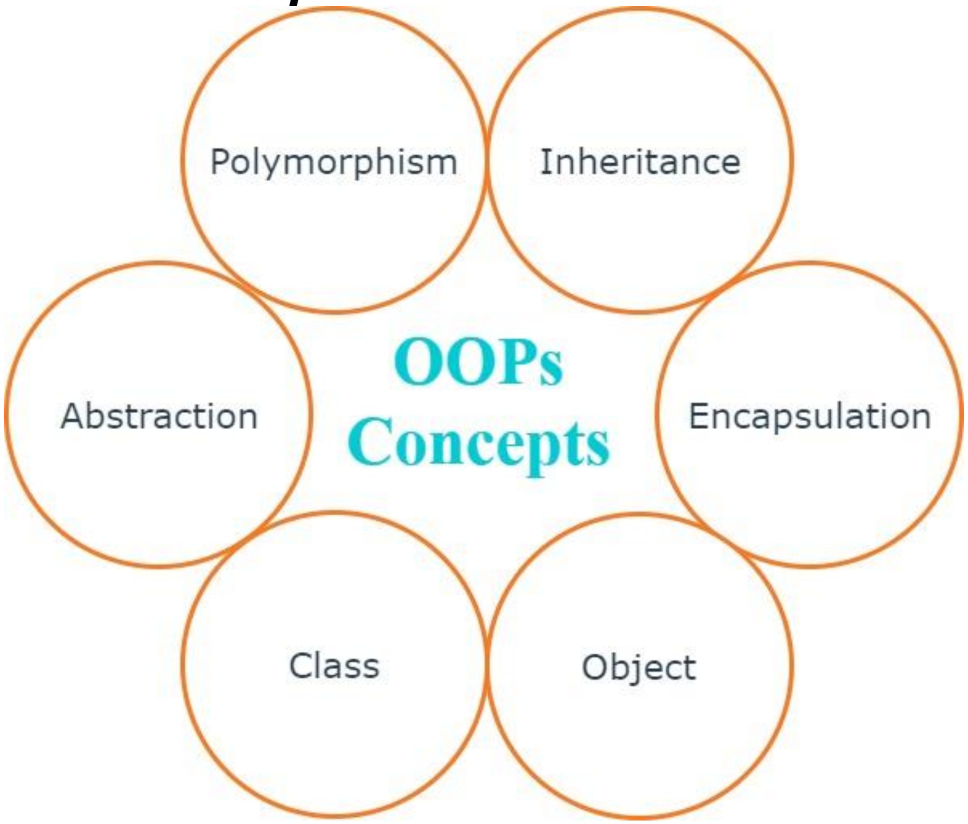
- ***Important Note C++***
- ***C++ is not a fully object-oriented programming language.***
- ***It is a hybrid language that combines object-oriented*** **and** ***Linear programming techniques.***
- ***Overall, C++ is a powerful and versatile programming language. However, it is not a fully object-oriented programming language.***

- ***Important Note C#***

- ***C# Modern Programming language.***
- ***C# is Cross Platform.***
- ***C# Is Managed Code (GC).***
- ***C# MultiPorpose Programing Language.***
- ***C# Is Fully object-oriented programming language.***
- ***C# is a statically typed language.***
- *"That means that the data type of a variable must be declared before it can be use".*
- ***C# Is Strongly Typed.***
- *"That meaning variables and objects must have a specific type declared at compile-time. "*
- ***C# includes Garbage Collection.***
- *"That meaning automatic memory management through a garbage collector"*
- ***C# Is Platform Independence(Cross Platform.)***

- ***OOP Concepts***



- ***main goals of OOP***
- ***Encapsulation***
- *Encapsulation is the process of hiding the implementation details of an object from the outside world.*
- ***Abstraction***
- *Abstraction is the process of representing an object in terms of its essential features.*
- ***Polymorphism***
- *Polymorphism is the ability of an object to take on different forms.*
- *allows you to create code that is more flexible and adaptable.*

- ***benefits of using OOP***
- ***Reusability****: OOP makes it easier to reuse code.*
- ***Maintainability****: OOP makes it easier to maintain code.*
- ***Flexibility****: OOP makes it easier to create flexible code.*

- ***Variables***
- ***Datatype must be***

  -> Size

  -> Validation

  -> Operation
- ***Value Datatype Vs Reference Datatype .***

- ***Note: Nullable Type***

  => *int? X=null;*

- ***Note : Casting***

```
            ////Convert Same Data Type (int , lonng Decimal)

         //---------- implicit -> easy--------
                    int x = 1200;
                    long y = x;

         //------------------------------------

         //----------Exceplicit----------------
                    long a = 54545454545;
            int b = (int)a;// Casting Operation =>  //Over Flow Canbe Occure.

       //-----------------------------------
        // checked block Used To Check if over Flow occures Throw Exp..
            checked
            {
                long m = 54545454545;
                int n = (int)m;
            }
```

```
        ////Convert Different Data Type(String -> int || double)

       //Helper Class
            string str= "125445";
            int x =Convert.ToInt32(str);
            int y =int.Parse(str);
              //---------
            int A = 254588556;
            string txt=A.ToString();

       //User Define Casting (Not Now)
```

- ***Common Type System*** ***Value Type Vs Reference Type***

Feature	Value Type	Reference Type
Storage	• Stack	• Heap but (Reference in Stack),must Use 'new Key Word'
Deletion	• When the variable is deleted	• When the reference variable is deleted
Copy	• A copy of the value is made	• Only the reference is copied
Equality	• Two value types are equal if they have the same value	• Two reference types are equal if they point to the same object
Passing to Methods	• The value is passed by value	• The reference is passed by reference
Boxing	• Not required	• Required when a value type is used in a context that requires a reference type
Unboxing	• Not required	• Required when a reference type is used in a context that requires a value type
Example	Integer, Float, Boolean, Char	• Object, Array, class, String.

Note:

-> reference Data Type is Complex Data Type

```
class student
 {
    public int Id;
    public string Name;
 }
```

Note: *Address Vs Reference*

-> Address: *The address refers to the specific memory location where the data is stored.*

-> References: *in C# simply store memory addresses, and they are not involved in encryption directly.*

- ***Control Statement(Done)***
- ***Conditional Statement***
- *If*
- *If else*
- *If ,else if , else*
- *Switch*
- ***Looping Statement***
- *Loop.*
- *While.*
- *Do While.*
- *Foreach.*
- ***Array***
- ***Declaration and Initialization Arrays***
- ***DataType + [] + Arr_Name = new + DataType[Size];***
- *int[] Arr1 = new int[5];*
- *int[] Arr2 = new int[] { 1, 2, 3, 4, 5 };*
- *int[] Arr3 = { 1, 2, 3, 4, 5 };*
- *Can Use Same Structure Of Declaration and Initialization*
- *int[,] Arr2D = new int[3, 4] { { 1, 2, 3, 4 }, { 1, 2, 3, 4 }, { 1, 2, 3, 4 } };*
- ***Notes:***
- ***Fixed Size.***
- ***Same DataType.***
- ***Array Zero-based Indexing.***
- ***Directly Access By Index "Arr[0]".***
- ***Array class in the System namespace provides a number of methods for working with arrays. These methods include methods for creating, initializing, accessing, sorting, and searching arrays.***

///User Define DataType -> Struct , Class ,Enum, interface , Delegate, Record-> 'Complex DataType'.
/// any User Define DataType ...Define in NameSpace lVl.
////DataType -> Storage, Valdiation, operation
//// Value Type Fast Access Compare Between Reference DataType.

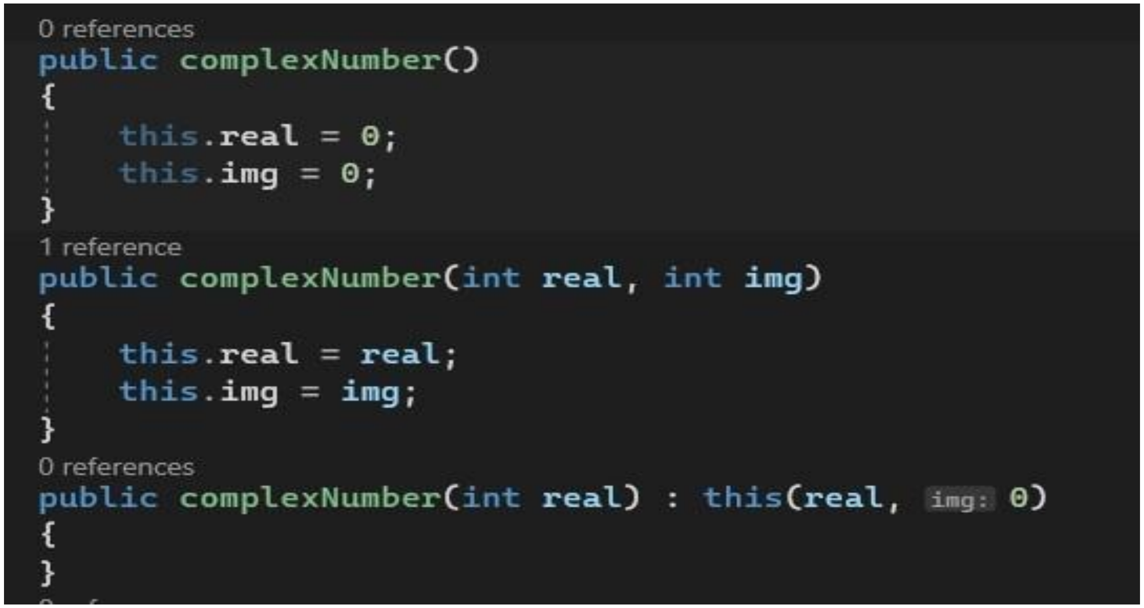
Struct	• Value Type
Class	• Reference Type
interface	• Reference Type
Enum	• Value Type
Delegate	• Reference Type
Record	• Reference Type

Type	Description
Struct	A struct is a lightweight data type that can be used to store data. Structs are similar to classes, but they do not support inheritance or polymorphism.
Class	A class is a data type that can be used to store data and define methods. Classes can be inherited from other classes, and they can be used to create objects.
Enum	An enum is a type that represents a set of named constants. Enums are often used to represent values that can have a limited number of possible values, such as the days of the week or the months of the year.
Interface	An interface is a contract that defines a set of methods that a class must implement. Interfaces are often used to decouple different parts of an application.
Delegate	A delegate is a type that represents a method call. Delegates are often used to implement events or callbacks.
Record	A record is a new type in C# 9 that is similar to a struct, but it supports inheritance and polymorphism. Records are often used to represent data that is related to each other.

- *Boxing refers to the process of converting a value type to an object type, and unboxing is the reverse process.*

- *Struct Notes*
- *Struct is a value type.*
- *Limited Inheritance: They cannot be derived from other structs or classes, and they cannot be used as a base for other types.*
- *Default Constructor: By default, structs have an implicit parameterless constructor provided by the compiler.*
- *(in Case If Create constructor must initialize all the fields of the struct.)*
- *Structs are commonly used for representing small, simple, and immutable data structures .*
- *Structs are value types, and they are not subject to boxing and unboxing like reference types.*
- *Size Limitation: The size of a struct is limited to a maximum of 16 bytes in C#.(else Use Class's)*
- *Struct can implement interfaces.*
- *Constructors in Struct*
- *By default, structs have an implicit parameterless constructor.*
- *struct constructor, you are responsible for explicitly initializing all the fields of the struct.*
- *can overload constructors in structs by providing different parameter lists.*

- ***Each struct constructor is specific to the struct type and is automatically invoked when a struct instance is created.***
- ***Constructor Chaining (Calling Another Constructor)***



```
0 references
public complexNumber()
{
    this.real = 0;
    this.img = 0;
}
1 reference
public complexNumber(int real, int img)
{
    this.real = real;
    this.img = img;
}
0 references
public complexNumber(int real) : this(real, img: 0)
{
}
```

- ***Class Notes***
- ***Constructors are special methods in a class that are called when an object of that class is created using the new keyword.***
- ***Constructors have the same name as the class and do not have a return type.***
- ***Constructors are used to initialize the initial state of an object by setting the values of its fields or performing other initialization tasks.***
- ***Constructors can be overloaded, allowing for multiple constructors with different parameter lists .***

- ***Class Relationships - "Is-A" and "Has-A"***
- ***"Is-A" Relationship (Inheritance)***
- ***"Has-A" relationship represents a composition or aggregation association between classes, indicating that a class has another class as a part or member.***
- ***Composition:***

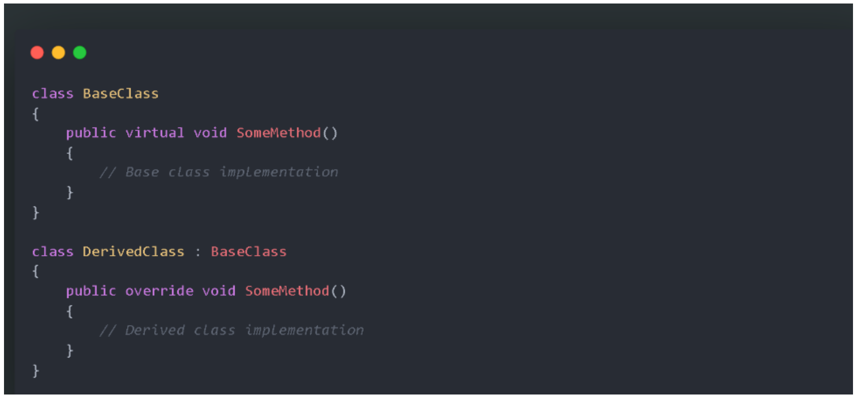
->Composition implies that the contained object cannot exist independently of the container object.

- ***Aggregation:***

->Aggregation signifies that the contained object can exist independently of the container object.

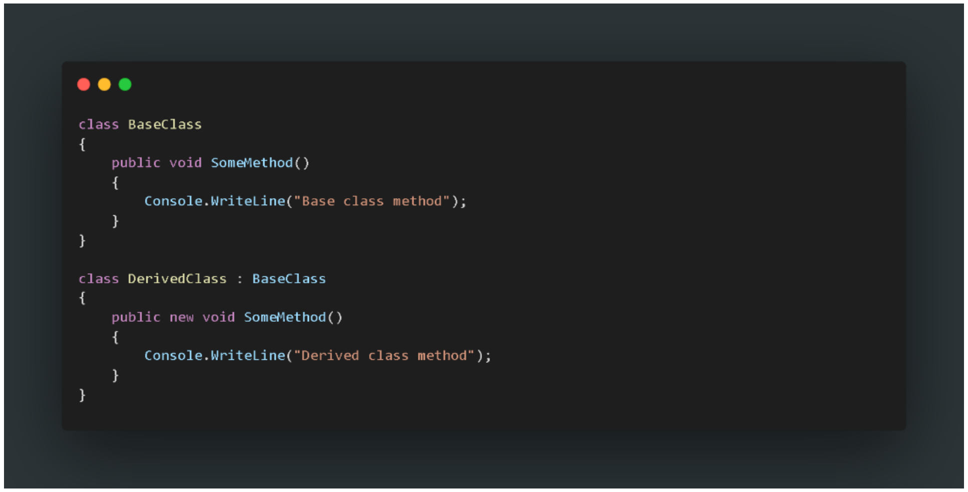
Access Modifier	Accessibility
Public	• *Everywhere*
Private	• *Only within the class*
Protected	• *Within the class and any subclasses*
Internal	• *Within the assembly*
Protected Internal	• *Within the class, subclasses, and the same assembly*

- ***Inheritance.***
- *Inheritance creates a hierarchical relationship between classes, where a derived class (also known as a child class or subclass) inherits members from a base class (also known as a parent class or superclass).*
- *The child class can access the public and protected members (fields, properties, and methods) of the base class.*
- ***NOTES***
  - ***Method Overriding***
- *allows the child class to provide a different implementation for a method that is already defined in the base class.*
- *use the override keyword in the child class method declaration.*
- *The base class method must be marked as virtual or abstract to allow overriding.*

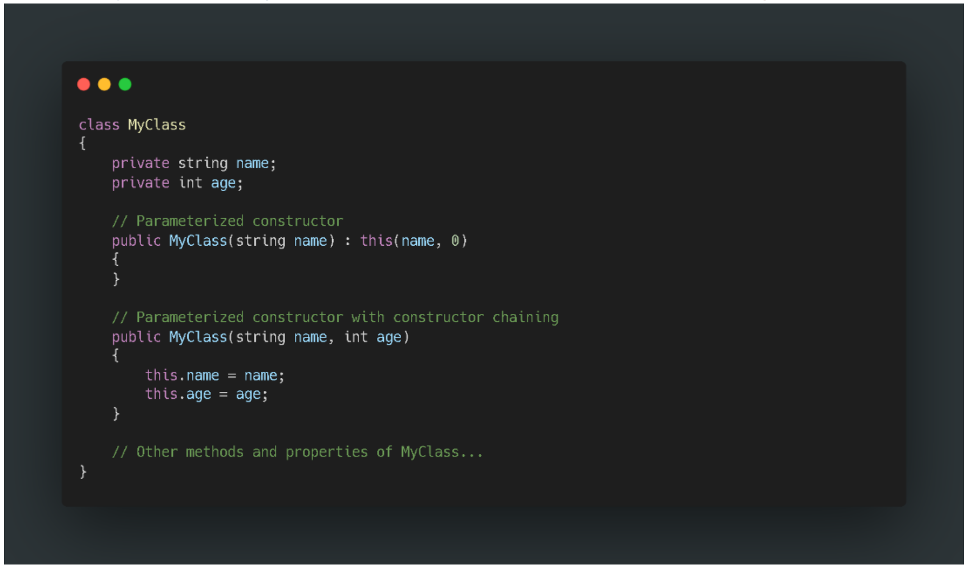


```
class BaseClass
{
    public virtual void SomeMethod()
    {
        // Base class implementation
    }
}

class DerivedClass : BaseClass
{
    public override void SomeMethod()
    {
        // Derived class implementation
    }
}
```

2.Method Hiding

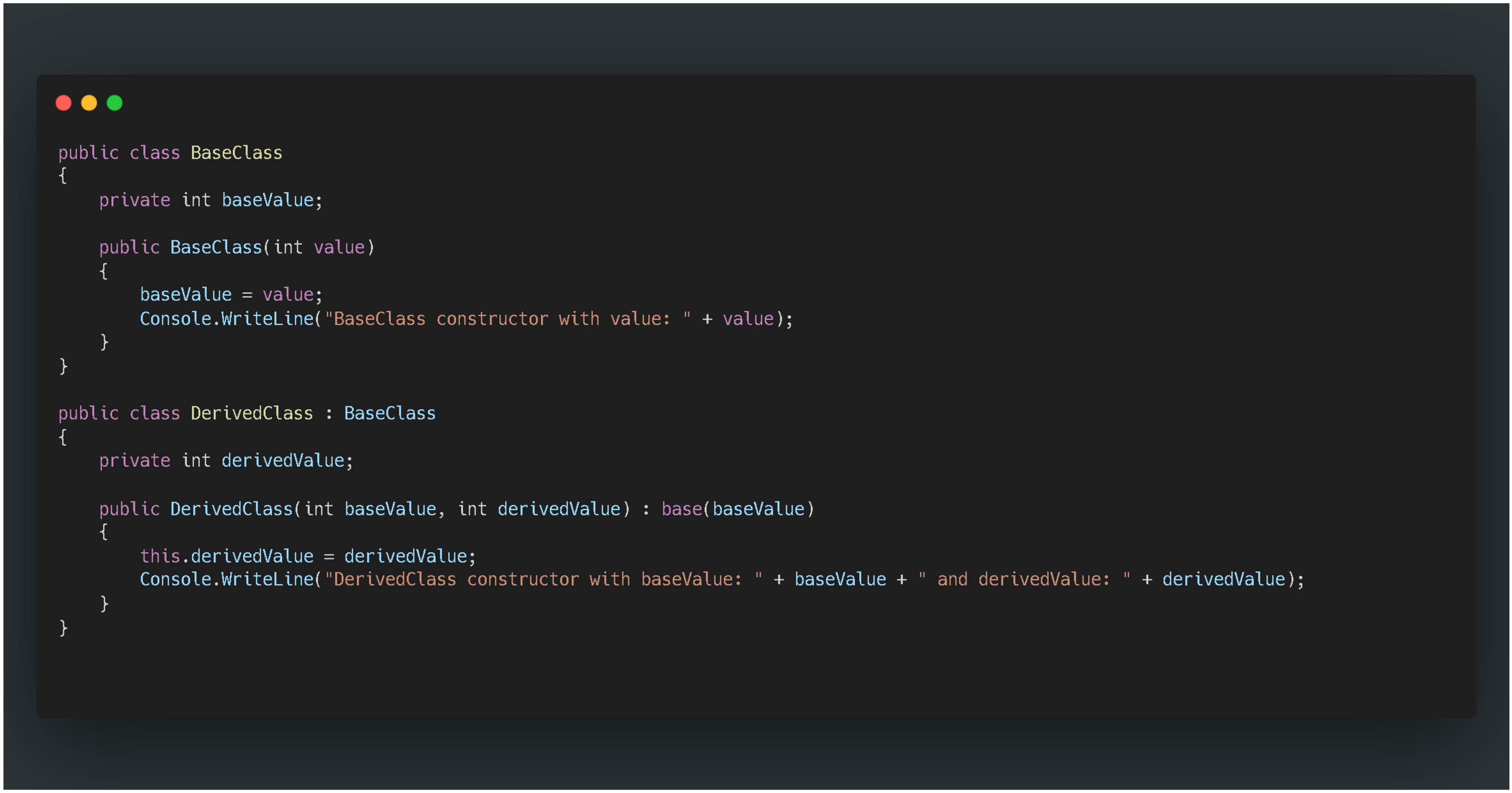
- *If a child class has a member with the same name as a member in the base class, the derived class member can hide the base class member using the new keyword.*
- *create a new member in the derived class without any relationship to the base class member.*
- 

```
class BaseClass
{
    public void SomeMethod()
    {
        Console.WriteLine("Base class method");
    }
}

class DerivedClass : BaseClass
{
    public new void SomeMethod()
    {
        Console.WriteLine("Derived class method");
    }
}
```

- ***Base Class Constructors:***
- *When a child class is instantiated, the base class constructor is called first to initialize the inherited members.*
- *If the base class has multiple constructors, the derived class constructor can use the base keyword to explicitly invoke a specific base class constructor.*
- ***Constructor chaining in Same Class.***
- *allows one constructor to call another constructor within the same class or in the base class.*
- *To call another constructor from within a constructor, you use the this keyword .*
- *this keyword refers to the current instance of the class.*
- 

```
class MyClass
{
    private string name;
    private int age;

    // Parameterized constructor
    public MyClass(string name) : this(name, 0)
    {
    }

    // Parameterized constructor with constructor chaining
    public MyClass(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Other methods and properties of MyClass...
}
```

- ***Chaining to Base Class Constructor.***
- *When a child class constructor is called, it can chain to a constructor in the base class using the base keyword.*
- *The base class constructor is called before the child class constructor initializes its own members.*

- 

```
public class BaseClass
{
    private int baseValue;

    public BaseClass(int value)
    {
        baseValue = value;
        Console.WriteLine("BaseClass constructor with value: " + value);
    }
}

public class DerivedClass : BaseClass
{
    private int derivedValue;

    public DerivedClass(int baseValue, int derivedValue) : base(baseValue)
    {
        this.derivedValue = derivedValue;
        Console.WriteLine("DerivedClass constructor with baseValue: " + baseValue + " and derivedValue: " + derivedValue);
    }
}
```

○ ***sealed class.***

- *class that cannot be inherited by other classes.*
- *cannot serve as a base class for other classes.*
- *Once a class is sealed, all its methods are implicitly sealed and cannot be further overridden.*
- *It allows for better encapsulation.*

- ***Sealed Function.***

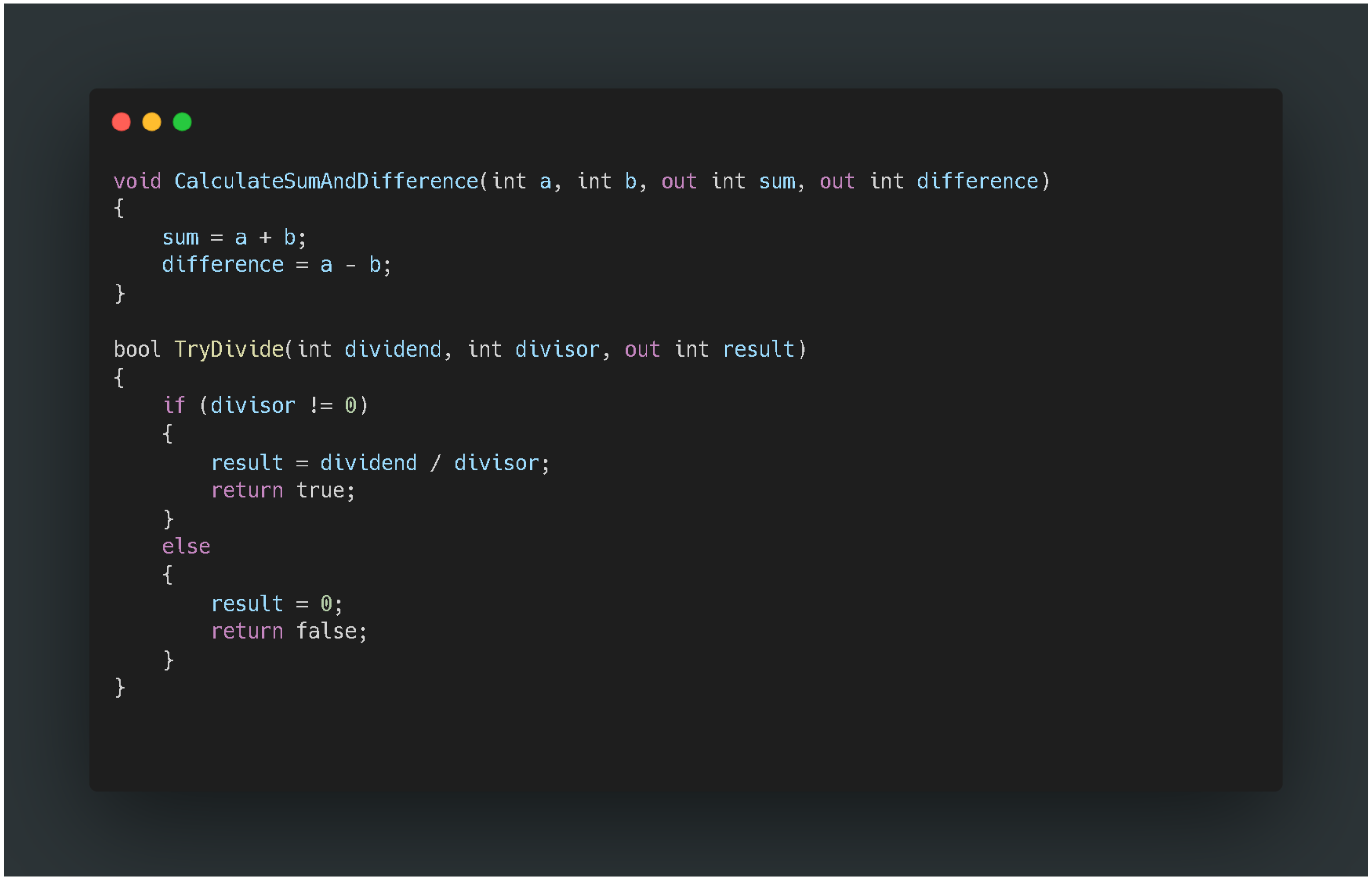
- *sealed method is a method that cannot be overridden by derived classes.(Stop For Extension Of Virtually).*
- *Once a method is sealed in a base class, it cannot be overridden by derived classes.*
- *sealed method in the base class is the final implementation.*
- *To sealed a method, it must be declared as virtual or override in the base class.*
- ***NOTES***
- ***Static variables***
- *Variable shared among all instances of a class.*
- *must be initialized at the time of declaration or within a static constructor.*
- *Static variables are accessible within the entire class and can be accessed using the class name followed by the variable name.ClassName.StaticVariable)*
- *Static variables are initialized before any instance of the class is created*
- *Access to static variables from multiple threads can cause race conditions and concurrency issues.*
- *Must Use synchronization mechanisms, such as locks or other thread-safe constructs, should be used when accessing or modifying static variables in a multi-threaded environment.*
- *Static variables are useful for maintaining shared state or storing data that needs to be shared across all instances of a class.*
- ***Static Method.***
- *static method is a method that belongs to the class itself rather than to instances of the class.*
- *They can be accessed directly using the class name followed by the method name.*
- *static methods do not require an instance of the class to be called.*
- *called directly using the class name without creating an object of the class.[ClassName.StaticMethod();]*
- *Static methods can access other static members (variables, methods) within the same class without the need for an instance reference.*
- *They cannot be marked as virtual, abstract, or override.*
- *Static methods cannot be marked as async or await.*

- ***Static Class***
- *Static classes cannot be instantiated (Sealed Behavior).*
- *Static classes cannot create Object using the new keyword because they are implicitly sealed.*
- *Static classes can only contain static members, including static methods, properties, fields, and events*
- *Static classes are defined at the namespace level and are accessible throughout the same namespace.*
- *Static classes are commonly used to group together utility functions and helper methods that provide common functionality without requiring object-specific data.*
- ***Static constructor***
- *static constructor (also known as a type initializer) is a special constructor that initializes the static members of a class.*
- *A static constructor does not have any parameters and is declared using the static keyword followed by the class name.*
- *has no access modifiers, return type, or method name.*
- *It is called only once during the lifetime of the program.*
- *Only one static constructor is allowed per class.*
- *Static constructors are typically used to initialize static members, including static variables, static properties, and other static data structures.*

- *passing parameters to functions in C#.*

- **Call (Value Type by Value).**
  - ✓ *By default, parameters in C# are passed by value.*
  - ✓ *When passing by value, a copy of the value is passed to the function.*
  - ✓ *inside the function do not affect the original value in the calling code.*
- **Call (Value Type by Reference).**
  - ✓ *To pass parameters by reference, you can use the `ref` Keyword.*
  - ✓ *When passing by reference, the memory address (reference) of the variable.*
  - ✓ *Allowing changes to affect the original value.*
- **Call (Reference Type by Value).**
  - ✓ *When you pass a reference type by value to a method:*
  - ✓ *' copy of the reference (memory address) is passed, not the actual object'.*
  - ✓ *The method receives a copy of the reference, allowing access to the same object in memory.*
  - ✓ *Can be Access Data in This Reference(Applied Some Operation), Such '++';*
- **Call (Reference Type by Reference).**
  - ✓ *When you pass a reference type by reference to a method.*
  - ✓ *'you are passing a reference to the original reference variable, not just a copy of the reference.'.*
  - ✓ *modifications made to the reference inside the method will affect the original reference.*
  - ✓ *Such As 'Swap 2 Array'.*
- **Call by 'out'.**
  - ✓ *The out parameter modifier is similar to the ref modifier.*
  - ✓ *The out modifier is used when a method needs to return multiple values.*
  - ✓ *must be assigned a value inside the method before it returns.*
  - ✓ *useful when a method needs to modify the value of a parameter and return it as an output.*

- ***Note: Differences between 'out' and 'ref':***
    - **In out must be '** *assigned a value inside the method before it return*"



```
void CalculateSumAndDifference(int a, int b, out int sum, out int difference)
{
    sum = a + b;
    difference = a - b;
}

bool TryDivide(int dividend, int divisor, out int result)
{
    if (divisor != 0)
    {
        result = dividend / divisor;
        return true;
    }
    else
    {
        result = 0;
        return false;
    }
}
```

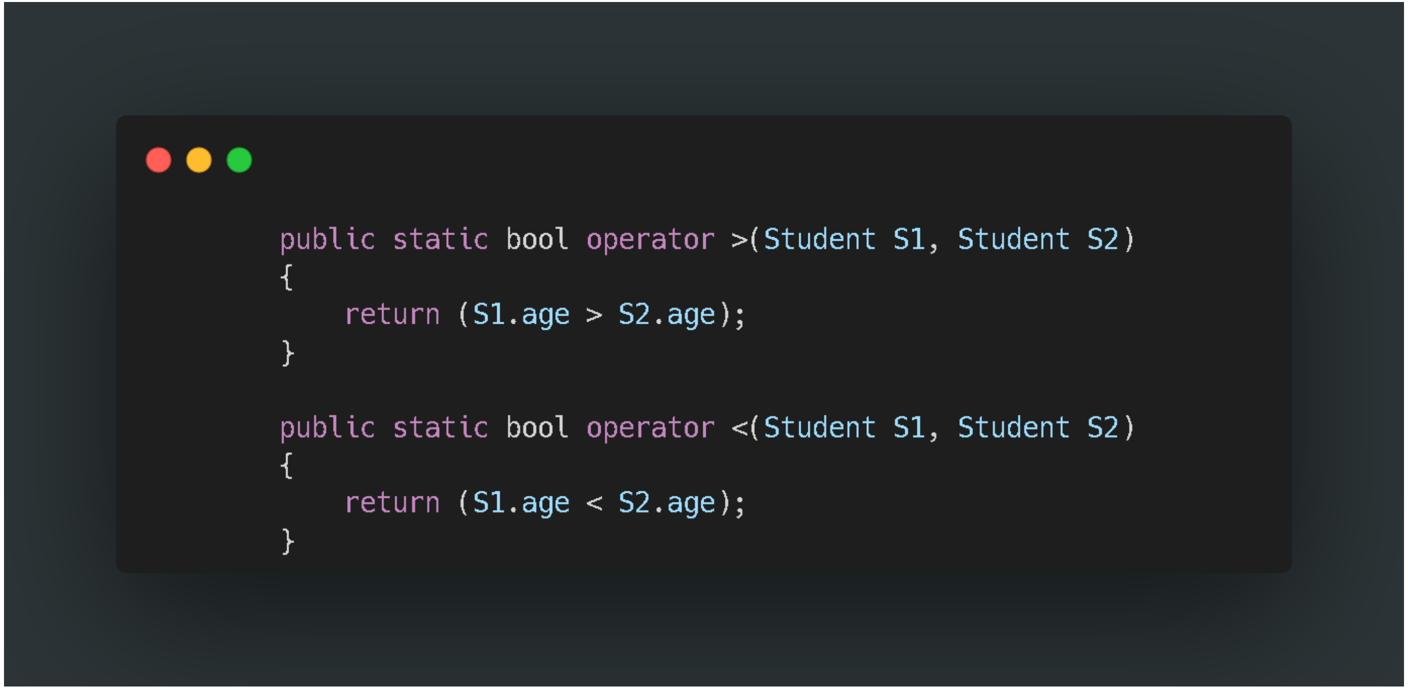
- *Operator overloading*

- *in C# allows you to redefine the behavior of operators such as +, -, *, /, ==, !=, <, >, etc.*

Or (custom behaviors for operators).

- *The overloaded operator method must be declared as `public` and `static`.*
- *Must defined inside a class using the `operator` keyword followed by the operator being overloaded.*

Example:



```
public static bool operator >(Student S1, Student S2)
{
    return (S1.age > S2.age);
}

public static bool operator <(Student S1, Student S2)
{
    return (S1.age < S2.age);
}
```

- ***User Define Casting (custom type conversion).***

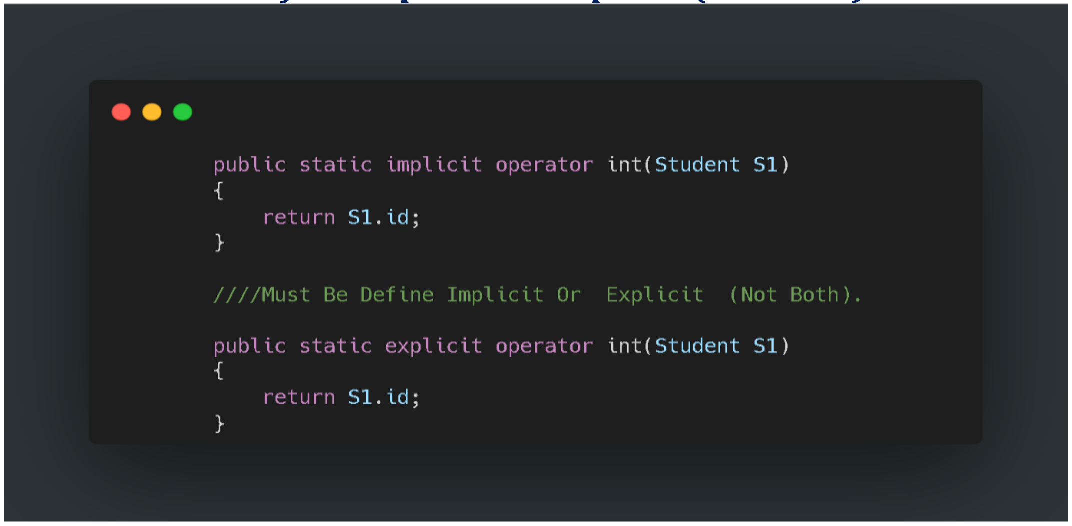
✓ ***Allows you to define how objects of a user-defined type are converted to other types.***

✓ ***you can define explicit and implicit conversion operators for your classes.***

✓ ***Implicit Conversion: Implicit conversion allows automatic type conversion from one type to another without explicit casting syntax.***

✓ ***Explicit Conversion: Explicit conversion requires explicit casting syntax.***

✓ ***Must Be Define Implicit Or Explicit (Not Both).***

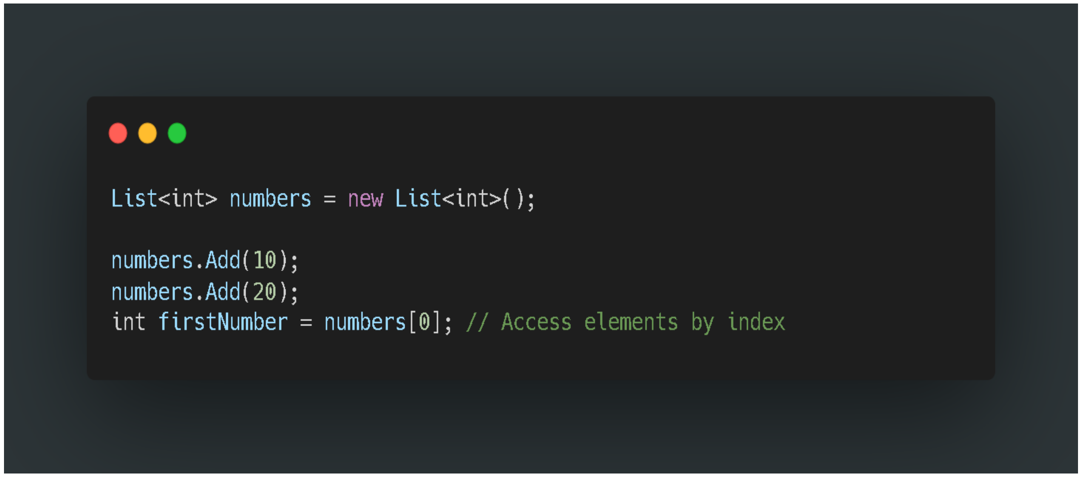


```
public static implicit operator int(Student S1)
{
    return S1.id;
}

////Must Be Define Implicit Or  Explicit  (Not Both).

public static explicit operator int(Student S1)
{
    return S1.id;
}
```

- ***Access Modifier:***

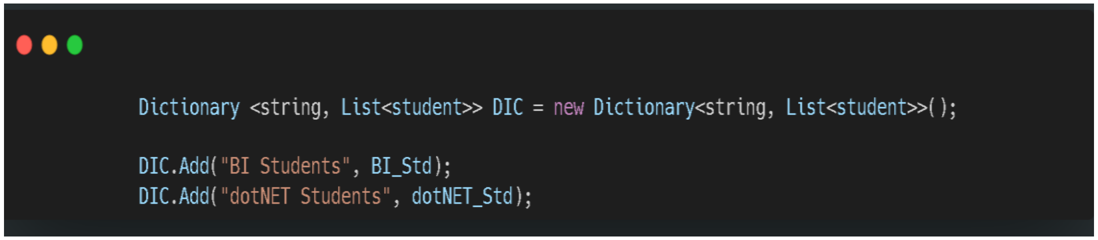
Access Modifier	***Accessibility***
public	➢ *Everywhere*
private	➢ *Only within the declaring class or struct*
protected	➢ *Within the declaring class or struct and its subclasses*
internal	➢ *Within the assembly that declares it and other assemblies in the same .NET Framework version*
protected internal	➢ *Within the declaring class or struct, its subclasses, and other assemblies in the same .NET Framework version.*

- ***Generic In C#***
    - *Generics are a way to create reusable code that can work with different types of data.*
    - *Generic types are declared using angle brackets (<>).*
    - *can be any type that is supported by the .NET.*
    - *Generic types can be used to create classes, structures, interfaces, and methods.*
    - *Generics provide compile-time type checking, ensuring type safety and reducing runtime errors.*
    - *Code Reusability: Generics enable you to write generic algorithms and data structures that can be used with different data types, promoting code reusability.*
    - *improved performance: Generics avoid boxing and unboxing operations for value types.*
- ***Generic.Collection.***
    - **List<T>**
    - *List<T> is a dynamic array that can grow or shrink in size.*
    - *It allows you to store elements of a specific data type T.*
    - *Provides methods to add, remove, access elements, and more.*



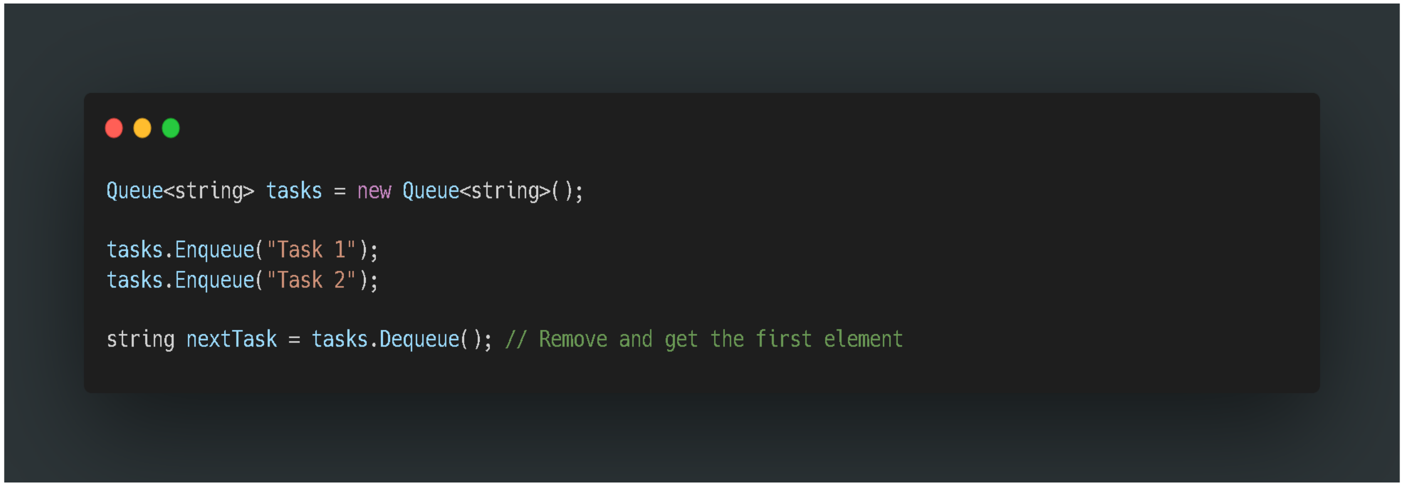
```
List<int> numbers = new List<int>();

numbers.Add(10);
numbers.Add(20);
int firstNumber = numbers[0]; // Access elements by index
```

- **Dictionary<TKey, TValue>**
- *Dictionary<TKey, TValue> is a collection of key-value pairs.*
- *It allows you to store elements with unique keys of type TKey and corresponding values of type TValue.*
- *Provides methods to add, remove, access elements, and more using the keys.*



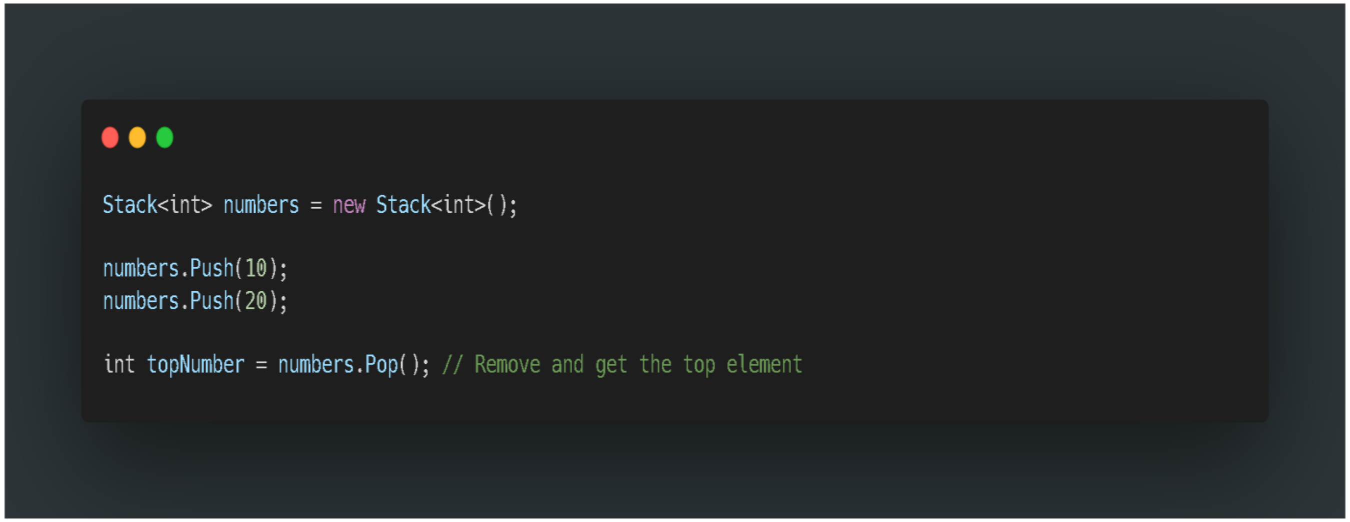
```
Dictionary <string, List<student>> DIC = new Dictionary<string, List<student>>();

DIC.Add("BI Students", BI_Std);
DIC.Add("dotNET Students", dotNET_Std);
```

- **Queue<T>**
- *Queue<T> is a first-in-first-out (FIFO) collection.*
- *where elements are added at the end and removed from the beginning..*
- *It allows you to store elements of type T.*
- *Provides methods to enqueue (add), dequeue (remove), and access elements.*



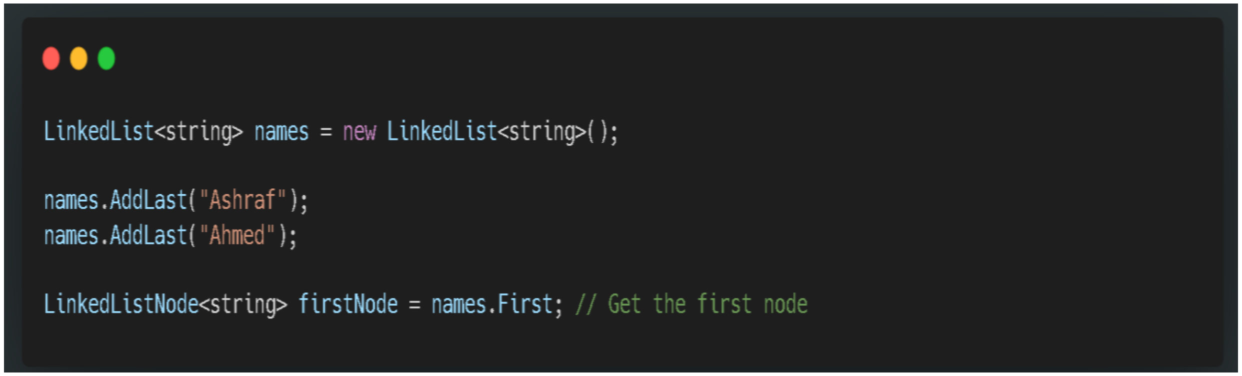
```
Queue<string> tasks = new Queue<string>();

tasks.Enqueue("Task 1");
tasks.Enqueue("Task 2");

string nextTask = tasks.Dequeue(); // Remove and get the first element
```

- **Stack<T>**
- *Stack<T> is a last-in-first-out (LIFO) collection.*
- *where elements are added and removed from the top (end) of the stack.*
- *It allows you to store elements of type T.*
- *Provides methods to push (add), pop (remove), and access elements.*



```
Stack<int> numbers = new Stack<int>();

numbers.Push(10);
numbers.Push(20);

int topNumber = numbers.Pop(); // Remove and get the top element
```

- **LinkedList<T>**
- *LinkedList<T> is a doubly linked list collection that allows you to store elements of type T.*
- *Provides methods to add, remove, and access elements efficiently.*



```
LinkedList<string> names = new LinkedList<string>();

names.AddLast("Ashraf");
names.AddLast("Ahmed");

LinkedListNode<string> firstNode = names.First; // Get the first node
```

Feature	*List*	*LinkedList*
Pros	• *Efficient for accessing elements by index*	• *Efficient for inserting and removing elements in the middle*
Cons	• *Inefficient for inserting and removing elements in the middle*	• *Less efficient for accessing elements by index*

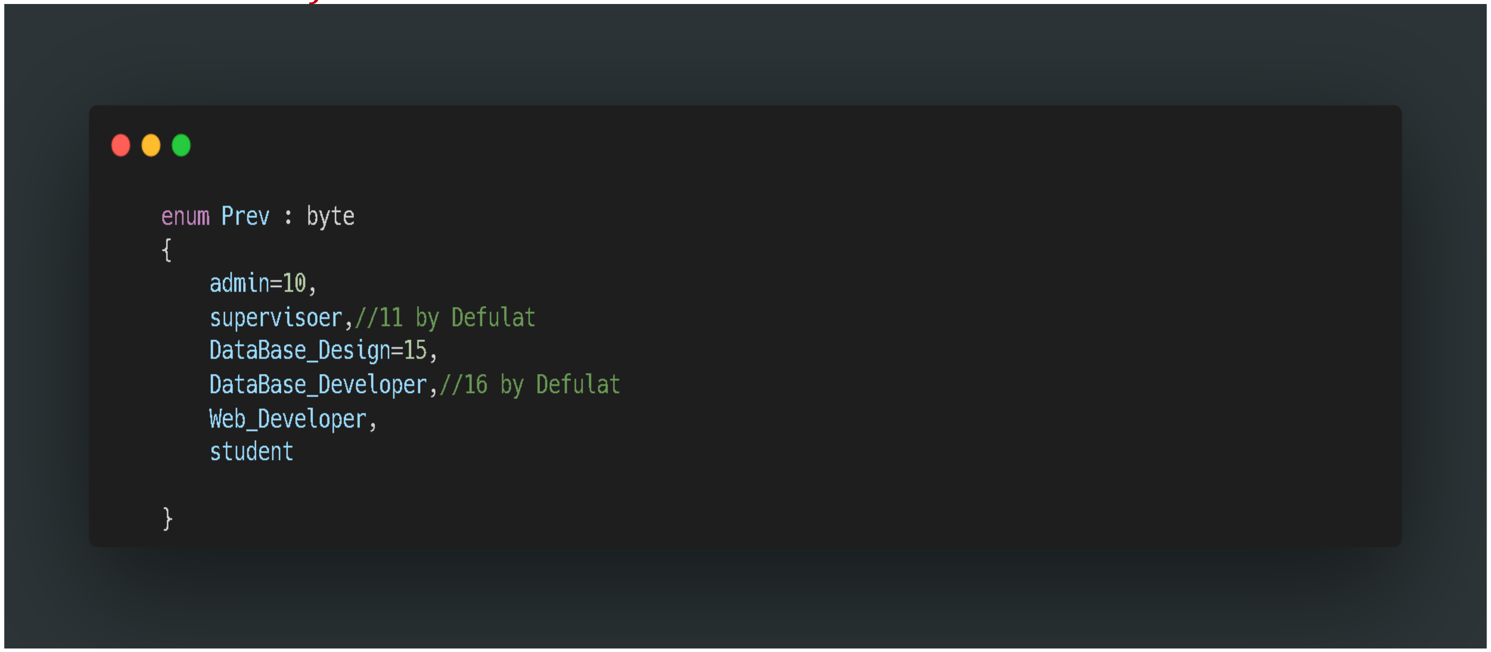
Feature	*List*	*LinkedList*
Storage	• *Dynamic array*	• *Doubly linked list*
Access by index	• *Efficient*	• *Inefficient*
Insert/remove in the middle	• *Inefficient*	• *Efficient*
Memory usage	• *More efficient*	• *Less efficient*

`Object` class:

  - *The `Object` class is the base class for all other classes in C#.*
  - *Every class implicitly or explicitly inherits from the `Object` class.*
  - *The default implementation of `<u>Equals</u>` in the `Object` class performs reference equality comparison.*
  - *Boxing is the process of converting a value type to a reference type (`Object`).*
  - *unboxing is the reverse process of converting a reference type (`Object`) To value type.*

- *Common Members:*
  - *`Equals`: Compares two objects for value equality.*
  - *`GetHashCode`: Returns a hash code value for the object.*
  - *`ToString`: Returns a string representation of the object.*
  - *`GetType`: Returns the runtime type of the object.*

- ***Enumerations (Enum)***
    - *Enums are a type of value type in C# (means that they are stored on the stack).*
    - *Enums are immutable -> means that their values cannot be changed after they are created.*
    - *Enum members are named constants that represent specific values within the enum type.*
    - *(To make your code more readable and maintainable.)*
    - *Enums can be used in switch statements.*
    - *Enums can be explicitly converted to strings (useful for displaying the value of an enum).*
    - *Enums can be used to implement enumerations.*
    - *Inheritance from*



```
enum Prev : byte
{
    admin=10,
    supervisoer,//11 by Defulat
    DataBase_Design=15,
    DataBase_Developer,//16 by Defulat
    Web_Developer,
    student

}
```

- ***Interfaces in C#***
    - *Allow you to specify a set of method and property signatures without providing implementation details.*
    - *Inside the interface, you can define method signatures, property declarations, events, and indexers, but you cannot provide implementation details.*
    - *Classes that implement an interface must provide implementations for all the members defined in the interface.*
    - *A class can implement multiple interfaces, allowing it to adhere to multiple contracts.*
    - *Interfaces can inherit from other interfaces.*
    - *Interfaces provide a way to achieve abstraction and polymorphism in C#.*
    - *Interfaces play a crucial role in achieving loose coupling between components in object-oriented programming.*
    - *good design principles like separation of concerns and facilitate code reuse and maintainability.*