C#

C# new features

|  |  |  |
| --- | --- | --- |
| C# 8.0 | C# 9.0 | C# 10.0 |
| Published during 09/21 | Published during 11/21 | Published during 12/21 |
| Supported on .NET Core 3.x and .NET standard 2.1 | Supported on .NET 5. | Supported on .NET 6 |
| Features |  |  |
| Readonly members |  |  |
| Default interface methods |  |  |
| Pattern matching enhancement   * Switch expressions * Property patterns * Tuple patterns * Positional patterns |  |  |
| Using declarations |  |  |
| Static local functions |  |  |
| Disposable ref structure |  |  |
| Nullable reference types |  |  |
| Asynchronous streams |  |  |
| Asynchronous disposable |  |  |
| Indices and ranges |  |  |
| Null-coalescing assignments |  |  |
| Unmanaged constructed types |  |  |
| Stackalloc in nested expressions |  |  |
| Enhancement of interpolated verbatim strings |  |  |

**Difference between .NET framework v/s .NET Core v/s .NET 5.0**

|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **.NET Framework** | **.NET Core** | **.Net 5.0** |
| Cross Platform | Only for windows | Works cross platform | .Net 5.0 is a **unified** platform which unifies all .Net runtime like .NET framework, .NET core and so on.  Developers no longer need to choose between .NET framework, .NET core and mono etc.  They provides common experiences to developers irrespective on which .NET version they are working. |
| Performance | Slow comparing to .NET core | Better |
| CLI tools | More IDE based | Full CLI command support. We can create project, build, run and much more from command prompt |
| Microservices support | No | Yes |
| Cloud | Works but only windows | Yes |
| Packaging | Packaged as one big framework | Delivered via modular via Nuget. We can download everything as a package from Nuget like SSDT, entityframe etc.. even dotnet runtime itself a package |
| Types of application supported? | Winforms, WPF, ASP.NET, Webforms, ASP.NET MVC 5 | MVC core 3.x |
| WCF, WPF, WWF | Yes | No |
| Mobile Support | No | Yes |
| Desktop based | WPF, Winfors | Nothing for desktop |

**Feature#1 > Init only setter (C#9.0)**

**Background**: Be default when we have property with getter and setters, the setter’s property will behave like a mutable. Means we can modify the property of value after object initializer. Let’s closely understand what is means.

|  |
| --- |
| WeatherForecast – Model class  public class WeatherForecast  {  public DateTime Date { get; set; }  public int TemperatureC { get; set; }  public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);  public string Summary { get; set; }  } |

|  |
| --- |
| Controller  public IEnumerable<WeatherForecast> Get()  {  var rng = new Random();  return Enumerable.Range(1, 5).Select(index =>  {  var wheather = new WeatherForecast  {  Date = DateTime.Now.AddDays(index),  TemperatureC = rng.Next(-20, 55),  Summary = Summaries[rng.Next(Summaries.Length)]  };  wheather.Summary = "Modified summary";  return wheather;  })  .ToArray();  } |

Output:

|  |
| --- |
| [      {          "date": "2022-01-18T22:36:45.1826042+05:30",          "temperatureC": 1,          "temperatureF": 33,          "summary": "Modified summary"      },      {          "date": "2022-01-19T22:36:45.1826192+05:30",          "temperatureC": 18,          "temperatureF": 64,          "summary": "Modified summary"      }  ] |

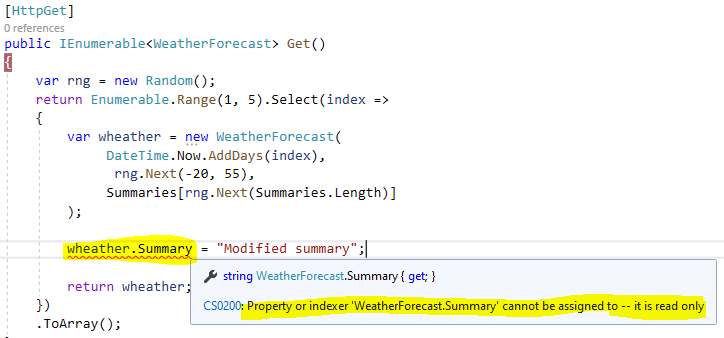
If we want to restrict the property value to modified after the object initialization, we follow some approaches like

* Approach#1: we can create a constructor and initialize property while creating and object, without using setter properties like below.

Updated model class: with default parameterized constructor and without setters.

|  |
| --- |
| public class WeatherForecast  {  public WeatherForecast(DateTime dt, int temp, string summ)  {  this.Date = dt;  this.TemperatureC = temp;  this.Summary = summ;  }  public DateTime Date { get; }  public int TemperatureC { get; }  public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);  public string Summary { get; }  } |

Controller code



With the code changes in controller, we can see that we are getting compile time error when we try to modify the property after initializer.

**Pros**: so, using parametrized constructor we can restrict the properties to assign with another values after object construction.

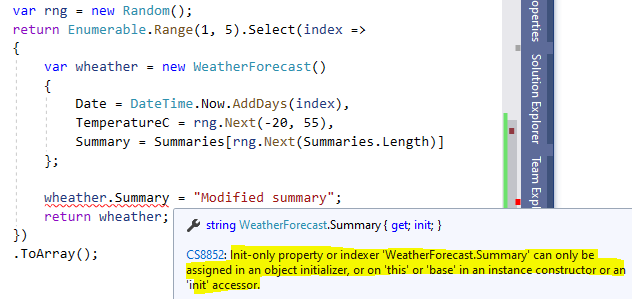
**Cons:** all the properties need to access in constructors which leads some additional boiler plate code.

**Approach#2**: we can achieve the same functionality, while using **init** only setter (C# 9.0 feature)

Updated model with **init** key word

|  |
| --- |
| public class WeatherForecast  {  public DateTime Date { get; init; }  public int TemperatureC { get; init; }  public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);  public string Summary { get; init; }  } |

**Updated controller**



**Pros**: we can see that summary property is not accessible after object initializer. In this way, we make property as immutable.

Also, we don’t need to write a lot of boiler plate code to achieve this. Just introducing **init** in place of setter property will take care.

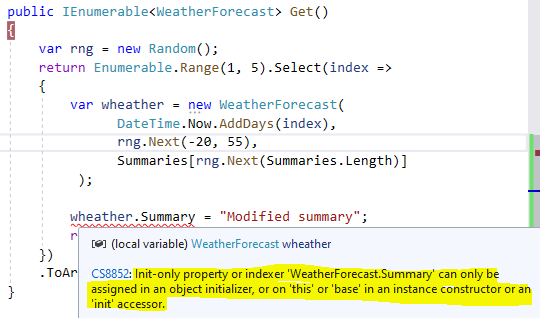
**Feature#2 > record type**

You use the record keyword to define a reference type that provide built-in functionality for encapsulating data. You can create record type with immutable properties by using positional parameters or standard property syntax.

|  |
| --- |
| Model.cs class  public record WeatherForecast(DateTime Date, int TemperatureC, string Summary)  {  public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);  } |

DateTime Date, int TemperatureC, string Summary: This parameters for **record** type acts as an **getters** and **init** by default.

**Updated controller**



In the above exception, we can notice that all the properties pass as part of object initializer. If we try to access any property outside will get and error.

Another feature with record type is “**with** expression”

* Using **with** expression we can create an object with existing object and modify few properties which are needed.
* Which means, only the modified properties value will change other property value will remains same

**Demonstration**

|  |
| --- |
| var rng = new Random();  return Enumerable.Range(1, 5).Select(index =>  {  var wheather = new WeatherForecast(  DateTime.Now.AddDays(index),  rng.Next(-20, 55),  Summaries[rng.Next(Summaries.Length)]  );  var newWheather = wheather **with** { TemperatureC = 97 };  Debug.WriteLine($"Existing Wheather summary: {wheather.Summary} and temperature: {wheather.TemperatureC}");  Debug.WriteLine($"New Wheather summary: {newWheather.Summary} and temperature: {newWheather.TemperatureC}");  return wheather;  })  .ToArray(); |

Output

|  |
| --- |
| New Wheather summary: Freezing and temperature: 97  Existing Wheather summary: Cool and temperature: 54  New Wheather summary: Cool and temperature: 97  Existing Wheather summary: Sweltering and temperature: 22  New Wheather summary: Sweltering and temperature: 97  Existing Wheather summary: Mild and temperature: 9 |

We can observe that, only the temperature property of new object is update, other properties values are carry from existing object.

**Feature#3 Top Level statements**

Top-level statements remove unnecessary ceremony from many applications. Consider the following Hello world program.

**Traditional Approach without Top level statements**

|  |
| --- |
| using System;  namespace CoreProject  {  public class Program  {  public static void Main(string[] args)  {  Console.WriteLine("Hello World..!!");  }  }  } |

In the above program we can see, just to write a single line of code “Hello World” we need to write a lot of boilerplate code like main method and all.

We can re-write the same code, with only one like, which is a feature of C#9.0. (removing all the boilerplate code)

|  |
| --- |
| System.Console.WriteLine("Hello World..!!"); |

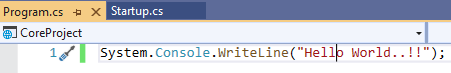
The above line of code will work fine without main method.

**Limitations**

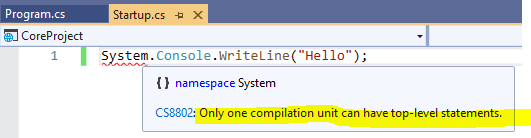
* Only one file in your application may use top level statement.

If we try to use top level statements in multiple files, will get compile time error.

Program.cs file



Setup.cs file



Pros:

* Top level statements enable a scrip like experience for experimentation purpose like what jupyter notebook provides
* Top level statements are great for small console programs and utilities. Azure function is a ideal use case of top-level programming.

Cons:

* Per application we can have only one top level statement.

**Feature#4 Pattern Matching enhancements (C#9.0)**

C# 9 includes new pattern matching improvements:

* ***Type patterns*** match a variable is a type
* ***Parenthesized patterns*** enforce or emphasize the precedence of pattern combinations
* ***Conjunctive and patterns*** require both patterns to match
* ***Disjunctive or patterns*** require either pattern to match
* ***Negated not patterns*** require that a pattern doesn't match
* ***Relational patterns*** require the input be less than, greater than, less than or equal, or greater than or equal to a given constant.

|  |
| --- |
| public static bool IsLetter(this char c) =>  c is >= 'a' and <= 'z' or >= 'A' and <= 'Z'; |

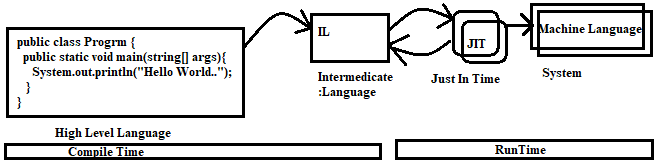
|  |
| --- |
| public static bool IsLetterOrSeparator(this char c) =>  c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ','; |

**IL (Intermediate Language) vs JIT(Just In Time) compiler**

As we know machines will understand only Assembly language. So here is the flow of our .net program.

IL code is partially compiled code

JIT compiled IL code to Machine Language



Purpose of IL and why can’t we directly compile .Net program to Machine Language?

* The runtime environment and development environment can be different. So, depending on the runtime environment JIT compile the best code as per that environment.

**CLR – Common Language Runtime**

CLR does many things behind the scenes. But some the unique responsibilities are

* CLR invokes JIT to compile to IL Code
* Clears unused objects by using Garbage collector

**Managed v/s Un Managed Code**

**Managed** - Code that run under CLR execution environment is called as managed code

**Un-Managed –** unmanaged code executes outside CLR boundary. Unmanaged code is nothing, but code written in C++, VB6, C++ etc.

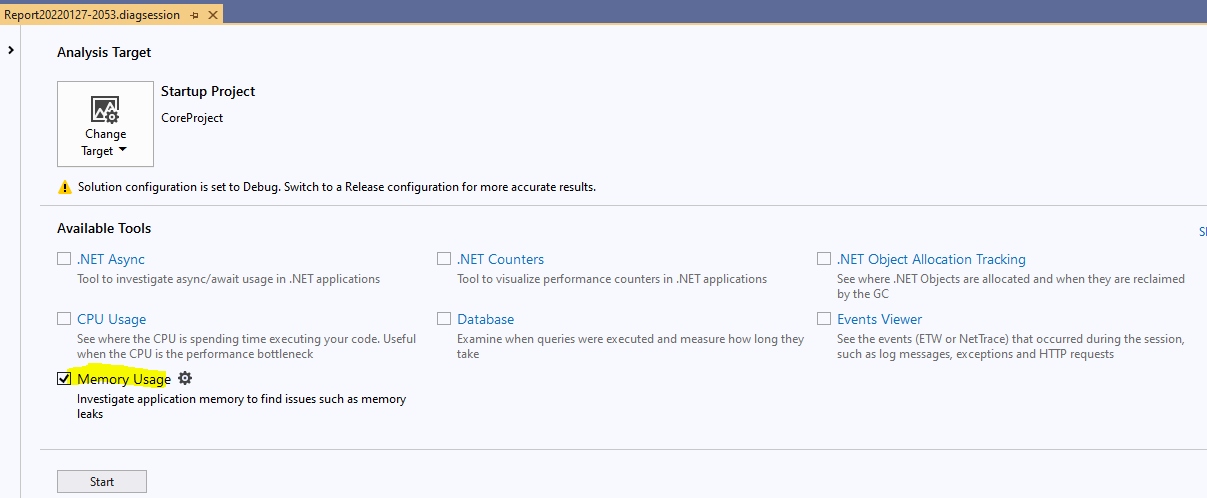
Unmanaged code has their own environment in which the code runs, and it is completely outside the controls of CLR.

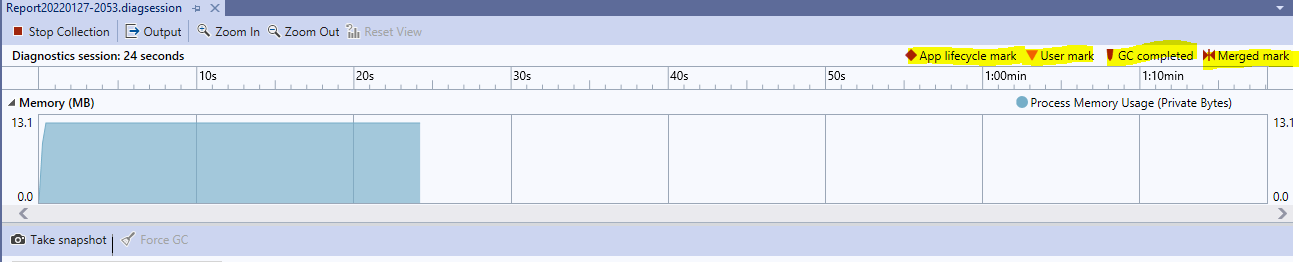
Unmanaged code executes under the supervision of Operating System (COM component, file management etc)

**Garbage Collection and its importance**

Garbage collector is a background process which cleans **unused** **managed** Heap resources.

We can visualize the Garbage collector from Performance Profiler (Debug > Performance Profiler). This preview feature is available from .net 4.5 or greater version





Can garbage collector collects/claim unmanaged resources?

* No, garbage collector not collected unmanaged resources which are created outside of CLR.

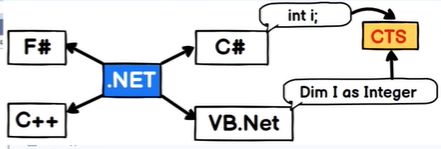
**CTS (Common Type System)**

CTS ensures that data types defined in two different languages get compiled into common data types.

With reference to below screen shot, we can see

* In C# we declare [int I = 0;] as integer
* In VB.NET we declare as [dim I as Integer]

When DotNet runtime is running, CTS will ensure the datatype get compiled as Int 32 regardless of language.



**CLS – Common Language Specification**

CLS is a specification or set of rules or guidelines. When any .NET programming languages adheres(follows) to this set of rules it can be consumed by any programming language following .Net specifications.

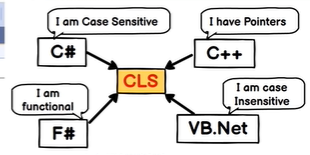
e.g)

> In C#, variable declaration is case sensitive (that is int x = 0; int X=0; are two different int variable)

> In VB.NET, variable is not case sensitive ((that is int x = 0; int X=0; is not valid and be compile time error)

Another example may be in c++ pointers are valid but in C# pointers are not valid.

To overcome all this. CLS is a guideline if it is followed, we can use language interop ability, that is we can use another programming language in some other program. (like using C# code along with VB.net)



Note:

**CTS** deals with different **data** types of the languages

**CLS** deals with different **behaviour** of the languages.

**Stack v/s Heap**

Stack and Heap both are the memory types in an application.

* Stacks stored the Primitive data types like int, bool, float etc
* Heap stores the non-primitive types like objects, strings etc.
* In stack, both variable and values are stored in the same place
* In Heap, the variable & pointer will be stored in stack and value will store in Heap memory.

**Value types v/s Reference Type**

Values types contains actual data (as in case of stack both variable and data stores at same place) while reference types contain pointers and the pointer points to the actual data.

Value types are stored on stack while reference type stored on heap.

Value types are the data types like int, bool, double etc

Reference types are all objects.

**Boxing v/s Un boxing**

Boxing > when value types move to reference type

Un Boxing > when reference type moves to value type

|  |
| --- |
| Int x = 1;  Object y = x; // Boxing: Value type of reference type [implicit casting]  Int z = (int) y; // Unboxing: reference type to value type [explicit casting] |

**Consequences**: boxing and unboxing will lead to performance consequent.

[implicit casting] – converting lower datatype to higher data type (int to object in above case)

[explicit casting] – Converting higher datatype to lower data type (object to int in above case). It will result in data loss (e.g converting float to int will result in losing decimal points)

**Array vs ArrayList**

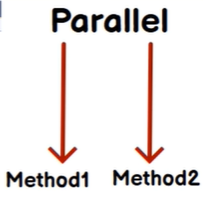
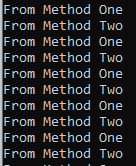
|  |  |  |
| --- | --- | --- |
| Features | Array | ArrayList |
| Fixed Length | Yes | No (flexible) |
| Strongly Typed | Yes | No |
| Performance | Better than array list | Slower because of boxing and unboxing |

|  |
| --- |
| Int [] arr = {1,2,3,4,5}; // fixed; strongly typed (case hold only int in this case)  ArrayList arrList = new ArrayList();  arrList.Add(1);  arrList.Add(‘L’);  arrList.Add(“sai”); // size not fixed; not strongly typed. |

**Thread (Multi-threading) in C#**

* Thread or multi-threading in c# help us to execute the code parallelly.
* we can implement Thread using **System.Threading** namespace.

|  |
| --- |
| static void Main(string[] args)  {  Console.WriteLine("Hello World!");  Thread t1 = new Thread(MethodOne);  t1.Start();  Thread t2 = new Thread(MethodTwo);  t2.Start();  }  public static void MethodOne()  {  for (int i = 0; i < 1000; i++)  {  Thread.Sleep(500);  Console.WriteLine("From Method One ");  }  }  public static void MethodTwo()  {  for (int i = 0; i < 1000; i++)  {  Thread.Sleep(500);  Console.WriteLine("From Method Two ");  }  } |



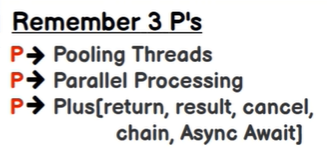
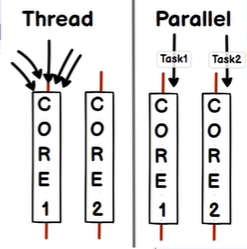
**Thread v/s TPL (Task Parallel Library)**

* we can achieve the same thing with Task like thread/multithreading. But Task will have some more features.
* Task are from **System.Threading.Task** namespace.

|  |
| --- |
| static void Main(string[] args)  {  Console.WriteLine("Hello World!");  Task t1 = new Task(MethodOne);  t1.Start();  Task t2 = new Task(MethodTwo);  t2.Start();  } |

Difference between **Thread** & **TPL**

* Task will do the Parallel Processing. Which means if an operation is carry out with Task. It will do the parallel processing on the **processor** and divide the **load** on the Processor **Cores** **parallelly**.
* Whereas a thread will try to load on the single processor core. (if we want to achieve same Parallel processing with Thread, we need to implement lot of Process Code)
* Task will utilize hardware properly
* Thread will have the CPU affinity.
* Task can return the result, whereas Thread will not.
* Some of the more advantage of Threads as below (3 P’s)



Note:

* For the new project or new implementation it always recommended to use Task as it is a wrapper build on System.Threading.Task namespace.
* If we are supporting any legacy system where we don’t have support for Task(usually .net legacy frameworks) we can use Thread.

# Ref v/s out keywords

**Background**: by using the return type we can return only one value at a time. What if we need to return multiple values?

Out: using out keyword we can return multiple values

|  |
| --- |
| static void Main(string[] args)  {  int add ; // valid in case of out  int sub ; // valid in case of out  MyMath(20, 10, out add, out sub);  Console.WriteLine($"Addition : {add} \t Substration : {sub}");  }  public static void MyMath(int x, int y, out int add, out int sub)  {  add = x + y;  sub = x - y;  } |

Ref: using ref keyword also we can return multiple values but for ref variable. We must initialize with values. Whereas in case of out initializing value is optional

Int add ; // invalid for ref

Int sub ; // invalid for ref

Int add = 0 ; // valid for ref

Int sub = 0; // valid for ref

**Delegates**: Delegate is a pointer to a function and very useful as call-backs to communicates between threads.

**Event:** Event are encapsulation over delegates

**Abstract Class v/s Interface:** Abstract class is a half-defined parent class whereas interface is a contract.

Abstract class is inherited whereas interface need to be implemented.

**OOPs – Object Oriented Programming**

**OOPS** – object-oriented programming helps us to think in real world object.

Like we have an Employee, and an employee will have a name, Id, department, address etc. putting this thing in practical as.

|  |
| --- |
| public class Employee  {  public string EmployeeID { get; set; }  public string Name { get; set; }  public Department Department { get; set; }  }  public class Department  {  public string DepartmentID { get; set; }  public string Name { get; set; }  } |

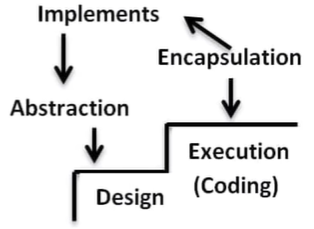
**OOPs Pillars**

1. Abstraction – show only what is necessary
2. Polymorphism – object can act differently under different conditions
3. Inheritance -- Parent child relationship
4. Encapsulation -- hide complexity

**Class & Object**: A class is a **type, blueprint** and object are an instance of the class

**Abstraction v/s Encapsulation**

|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| Show only what is necessary | Hide complexity |
| Abstraction happens at **Design** phase | Encapsulation comes during **development** phase |
| It deals with what has to be shown, in the below Employee class, properties Name and ValidateEmployee() is exposed for public access. | It deals with hiding complexity, in the below Employee Class, ValidationBy\_Aadhar(), ValidationBy\_PAN() uses to validate employee. Whereas this method details not exposed outside for public access |
|  | In real world, Encapsulation implements Abstractions. |



|  |
| --- |
| public class Employee  {  public string EmployeeID { get; set; }  public string Name { get; set; }  public void ValidateEmployee()  public virtual void ValidateEmployee()  {  ValidationBy\_Aadhar();  ValidationBy\_PAN();  }  private void ValidationBy\_Aadhar()  {  }  private void ValidationBy\_PAN()  {  }  public class Manager : Employee  {  public override void ValidateEmployee()  {  //base.ValidateEmployee();  // write your own logic here  }  } |

**Virtual keyword/ Overridding**

Virtual keyword helps us to define some logic in the parent class which can be overriding in the child class.

In the above program, Manager class override ValdiateEmployee() method of base class Employee.

**Method Overloading** Same method name with different signature (different parameter types)

|  |  |
| --- | --- |
| Method Overloading | Method Overriding |
| Method with same name and with different parameter types | Using virtual keyword and overriding in child class |
| Achieve within the same class | Achieve in child class by overriding parent class behaviour |

**Polymorphism**

Its an ability of an object to act differently under different conditions.

We can achieve polymorphism **only** with **Inheritance (parent child relation is must)**

**Static (compile time) v/s Dynamic (Runtime) polymorphism**

Statis polymorphism is implemented by method overloading, which is occurs at compile time.

Dynamic polymorphism is implemented by method overriding, which is occurs at runtime.

|  |
| --- |
| public class Employee  {  public string EmployeeID { get; set; }  public string Name { get; set; }  public virtual void ValidateEmployee()  {  }  // Overloading method ValidateEmployee()  private void ValidateEmployee(string aadhar)  {  }  // Overloading method ValidateEmployee()  private void ValidateEmployee(string aadar,string PAN)  {  }  } |

|  |
| --- |
| static void Main(string[] args)  {  Employee emp = new Employee();  emp.ValidateEmployee();    // Overriding ValidateEmployee Runtime  emp = new Manager();  emp.ValidateEmployee();  // Overriding ValidateEmployee Runtime  emp = new Admin();  emp.ValidateEmployee();  }  public class Manager : Employee  {  }  public class Admin : Employee  {  } |

**Operator Overloading**

Operator overloading help us to redefine additional functionalities for plus, minus, multi operators.

Example

Var data = 1+ 2; // output is 3 here + acts like an additional operator

Var data = “Sai” + “ Hari” // output is Sai Hari, here + acts like an string concatenation.

Here we can see that, by default c# comes with the features of operator overloading.

**Is Custom Operator Overloading being possible?**

Yes, we can have custom operator overloading. We can achieve additional functionality of operator overloading by using the key work “**operator**”

|  |
| --- |
| public class MyOperator  {  private int v;  public MyOperator(int v)  {  this.v = v;  }  public static MyOperator operator +(MyOperator op1, MyOperator op2)  {  return new MyOperator(op1.v + op2.v);  }  }  static void Main(string[] args)  {  var op1 = new MyOperator(10);  var op2 = new MyOperator(20);  var op3 = op1 + op2;  Console.WriteLine(op3);  } |

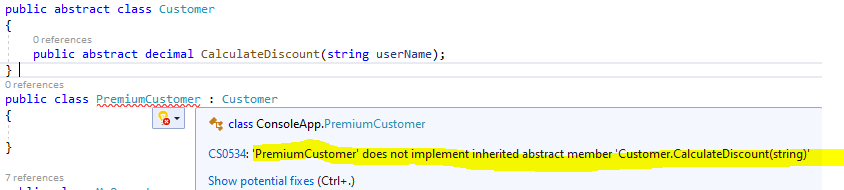
**Abstract Class**

An abstract class is **partially** defined parent class.

* Abstract methods in an abstract class are default **virtual**.
* Can we create a instance of an abstract class? **No, we can only inherit**
* Is it compulsory to implement Abstract methods of Abstract class? **Yes**, its compulsory need to implement. The compile will give error saying partial methods need to be implemented (refer below screenshot).

Why can’t a base class acts like an abstract class?

A base class cannot be acts like an abstract class. Because for a base class we need to write some sort of hack code to indicate we need to override and write some sort of dummy logic.



|  |
| --- |
| public abstract class Customer  {  public abstract decimal CalculateDiscount(string userName);  }  public class PremiumCustomer : Customer  {  public override decimal CalculateDiscount(string userName)  {  throw new NotImplementedException();  }  }  static void Main(string[] args)  {  PremiumCustomer premiumCustomer = new PremiumCustomer();  premiumCustomer.CalculateDiscount("sai");  } |

**Interface**

* An interface is a **Contract**
* By having contract, we have better control on impact analysis, change management and braking control.
* **Multiple inheritance** helps to add new methods without effecting the old interface.
* Can we write logic in an interface? **No**, we cannot write logic in interface, we can only have signatures.
* Can we have private methods or properties in interface? **No**, we cannot have private access modifier for properties, methods etc. all the members of interface are public by default.
* Can we create instance of interface? **No**
* Can we do multiple inheritance with abstract class? **No**.

**if we want to change (adding new method signature or property) an interface, what is the best practice?**

* We need try to change or add new method signature or properties in existing interface. If we do so, it may result in lot of areas in application we the class implementing an interface.
* So, the best approach to change or extent the interface is, we need to create a new interface (ICustomerWithInterest) and implement existing interface (ICutomer). May will lead multiple inheritance. (remember, multiple inheritance is possible with interface).

**Interface Segregation Principle(ISP)**

***“Clients should not be forced to depend upon the interfaces that they do not use”***

Assume we have an Interface ICustomer and a class Customer Implements ICustomer

|  |
| --- |
| interface ICustomer  {  public string Name { get; set; }  public decimal Amount { get; set; }  decimal CalculateInterest();  }  public class Customer : ICustomer  {  public string Name { get; set; }  public decimal Amount { get; set; }  public decimal CalculateInterest()  {  return 0;  }  } |

Let’s assume, we got requirement to add/enhance the interface by adding **CalculateDiscount()** method.

* We should not add **CalculateDiscount()** method signature in existing ICustomer interface, because doing so, will result in breaking code contract with existing classes with implements ICustomer interface.

To achieve this goal, we need to create a separate interface and inherit existing ICustomer interface as below.

|  |
| --- |
| interface ICustomerWithDiscount : ICustomer  {  decimal CalculateDiscount();  } |

To implement this behaviour in class, we can implement ICustomerWithDiscount along with ICustomer.

|  |
| --- |
| public class Customer : ICustomer,ICustomerWithDiscount  {  public string Name { get; set; }  public decimal Amount { get; set; }  public decimal CalculateDiscount()  {  return 0;  }  public decimal CalculateInterest()  {  return 0;  }  }  static void Main(string[] args)  {  ICustomer customer = new Customer(); // runtime polymorphism  customer.CalculateInterest();  ICustomerWithDiscount customer1 = new Customer();runtime polymorphism  customer1.CalculateDiscount();  } |

Check list

* We achieve contract by defining Interface.
* We achieve runtime polymorphism
* We achieve multiple inheritance
* We achieve Interface segregation principle (ISP) by not forcing clients to CalculatDiscount(), as we define this method in separate interface.

**Abstract class v/s Interface**

|  |  |
| --- | --- |
| **Interface** | **Abstract Class** |
| Interface is a **contract,** contains only property declarations and method signatures (without any implementation) | Abstract classes are half defined parent classes |
| Interface is like **planning abstraction**, which comes while designing phase. | Abstraction is like **implementation phase**, which is implementation. Which is **sharing common logic in child class**. |
| Interfaces are **implemented** | Abstract classes are **inherited** |
|  |  |

Flow is something like, **Interface** comes during the **Planning phase**, then all the common logic moves to **Abstract class** and then will have **fully implemented Concrete class**.