* Static Method

public class MathHelper

    {

        public static int Add(int num1, int num2)

        {

            return num1 + num2;

        }

    }

Call

     Console.WriteLine(MathHelper.Add(3, 6)); // 9

Notes

* + - We call it with class name
* Static Class

    public static class MathHelper

    {

        public static int Add(int num1, int num2)

        {

            return num1 + num2;

        }

    }

Notes

* + - Class each member inside it must be static
    - No needed to constructor so no object created
* Static Constructor

static Employee()

        {

            count = 5;

        }

public Employee()

{

}

Notes

* + - It has no access modifier even public
    - Doesn’t conflict with default constructor
    - It will be called once per program call ONLY
* Static Constructor (call once)

public static List<Employee> GetEmployees { get; set; } =

        new List<Employee>()

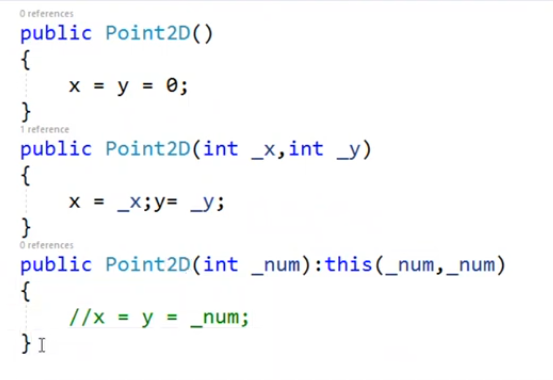
        {

            new Employee(){Age = 22, Id = 1, Name = "amira"}

        };

Call

var employees = Repository.GetEmployees;

* Constructor Chain
* Operator Overloading

public class Complex

    {

        public int Real { get; set; }

        public int Img { get; set; }

        public Complex()

        {

            Real = Img = 0;

        }

        public Complex(int \_real, int \_img)

        {

            Real = \_real;

            Img = \_img;

        }

        public override string ToString()

        {

            return $"{Real}+{Img}i";

        }

// Addition

        public static Complex operator+(Complex left, Complex right)

        {

            Complex retval = new Complex() { Real = left.Real + right.Real, Img = left.Img + right.Img };

            return retval;

        }

// Incremental

public static Complex operator++(Complex com)

        {

            Complex result = new Complex() { Real = com.Real + 1, Img = com.Img + 1 };

            return result;

        }

// Check

public static bool operator >(Complex left, Complex right)

        {

            return (left.Real > right.Real) && (left.Img > right.Img);

        }

        public static bool operator <(Complex left, Complex right)

        {

            return (left.Real < right.Real) && (left.Img < right.Img);

        }

        public static bool operator ==(Complex left, Complex right)

        {

            return (left.Real == right.Real) && (left.Img == right.Img);

        }

        public static bool operator!=(Complex left, Complex right)

        {

            return !(left == right);

        }

// Implicit & Explicit

public static implicit operator int(Complex com)

        {

            return com.Real;

        }

        public static explicit operator string(Complex com)

        {

            return com.ToString();

        }

    }

Call

Addition

Complex left = new Complex(){ Real = 22, Img = 3 };

    Complex right = new Complex(){ Real = 10, Img = 2 };

    Complex result = left + right;

    Console.WriteLine(result.ToString()); // 32+5i

Incremental

Complex com = new Complex() { Real = 22, Img = 3 };

    Console.WriteLine(com.ToString()); // 22+3i

    Complex resultIncrement = com++;

    Console.WriteLine(com.ToString()); // 23+4i

    Console.WriteLine(resultIncrement.ToString()); // 22+3i

Check

Complex left = new Complex(){ Real = 22, Img = 3 };

    Complex right = new Complex() { Real = 10, Img = 2 };

    bool resultGreater = left > right;

    bool resultLess = right < left;

    bool resultEqual = right == left;

    bool resultNotEqual = right != left;

    Console.WriteLine(resultGreater.ToString()); // True

    Console.WriteLine(resultLess.ToString()); // True

    Console.WriteLine(resultEqual.ToString()); // False

    Console.WriteLine(resultNotEqual.ToString()); // True

Implicit & Explicit

Complex com = new Complex() { Real = 22, Img = 3 };

      int real = com;

      string comExplicit = (string)com;

      Console.WriteLine(real); // 22

      Console.WriteLine(comExplicit); // 22+3i

* Sort Array of user-defined DT

public class Employee:IComparable

    {

        int age;

        public int Id { get; set; }

        public int Age

        {

            set

            {

                age = value;

            }

            get

            {

                return age;

            }

        }

        public string Name { get; set; }

        public override string ToString()

        {

            return $"Id:{Id}, Age:{Age}, Name:{Name}";

        }

// Sorting

        public int CompareTo(object? obj)

        {

            Employee emp = obj as Employee;

            if(Age.CompareTo(emp.Age) == 0)

            {

                return Name.CompareTo(emp.Name);

            }

            return Age.CompareTo(emp.Age);

        }

    }

Notes

* + - We use IComparable Interface to Implement CompareTo() that I want to it.
    - If age equality compare with name.

Call

Employee[] employees = new Employee[]

            {

                new Employee{ Id = 1, Age = 22, Name = "amira" },

                new Employee{ Id = 2, Age = 11, Name = "mira" },

                new Employee{ Id = 3, Age = 22, Name = "moraa" }

            };

            Array.Sort(employees);

            foreach(var emp in employees)

            {

                Console.WriteLine(emp.ToString());

            }

Output

Id:2, Age:11, Name:mira

Id:1, Age:22, Name:amira

Id:3, Age:22, Name:moraa

* Clone Object

Problem

int[] arr = { 1, 2, 3, 4, 5 };

   int[] tmpArr = arr;

   tmpArr[1] = 1000;

   foreach(int i in arr)

   {

       Console.WriteLine(i); // 1, 1000, 3, 4, 5

   }

Solution with clone

   int[] arr = { 1, 2, 3, 4, 5 };

   int[] tmpArr = (int[])arr.Clone();

   tmpArr[1] = 1000;

   foreach (int i in arr)

       {

          Console.WriteLine(i); // 1, 2, 3, 4, 5

       }

* Clone Object of user-defined DT

    public class Employee:ICloneable

    {

        int age;

        public int Id { get; set; }

        public int Age

        {

            set

            {

                age = value;

            }

            get

            {

                return age;

            }

        }

        public string Name { get; set; }

        public override string ToString()

        {

            return $"Id:{Id}, Age:{Age}, Name:{Name}";

        }

        public object Clone()

        {

            return new Employee{Id=this.Id, Age=this.Age, Name=this.Name};

        }

    }

call

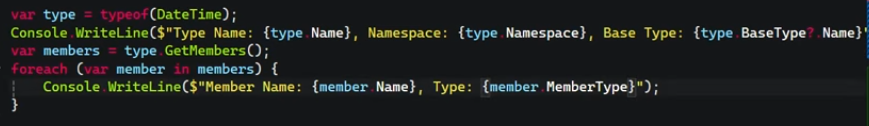
Employee emp = new Employee { Id = 1, Age = 22, Name = "amira" };

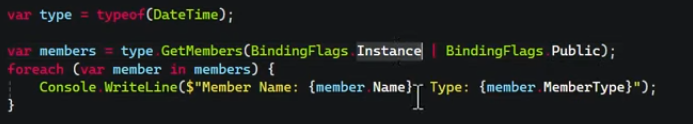
Employee cloneEmp = emp.Clone() as Employee;

cloneEmp.Age = 33;

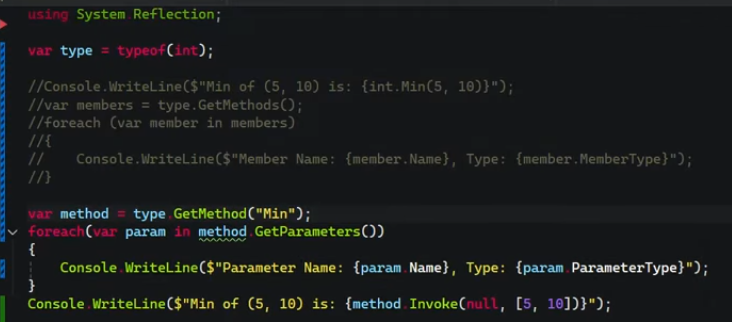
Console.WriteLine(emp.ToString()); // Id:1, Age:22, Name:amira

Assembly (Reflection)

* Static Methods
* GetExecutingAssembly()
  + - * .FullName
      * .Location
      * .GetManiFestResourcesStream(“projectName.FolderName.FileName.Extension”)
        + .Length
        + .Read(fileBytes)
      * .GetTypes()
        + .Name
        + .NameSpace
        + .BaseType?.Name
      * .GetExpertedTypes()
* .GetCallingAssemply()
* .GetEntryAssemply()
* TypeOf(---)
* GetMembers()
  + Return all methods in DateTime



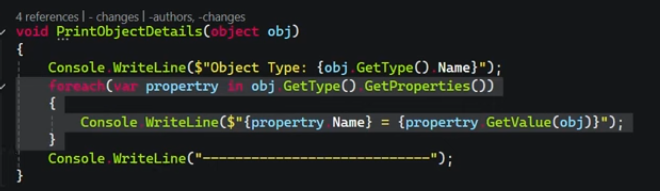
* + Identify return methods from BindingFlag
  + Here =>Return public methods.
* .GetProperties()
* .GetMethods()
* .GetMethod(“MethodName”)
  + - * .GetParameters()
        + Return list
        + .Name
        + .ParameterType
      * .Invoke(instanceName, arrayOfValues)
        + InstanceName => write null instead it if method is static



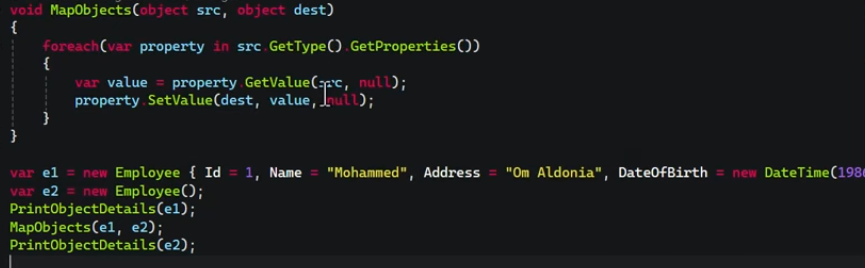
* Code



* To Access on classes in dll.



* Print all values in object



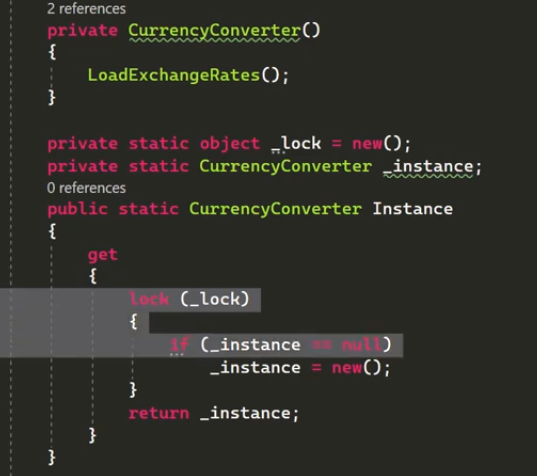
* Make mapping object => copying obj to another.

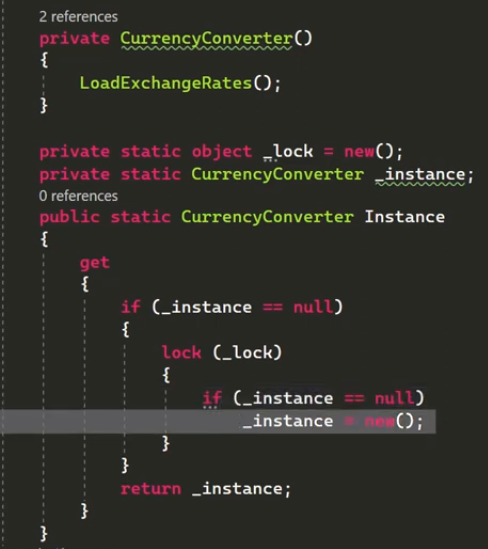
SOLID Principles

* Singe Responsibility Principle (SRP)
* Open / Close Principle (OCP)
* Lisoke Substitution Principle (LSP)
* Interface Segregation Principle (ISP)
* Dependency Inversion Principle (DIP)

Design Pattern

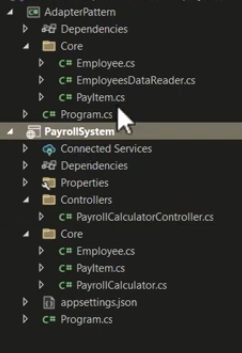
* Creational
* Singleton
  + Using
    - For create one instance alone from class
  + When
    - Static data
    - Code will run once.
  + Code



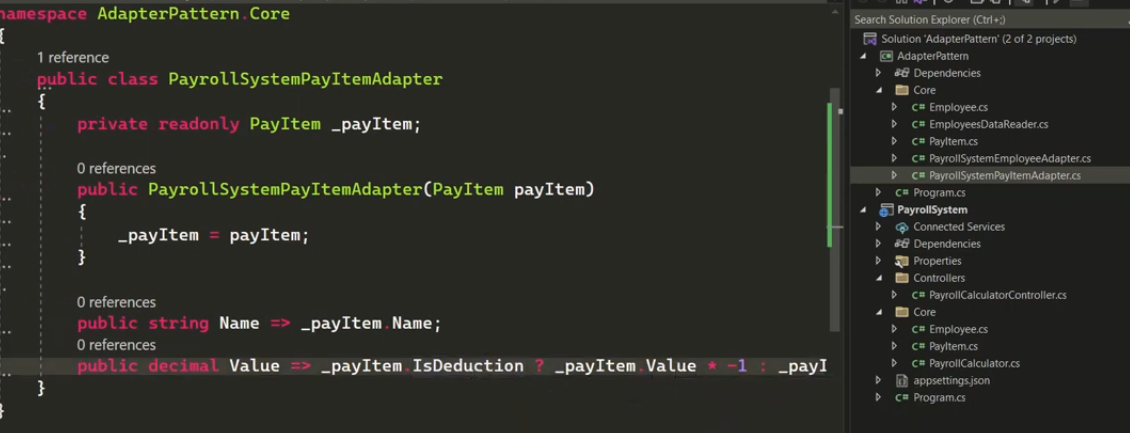
* + - Double check
  + Calling



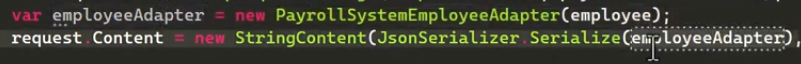
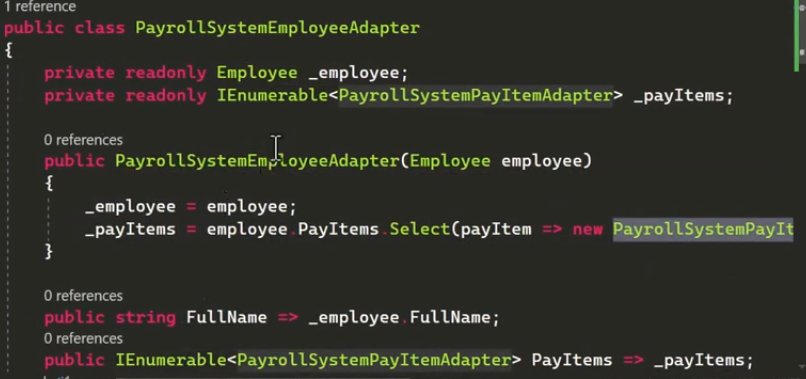
* + Steps
    - Private constructor
    - Instance called \_instance
    - Abstract on \_instance
      * Getter with Lock on it.
  + Benefit
    - In the first time, will load data, after that will return data fast.
  + VS Static Class
    - in static class
      * you can not apply object oriented
        + inheritance
        + implementation interface
        + method parameters
* Structural
* Adapter
  + Using
    - For change the shape of object
  + When
    - Legacy Sys
    - Any project you don’t have source code
  + Code



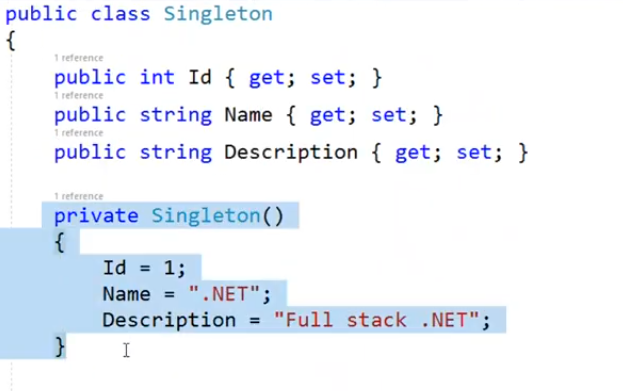
* + - PayrollSystem => you can not change it
    - AdapterPattern => this is your project
    - Editing => business want to add properity to IsDeduction and change in calc of Value



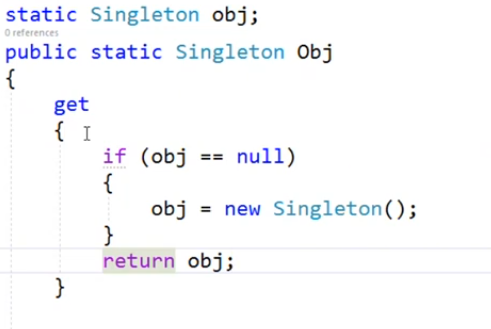
* + - Create two classes ended with Adapter word.
    - Calc of value.



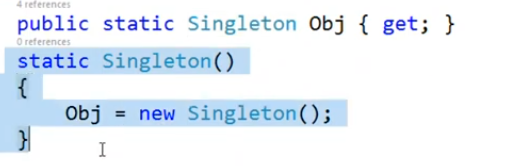
* + Steps
    - Legacy Sys
* Behavioral
* Strategy / Policy
  + Using
    - Identifying method in runtime.
  + When
  + Code
* Singleton



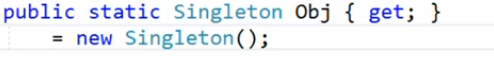
V1



V2



V3



* Observe Pattern (Subject & Publisher)



namespace DesignPattern;

// ObservePattern

public class VideoYoutubeChannel

{

    public delegate void VideoUpload(string videoTitle);

    public event VideoUpload videoUploadEvent;

    public void uploadVideo(string videoTitle)

    {

        Console.WriteLine($"New video uploaded {videoTitle}");

        if(videoUploadEvent != null)

        {

            videoUploadEvent.Invoke(videoTitle);

        }

    }

}

public class Subscriber

{

    public void subscribe(VideoYoutubeChannel videoYoutubeChannel)

    {

        videoYoutubeChannel.videoUploadEvent += watchVideo;

    }

    public void watchVideo(string videoTitle)

    {

        Console.WriteLine($"User watch {videoTitle}");

    }

}

Call

namespace DesignPattern;

internal class program

{

    static void Main(String[] args)

    {

        VideoYoutubeChannel video = new VideoYoutubeChannel();

        Subscriber subscriber1 = new Subscriber();

        Subscriber subscriber2 = new Subscriber();

        Subscriber subscriber3 = new Subscriber();

        Subscriber subscriber4 = new Subscriber();

        subscriber1.subscribe(video);

        subscriber2.subscribe(video);

        subscriber3.subscribe(video);

        video.uploadVideo("Video1");

    }

}

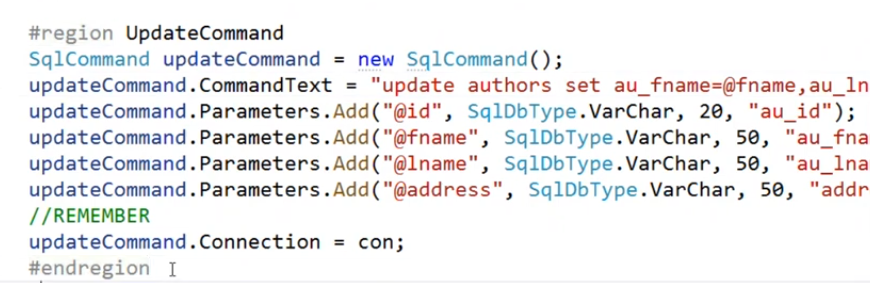
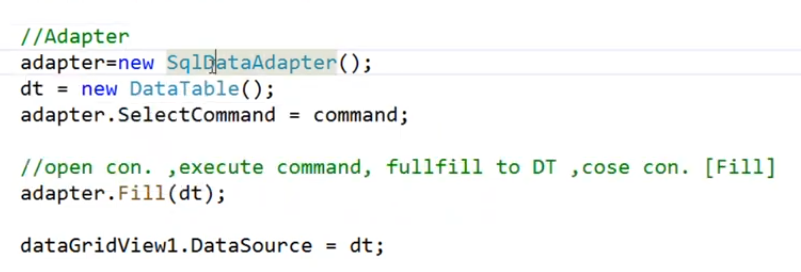
* Pub & Sub Pattern
  + Broker

Memory

* Stack
  + Temporary values
    - Local variables
    - Functions
* Heap
  + Reference
    - Any object created with new
* Method table / Type metadata
  + Class Info
    - Name
    - Properties
    - Methods
* Static Storage Area
  + R
    - Static
* Code Segment (IL Code)
  + R
    - Main code

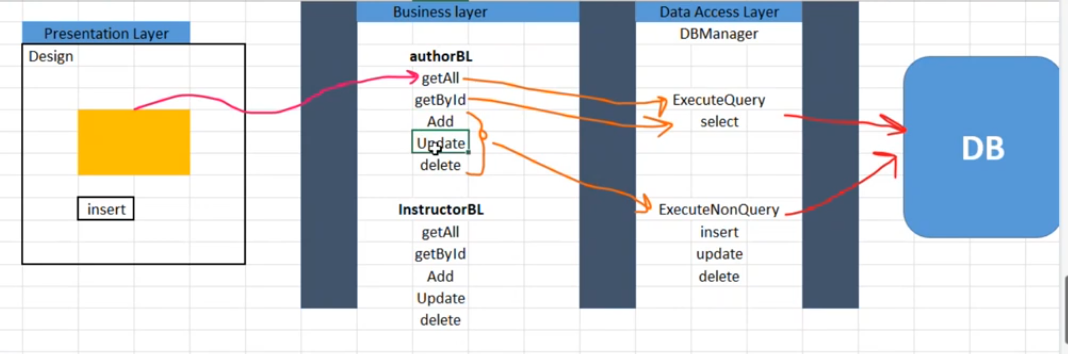
C# Server & DB Server

* Connected
* Disconnected
* Adapter

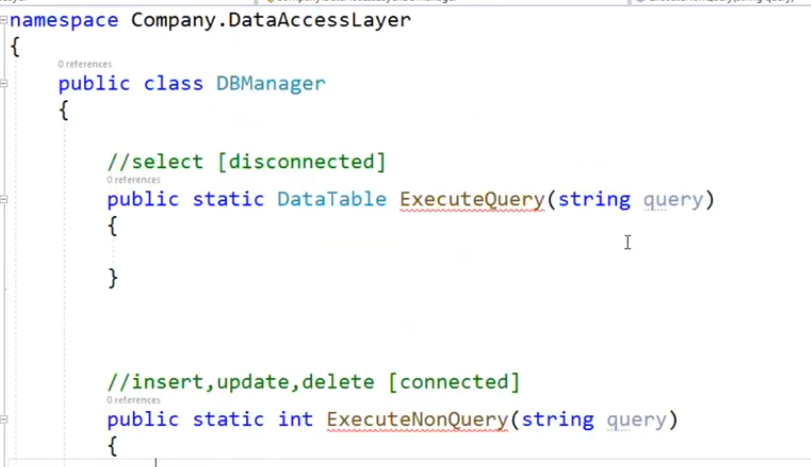




ADO.net



* Data Access Layer







Static => if you want to make object always program is run that caused multithread

* Business Layer



* Presentation Layer

Problem Solving

* Find the longest distance between two equal cells

// int[] nums = { 7, 0, 0, 0, 5, 6, 7, 5, 0, 7, 5, 3 }; // 8

            int[] nums = { 1, 1, 1, 1, 1, 1 }; // 4

            int maxDistance = 0, outputNum = -1;

            for (int first = 0; first < nums.Length - 1; first++)

            {

                for (int last = first + 1; last < nums.Length; last++)

                {

                    if ((nums[first] == nums[last]) && (maxDistance < last - first - 1))

                    {

                        maxDistance = last - first - 1; // 5

                        outputNum = nums[first];

                    }

                }

            }

            Console.WriteLine($"maxDistance: {maxDistance}, outputNum: {outputNum}");

* Reverse the order of the words

string text = Console.ReadLine(); // this is a test

       string[] sentence = text.Split(' ');

      List<string> reverseSentence = new List<string>();

       for (int word = sentence.Length - 1; word >= 0; word--)

         {

          reverseSentence.Add(sentence[word]);

        }

       Console.Write(string.Join(" ", reverseSentence)); // test a is this

* Count the occurrence of 1 from 1 to 99,999,999

0 -> 9 = 1

0 -> 99 = 20

0 -> 99,9 = 300

0 -> 99,99 = 4000

0 -> 99,999 = 5000,0

0 -> 99,999,9 = 6000,00

0 -> 99,999,99 = 7000,000

0 -> 99,999,999 = 8000,000,0

int num = 99999999; // 99,999,999

            int countZero = 0, countOne = 1;

            while (num / 10 != 0)

            {

                num = num / 10;

                countZero++;

            }

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

            {

                countOne = 10 \* countOne;

            }

            Console.WriteLine(countOne \* (countZero + 1)); // 8000,000,0

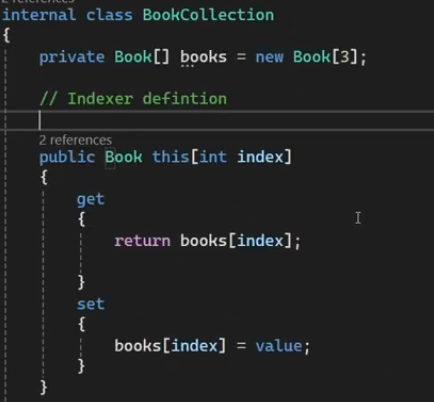
Shortcuts

* Prop => setter & getter (Automatic property)
* Ctor => default constructor
* Cw => Console.WriteLine();

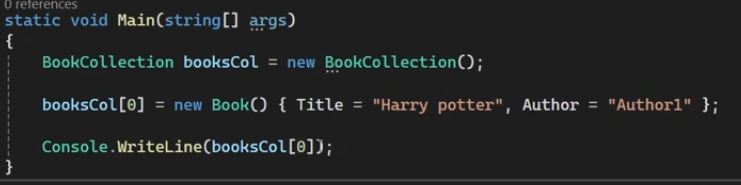
Notes

* Convert VS Parse
  + Convert
    - If takes null will return empty
  + Parse
    - If takes null will return error
* Array vs ArrayList vs List vs Dictionary
  + Array
    - Fixed size
    - Must all elements same type
    - Use indexer with insert, update, get
  + ArrayList
    - Take object (elements can be different data type)
    - Use indexer with insert, update, get
  + List
    - Dynamic size
    - Must all elements same type
    - Use indexer with update, get ONLY (insert => ERROR: OUT OF RANGE)
  + Dictionary
    - Dynamic size
    - It has key and value
    - Use indexer with insert, update, get 🡪 dic[“amira”] = 22
* Struct vs Class
  + Struct
    - Members are value type
    - No inheritance
  + Class
    - Members are reference type

Indexer



* + To use class as indexer
  + Call



Enhance

* Class Filtration
  + Make functions to filter from database
    - Functions are static