Singletons!!!

extension method!!!

Asyncronousness!!!

ConcurrentDictionary!!!

**Enums**

Enum classes can be created **inside** and **outside** of a class.

If we try to access enums as static data member **strings** will be returned instead of **int**.

**Console.WriteLine(Zoro.Bear);**

To get integer **convert** it:

**Console.WriteLine((int)Zoro.Bear);**

Basically enums are used to create names for different numbers.

**Operator Overloading**

To overload simple arithmetic **operators(+,-,\*,/)** we need to do it statically.When we overload simple operators, related ones(+=,-=,\*=,/=) will also be overloaded automatically.

public static MyClass operator +(MyClass M1,MyClass M2){

return (new MyClass(M1.Var + M2.Var));

}

If we want to overload **comparison** operators we are required to overload all **pairs**.

public static bool operator >(MyClass M1,MyClass M2){

return M1.Var > M2.Var;

}

public static bool operator <(MyClass M1,MyClass M2){

return M1.Var < M2.Var;

}

In C# **assignment** operator can’t be overloaded.

If we overload ++ or - - both **postfix** and **prefix** will be overloaded(but both of them will do the same thing ).

**Copy Constructor Overloading**

**Copy Constructor** has to be called differently :

MyClass M2 = new MyClass(M1);//Copy constructor call

**Implicit&explicit**

Distinguishing from c++ for implicit creation constructor is not a thing in C#. **Implicit** operator allows to convert from one type to another implicitly.

public static implicit operator MyClass(int M1){

return new MyClass(M1);}

To overload explicit conversion operator.

public static explicit operator MyClass(int M1){

return new MyClass(M1);

}

**Explicit** operator does not allow any more implicit conversion, but we have to hard code conversion explicitly.

MyClass m1 = 22;//Not valid(ERROR)

MyClass m2 = (MyClass)22;//Valid

**Ref**

In C# class(inherits reference type) objects are passed by **value**,but value of the user defined types is reference, it looks like reference passing. But if we assign them a new object inside, then the previous reference will be lost. (As in JS)

**Structs** inherit value types so it’s passed by value. Value of struct is not a reference so it’s passed and looks like passed as value.

**ref** keyword allows as to pass the references of the object, so every action which will be held on the given parameter will affect the original one.

FunctionName(ref Z1);

static void FunctionName(ref Zoro text){}

Ass C++ in c# we can:

I)Create reference to variable:

int a = 2;

ref int b = ref a;

II)Pass and get value by reference:

FuncName(ref varName);

static void FuncName(ref type varName){}

III)Return value by reference

ref int RefVarName = ref RetRef(ref varName);

static ref int RetRef(ref int par){return ref par;}

We can make reference variables **readonly,** so it’ll be immutable.

ref readonly int RefVarName = ref varName;

aw

if a function which returns ref can’t return it we must throw an **exception**.

**In&Out**

Both **in** and **out** key work is a **ref** keyword with extra functionality.

**Out** keyword just requires you to give a value to a passed parameter.

int arg = 11;

FuncName(out arg);

static void FuncName(out int par){

par = 0;//Should initialize

}

We **can’t** overload functions **distinguished** only by **ref** and **out** keyword.

static void FuncName(out int par){par = 0;}

static void FuncName(ref int par){}//This can’t be done

**Out** allows al create var inside function call, but **ref** doesn’t:

FuncName(out int name);//Valid

FuncName(ref int name);//Invalid

We can’t use **out** in **async** methods and some others.

When we are taking an argument using the ‘i**n**’ keyword, it becomes **immutable,** so it’s ‘**readonly**’ and we can’t change it’s value.

static void Zoro(in int Par){

Par = 22;//Not valid

}

To call this kind of function we shouldn’t use **in** keyword as in case of **ref** and **out**.

Because struct is value type and when we pass it to a function it creates copies and that’s bad for memory efficiency programmers prefer to use **ref** keyword**,** but then it allows us to change the actual values of struct object and that can lead to an idea of using **in** keyword,but we shouldn’t do this until the properties of the struct is not read only(otherwise Performance will be much slower, because **defensive copies** will be created behind the scene).

**Multiple returns**

When we are returning multiple variables from function we can give them **name** as well:

static (string Name1,string Name2) Func(){

return ("Val1","Val2");}

There are two ways of **utilizing** multiple returns.

//First way

var (Var1,Var2) = Func();

//Second way

var Name = Func();

**Exceptions**

Exception in C# work in the same way as in C++ and in addition as in JS we have a block called **finally, which is being executed always, no matter if an exception was thrown or not.**

**try{**

**Func();**

**}catch(MyException M1){**

**Console.WriteLine(M1.Message);**

**}finally{**

**Console.WriteLine("Finally has called");**

**}**

**Precision**

To change the precision of the number we should use **ToString()** method.

Console.WriteLin**e((2.42123).ToString("0.0"));**

//Output: 2.4

**List**

List has method **orderBy()** which orders the list in **ascending** order.It returns

**IEnumerable** or **IOrderedEnumerable** so if we want to assign this changes it should be converted into a list using ToList() method.

List<int> ListName = new List<int>(){6,5,4,3,2,1};

ListName = ListName.OrderBy(x => x).ToList();

**x => x**\_ is like a foreach loop. We say that x is the item of list(omitting type) and listing based on this item.(We should specify items, components or any int that can be sorted).

List has another method to order its element in descending order called **OrderByDescending()**.

But what if we don’t only want to order a list using 1 criteria,but multiple. Then we use **ThenBy** or **ThenByDescending**:

ListName = ListName.OrderBy(x => x).ThenBy(x => x.y).ToList();

ListName = ListName.OrderByDescending(x => x).ThenByDescending(x => x.y).ToList();

Then we have a **Where** method which IEnumerable or IOrderEnumerable subList of the given one, where all values of this subList satisfies the given condition(boolean expression).

ListName = ListName.Where(x => x > 3).ToList();

**Sum()\_**method takes value to be summed and returns sum.

int Sum = ListName.Sum(x => x);

**ArrayList**

To get access on the ArrayList data structure we need to include namespace **System.Collections.**

**ArrayList** in C# is similar to an array,except it grows automatically(Dynamic) and can hold any type of data(Non Homogeneous).

ArrayList is inherited from an **IEnumerable** which gives us the ability to use a foreach loop on it.

public class ArrayList : ICollection, IEnumerable, IList, ICloneable

**System.object** is the ultimate base class for all .Net classes and that’s the reason why we can have different types of elements in **ArrayList(**they are stored as objects).

**AddRange()** method is used to append a range of objects(Or another ArrayList) to an ArrayList.

A1.AddRange(new ArrayList(){"Zoro",12,42,"ninja"});

If we want to sort an ArrayList we need to make sure that it’s homogeneous, and then use **Sort()** method.

If we try to assign an ArrayList element to another variable we need to **convert it from object to desired type** explicitly.

int val = (int)A1[0];

That’s one of the reasons why we should not use ArrayList,because if we try to convert an arrayList item into another type it will give us a **runtime exception.**

**ArrayList A1 = new ArrayList(){213};**

**string val = (string)A1[0];//No CompileTime Error,but runtime**

**Generics**

All the .Net generic Data Structures are in **System.Collections.Generic;**

As in C++, C# gives us an ability to have **multiple** generic variables.

ClassOrMethodName<T1,T2,T3>

By default if we don’t add any **constraints** to Generic type it will be assumed as an **object**. So we won’t be able to make some operations such as comparisons. In order for a type to be comparable it has to implement **IComparable** interface(Now we can use this requirement for constrain).

static int Zoro<T>(T a,T b)where T : IComparable{

return a.CompareTo(b);//Returns 1 for true and -1 for false

}

We can create a generic method into non generic class.

In C# we have different types of constraints:

1) **To interface Ex: IComparable**

2) **Value type Ex: struct**

3) **Reference type Ex: class**

4) **Default constructor: new()**

**Delegat**

Delegates can hold multiple methods and when we call it all the methods will be invoked in an order they have been assigned.

**Anonymous** methods can be assigned to delegates but **can’t be removed.**

To add multiple/remove multiple methods from the delegat we use **+= and -=** operators.

newDel += M1.Meth;

We can use null checking invoke for delegates:

newDel?.Invoke("string");

We have **?.DynamicInvoke()** used for polymorphism, to define which delegate we are calling based on signature.But it’s like 10 times **slower** than normal invoke.

static void PassMeDel(Delegate myDel){

myDel?.DynamicInvoke(signature);

}

Delegat also has **Target** property which is lastly assigned methods class.

Console.WriteLine(D1.Target?.GetType().Name);

**Action&Func**

**Action** is a built-in delegate which has return type void(all the time) and takes functions with no signature. But if we use a generic version it can take some parameters(up to 16).

Action A1 = new Action(() => Console.WriteLine("Does something"));

Action<int> A2 = new Action<int>(x => Console.WriteLine(x));

**Func** is a delegate which should have a return type. Return type can be specified as a lastly given generic parameter. If Func holds multiple methods, it’ll only return lastly assigned methods returned value.

Func<int,int> Zoro = new Func<int,int>((x) => {return x +3;});

Zoro += (x) => x+4;

Console.WriteLine(Zoro(0));

**Events**

**Events** send data with notifications.

If we want to give ability to a class to publish event we need tree thing.

I)**Define delegate(**Contract or agreement between publisher and subscriber). Delegate determines the signature of the method and subscriber(which is also a method) that will be called when the publisher publishes an event.

public delegate void TalkEventHandler(object source,EventArgs args);

**Object and EventArgs** are just conventions, we can make any kind of delegate which may take only objects,only data, or nothing at all.

In .Net we have a built-in delegate with name **EventHandle** which has the generic version as well and it takes the object as its first parameter and all the others should be specified the **<>.**

**public event EventHandler<int>? WorkCommandHasGiven;**

II)**Define an Event** based on that delegate.

public event TalkEventHandler? ZoroHasSpoken;

III)**Raise or publish** the event.

protected virtual void OnSpeak(){

ZoroHasSpoken?.Invoke(this,EventArgs.Empty);

}

**Simler version for Event:**

public event EventHandler<EventArgs>? ZoroEvent;

public void OnCallEvent(){

ZoroEvent?.Invoke(this,EventArgs.Empty);

}

And that’s the subscriber:

public void Name(object? sender,EventArgs arg){

//Does something with given data

}

**Dictionary**

**Dictionary** is a generic version of H-table. It’s a key and value pair.

Dictionary<string,int> MyDic = new Dictionary<string,int>();

MyDic["Zoro"] = 212;

**ContainsKey()\_\_**method takes key and returns true if key is in the dictionary, else false.

MyDic.ContainsKey(key)

**TryGetValue(key,out var result)\_**returns boolean and if value is in the dictionary the result will get its value.

MyDic.TryGetValue(key,out var result)

Each element in Dictionary is struct named **KeyValuePair.**

**public readonly struct KeyValuePair<TKey, TValue>**

Each pair will have their index in the sequence and because of that we can use the **ElementAt()** method which takes an index and returns a pair.

KeyValuePair<string,int> pair = newDic.ElementAt(1);

**Remove()** method takes a key and returns true if removal was successful, else false.

Console.WriteLine(newDic.Remove(Key));

**Singleton Designe Pattern**

**Singleton** class provides application wide common functionality. Singleton class has **private constructor** to not create an instance of this class outside.

We will have only **one static instance** of the singleton class as its private data member, which can be initialized directly and we should also have property to this private field to get it.

static readonly MySing? instance = new MySing();

public static MySing? Instance {get{

return instance;

}}

But there is a problem, as soon as this app runs our instance is created which is not memory efficient(if we have many of them). So we can laze load the initialization, till we try to actually use the instance, so we will initialize it in the getter.

static MySing? instance = null;

public static MySing? Instance {get{

if(instance == null)

instance = new MySing();

return instance;

}}

**In the case of singletons everyone has access to the same data with the same values.** If you have a little bit of data which may be need at some points(Not often) of your application than singleton is nice option.