Day 6

StringBuilder

```
// Stringbuilder
// Garbage Collector

string str = "hello";
str = "a1";
//str[5] = 'W';
str += " another thing attached";

// str = "hello another string attached";
str = str + " a3";

StringBuilder stringBuilder = new StringBuilder("Hello World");

stringBuilder[0] = 'W';
Console.WriteLine(stringBuilder);
```

access modifier

Summary table

Caller's location	public	protected internal	protected	internal	private protected	private
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

Search | Copy

assembly means project

private and getter and setter

```
1 reference
int Id { get; set; }
0 references
public string Name { get; set; }
0 references
public string Email { get; set; }
```

constructors

```
3 references
public class Employee
{
    //Constructors:
    /*
     are used to create an object of the class and initialize class member
    Default constructors do not take any paramters
    if We create any other constructor, we are replacing the default constructor
    constructors can be overloaded.
    constructors cannot be inherited so a constructor cannot be overridden.
    by default, derivded class constructor will make a call to the base class parameterless constructor
     */
    0 references
    public Employee()
    {

    }

    1 reference
    public Employee(int a)
    {

    }
```

inherit

```
2 references
public class FullTime : Employee
{
    0 references
    public FullTime()
    {

    }

    0 references
    public FullTime(int id, string name, string email)
    {
        Id = id;
        Name = name;
        Email = email;
    }


}
```

create object

```
//List
var emp = new Employee(3);
emp.Id = 4;

var fullEmp = new FullTime(1, "Sam", "Sam@Sam.com");

fullEmp.DoWork("Dishi");
```

abstraction

```
namespace ConsoleAppD2.Abstraction
{
    //Abstract Classes: Abstract classes cannot be instantiated but contrete classes can
    // as long as there is a single abstract method in the class, the class must be abstract
    // Abstract classes can contain concrete methods as well as abstract methods.

    0 references
    public abstract class Shapes
    {

        1 reference
        public int Id { get; set; }

        0 references
        public abstract void SomeMethod();

        0 references
        public int GetId()
        {
            return Id;
        }
    }
}
```

```
//Abstract Methods: It is a method that can be used in an abstract class
//IT does not have a body and the body is provided int derived class
```

```
0 references
public class Square : Shapes
{
    1 reference
    public override void SomeMethod()
    {
        Console.WriteLine("This is the body/content of the some method method");
    }
}
```

virtual keyword

```
//Abstract vs virtual:
//Abstract method doenst provide implementation and forced derived class to override
//Virtual gives the option of overriding.
```

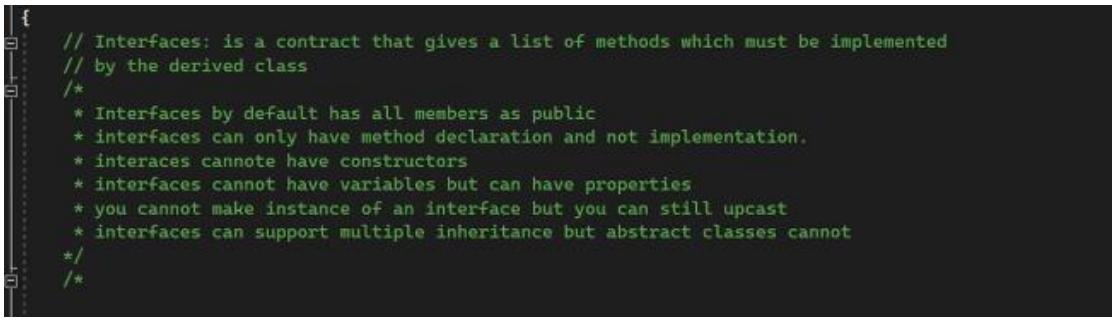
在 C#中，使用 `virtual` 关键字可以定义虚拟方法，这与普通方法有以下区别：

1. 可以在派生类中重写：虚拟方法可以在派生类中重写，子类可以根据需要提供自己的实现，而普通方法不能被子类重写。
2. 提供默认实现：虚拟方法可以在基类中提供默认实现，如果子类没有重写该方法，则将使用基类中的默认实现。普通方法不提供默认实现。
3. 运行时多态：虚拟方法支持运行时多态，即在运行时根据对象的实际类型调用相应的方法。普通方法在编译时就已经决定了调用哪个方法，无法实现运行时多态。

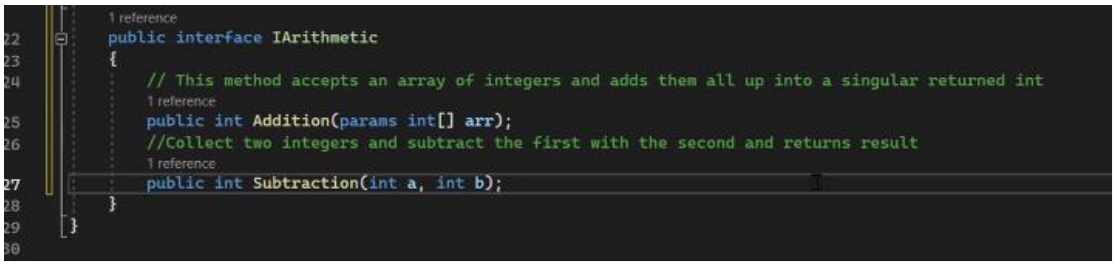
如果一个派生类重写了其基类中的非虚拟方法，则称为“隐藏”（Hiding）基类中的该方法。隐藏方法与重写虚拟方法的方式是不同的。如果派生类重写了基类

中的虚拟方法，那么在运行时将调用派生类中的实现，而如果派生类隐藏了基类中的非虚拟方法，则在编译时将始终调用隐藏的方法，不会调用基类中的实现。

interface

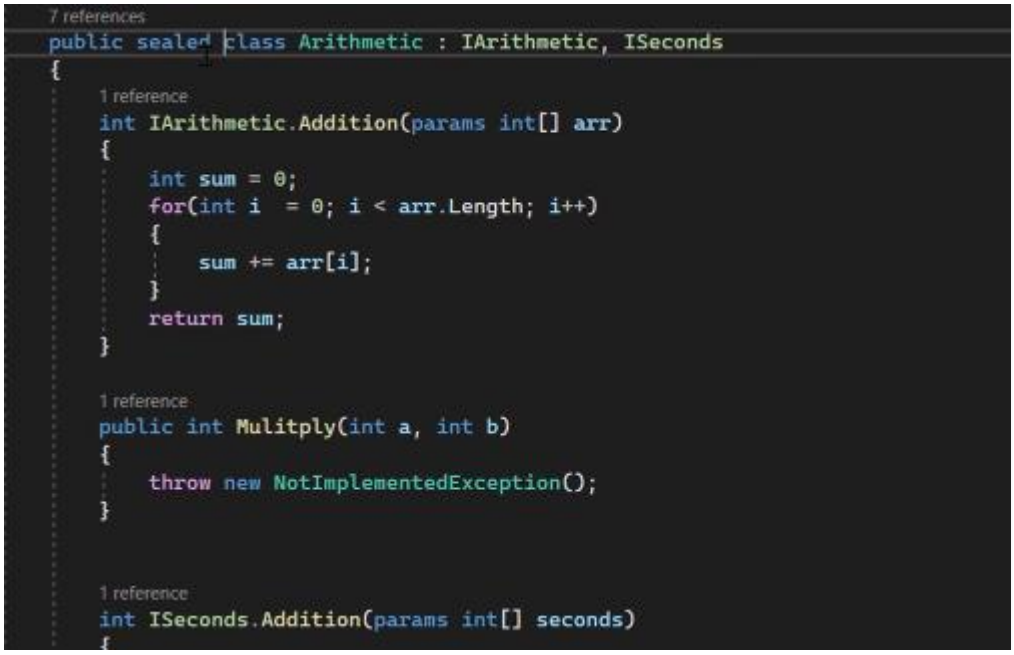


```
{
    // Interfaces: is a contract that gives a list of methods which must be implemented
    // by the derived class
    /*
     * Interfaces by default has all members as public
     * interfaces can only have method declaration and not implementation.
     * interaces cannote have constructors
     * interfaces cannot have variables but can have properties
     * you cannot make instance of an interface but you can still upcast
     * interfaces can support multiple inheritance but abstract classes cannot
    */
    /*
```



```
1 reference
public interface IArithmetic
{
    // This method accepts an array of integers and adds them all up into a singular returned int
    1 reference
    public int Addition(params int[] arr);
    //Collect two integers and subtract the first with the second and returns result
    1 reference
    public int Subtraction(int a, int b);
}
}
```

implement two interfaces with same function



```
7 references
public sealed class Arithmetic : IArithmetic, ISeconds
{
    1 reference
    int IArithmetic.Addition(params int[] arr)
    {
        int sum = 0;
        for(int i  = 0; i < arr.Length; i++)
        {
            sum += arr[i];
        }
        return sum;
    }

    1 reference
    public int Mulitply(int a, int b)
    {
        throw new NotImplementedException();
    }


    1 reference
    int ISeconds.Addition(params int[] seconds)
    {
```

boxing and unboxing. upcasting

```
//Boxing and unboxing
//Wrap a value type into a reference type
int box = 0;

object ob = (object)box; //heap

int unbox = (int)ob;

//upcasting
//Parent is limited to the declarations made in the parent class not the child class
Shapes shape = new Square();
var shapeSq = (Square)shape;
IArithmetic arithmetic = new Arithmetic();
```

hiding

如果一个派生类重写了其基类中的非虚拟方法，则称为“隐藏”（Hiding）基类中的该方法。隐藏方法与重写虚拟方法的方式是不同的。如果派生类重写了基类中的虚拟方法，那么在运行时将调用派生类中的实现，而如果派生类隐藏了基类中的非虚拟方法，则在编译时将始终调用隐藏的方法，不会调用基类中的实现。

```
Employee emp1 = new Employee();
Employee empfull = new FullTime();
FullTime fulFull = new FullTime();
Console.WriteLine("----------------------");
emp1.Hiding();
empfull.Hiding();
fulFull.Hiding();
```

```
----------------------
This is the hidden method
This is the hidden method
This is the hidden method In Fulltime
------------------------
```

factory design pattern

```
// Factory Design Pattern
0 references
public IArithmetic ReturnsArithObject(int i)
{
    switch (i)
    {
        case (1):
            return new AnotherArithmetic();
            break;
        default:
            return new Arithmetic();
            break;
    }
}
```

static keyword

```
// Static:
// Belongs to the class level meaning it becomes an available global instance
// We can access static methods without making an instance of these classes.
```

sealed class cannot be inherited

```
7 references
public sealed class Arithmetic : IArithmetic, ISeconds
{
    1 reference
```