نام و نام خانوادگی: علی مرادزاده

شماره دانشجویی: 40011541054027

دانشگاه: آزاد اسلامی واحد کرج

کلاس: روش پژوهش و ارائه 14:00

استاد: فرجادی نسب

منابع:

- Fundamentals of Computer Programming with C# Svetlin Nakov 2013 Edition
- Microsoft C# Documentation

فهرست

Programming Diagram…….3
Spaghetti Code……..……..…..3
Procedural Coding……...……4
Structural Programming……...……..5
Functional Programming(FP)……..……..6
Function Oriented Programming(FOP)……..7
Abstract Data Type(ADT)……...….9
Object Oriented Programming(OOP)……..…..10
Abstraction……..……..…16
Constructor……..……..……16
Destructor……..……...19
Encapsulation……..……21
Inheritance……..……..28
Polymorphism……..……...37
Examples……...46

Programming Paradigm:

در علوم کامپیوتر انواع برنامه نویسی وجود دارد که قبل از شی گرایی که اسمش object-based هم هست کد هایی از قبلی spaghetti coding, procedural coding و structural programming بود. جدول زیر انواع آنها را نشان میدهد:

Programming Paradigm:	
Imperative	**Declarative**
Procedural Coding Structured Programming Function Oriented Programming Object Oriented Programming Service Oriented Programming	Logic Programming Functional Programming

قصد نداریم در این ارائه عمیقا این روش های برنامه نویسی را پوشش دهیم و صرفا تعریفش میکنیم و مثالی میزنیم و سطحی ازش عبور میکنیم.

-Spaghetti Code:

در C# ما 5 تا حلقه داریم: for.1 while.2 do while.3 foreach.4 goto.5
در اکثر برنامه نویسی, goto به دلیل داشتن پتانسیل تبدیل شدن به حلقه ی بینهایت, زیاد توصیه نمیشود که برنامه نویسان از آن استفاده کنند. Spaghetti code هم به کدی گفته میشود که از پیچیدگی خاصی پیروی کند و بیش از حد از دستور goto استفاده شود, چرا که برای برنامه نویسان سخت است که برنامه ای را رفع اشکال کنند که زیادی از دستور goto استفاده میکند, چون در مواقعی ممکن است آنها خط را گم کنند. بخاطر همین ویژگی پیچیدگی و پریدن است که به آن spaghetti code میگویند. spaghetti code یک کد است و پارادایم نیست!. معمولا بدون تفکر و نقشه برنامه نویسی کردن منجر به تولید پیچیدگی و بعد spaghetti code میشود. منظور از نقشه داشتن برای برنامه نویسی کردن یعنی تمام حالات ممکنه را در نظر گرفتن است.
تصاویر زیر توصیحی و مثالی از spaghetti code هستند:

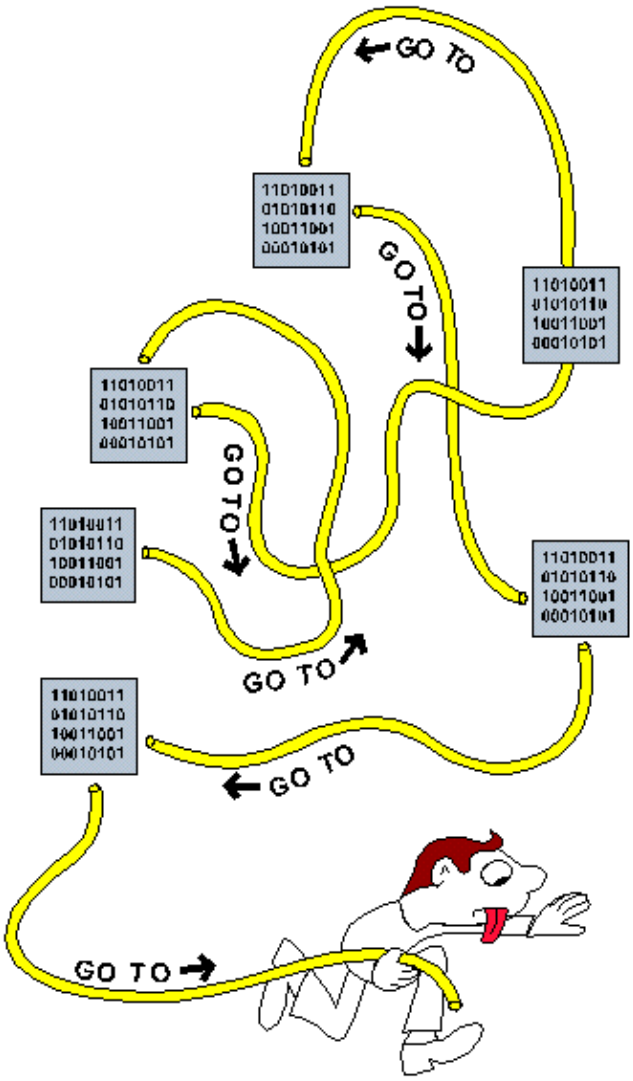
```
static void Main(string[] args)
{
    Console.WriteLine("Enter a number: ");

    int number = Convert.ToInt32(Console.ReadLine());

    if (number > 0) goto Positive;
    else if (number < 0) goto Negative;
    else goto Zero;

    Positive:
    Console.WriteLine("The number is positive.");
    goto End;
    Negative:
    Console.WriteLine("The number is negative.");
    goto End;
    Zero:
    Console.WriteLine("The number is zero.");
    goto End;
    End:;
}
```

```
C:\WINDOWS\system3:
Enter a number:
5
The number is positive.
Press any key to continue . . .
```



-Procedural Coding:

به کد بلاکی که فراخوانی میشود و بار ها و بار ها ازش استفاده میشود procedure یا رویه میگویند. معمولا در برنامه نویسی procedural ما برنامه را به بخش های مختلف جدا از هم تعریف میکنیم تا به صورت مجموعه جمله بندی شده بوجود آید تا به ترتیب کاری را انجام دهد. مثلا زیر تابع ها(subroutines, توابعی که خروجی ندارند) نوعی procedure هستند. معمولا بلوک procedure ها بیشتر از زیر تابع ها هستن و کنترل و توسعه ی زیر تابع ها راحت تر از procedure ها هستند. زبان های FORTRAN, ALGOL, COBOL, BASIC, Pascal و C زبان های procedural programming هستند. تصویر زیر مثالی از procedural code است و زیر تابع Reverse() یک procedure است:

```
class Program
{
    1 reference
    public void Reverse(ref string str) => str = string.Concat(str.Reverse());
    0 references
    static void Main()
    {
        Console.WriteLine("Enter a string: ");
        string input = Console.ReadLine();
        new Program().Reverse(ref input);
        Console.WriteLine("Reversed string: " + input);
    }
}
```

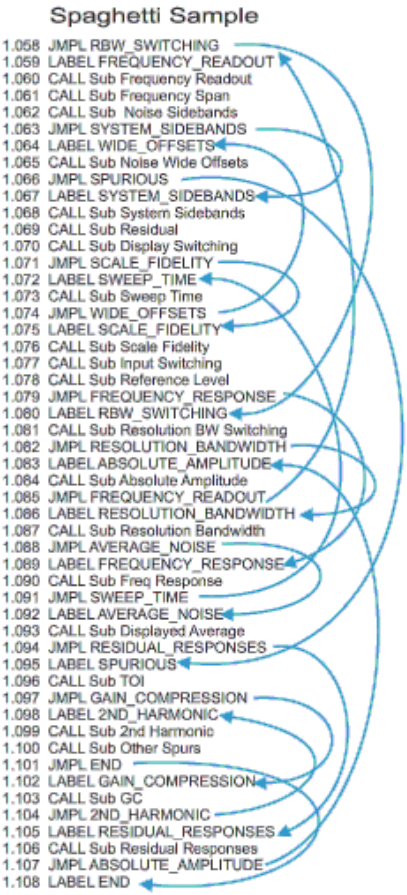
```
C:\WINDOWS\system3:
Enter a string:
Hello World
Reversed string: dlroW olleH
Press any key to continue . . .
```

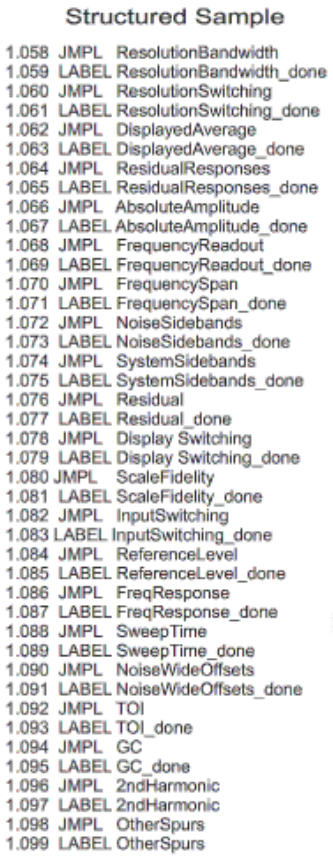
به مجموع Procedure, Function, Subroutine, Method Subprogram میگویند.

-Structural Programming:

برنامه نویسی ساخت یافته به برنامه ای گته میشود که از قبل نقشه ای برایش برنامه نویس برای کد زنی داشته باشد و به ترتیب کد ها اجرا شوند. در این برنامه نویسی انتظار میرود پیچیدگی و وابستگی کد ها به حداقل برسد. معمولا اکثره زبان ها از این پارادایم را دارند, مانند C, C++, C#, Java, JS, Python.

مقایسه spaghetti با structural:

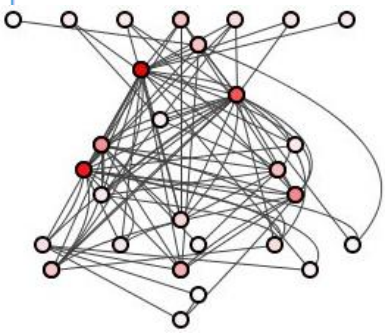


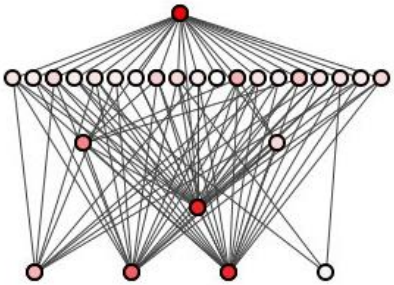



```
1 i=0;
2 i=i+1;
3 PRINT i; "squared=";i*i;
4 IF i>=100 THEN GOTO 6;
5 GOTO 2;
6 PRINT "Program Completed.";
7 END
```

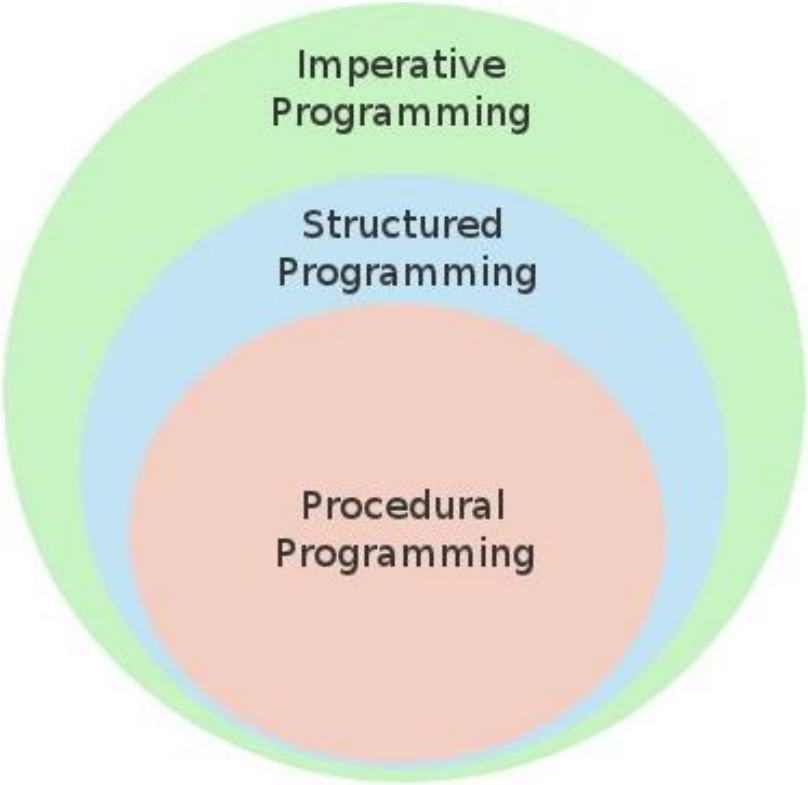
Here is the same code written in a structured programming style:

```
1 FOR i=1 TO 100
2     PRINT i;"squared=";i*i
3 NEXT i
4 PRINT "Program Completed."
5 END
```





مقایسه procedural با structural:



تصویر زیر مثالی از structural code است:

```
1 reference
static double Add(params double[] a)
{
    double b = 0;
    foreach (double i in a)
    {
        b += i;
    }
    return b;
}
1 reference
static double Multiply(params double[] a)
{
    double b = 1;
    foreach (double i in a)
    {
        b *= i;
    }
    return b;
}
0 references
static void Main()
{
    Console.WriteLine("Enter two numbers: ");
    int num1 = Convert.ToInt32(Console.ReadLine());
    int num2 = Convert.ToInt32(Console.ReadLine());

    Console.WriteLine("Sum: " + Add(num1, num2));
    Console.WriteLine("Multiplytion: " + Multiply(num1, num2));
}
```

```
C:\WINDOWS\system3:
Enter two numbers:
2
3
Sum: 5
Multiplytion: 6
Press any key to continue . . .
```

-Functional Programming(FP)

در این نوع برنامه نویسی از توابع استفاده میشوند و بیشتر از آن که بر روی "how to solve" متمرکز باشند بر بروی "what to solve" متمرکز هستند. زبان های Haskell, JavaScript, Python, Scala, Erlang, Lisp, Racket از آن پیروی میکنند ولی در کل FP از Lambda Calculus استفاده میکند و چون در C# ما Lambda expression داریم میتوانیم کدی بنویسیم که FP باشد. C# برای استفاده از این کد از کتابخانه ی LINQ (Language Integrated Query) استفاده میکند.

مزایا و معایب استفاده از FP:

مزایا:

- کد آسان تر برای تست و رفع اشکال میشود
- کد واحد بندی شده است (modular) و چندین بار میتوان ازش استفاده کرد
- اگر مقدار null به ورودی داده شود خطا نمیدهد و آن را نادیده میگیرد و در خروجی خالی میدهد (empty string)
- چندین خط را به یک خط خلاصه میکند(فشرده سازی)

معایب:

- برای کسانی که به Imperative Programming عادت کردند یاد گیری آن سخت است چون FP جزء Declarative Programming است
- کارایی کمتری دارد(همیشه استفاده نمیشود)

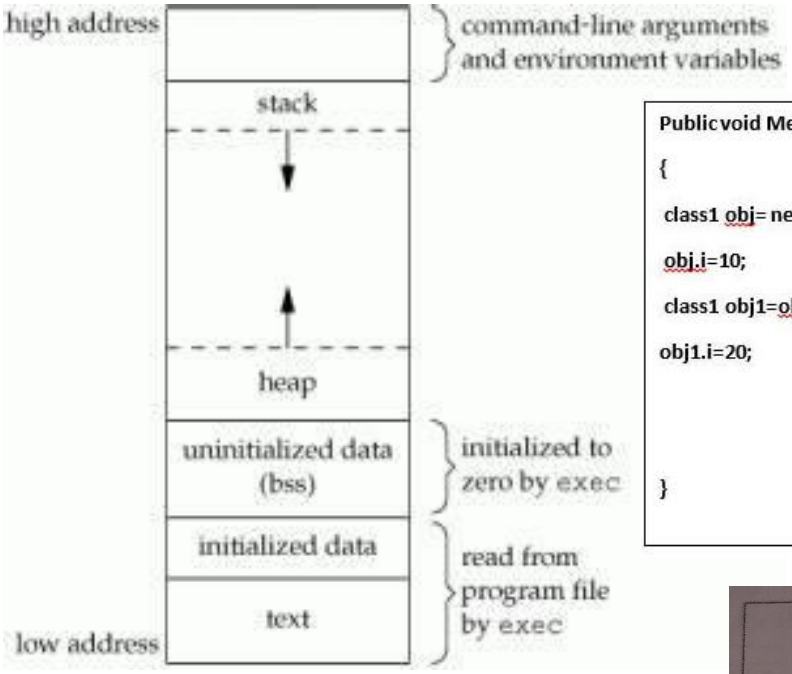
تصویر زیر مثالی از FP است:

```
static void Main()
{
    Console.WriteLine("Enter a list of numbers separated by spaces: ");
    int[] numbers = Console.ReadLine().Split(' ').Select(int.Parse).ToArray();
    int[] evenNumbers = numbers.Where(x => x % 2 == 0).OrderBy(x => x).ToArray();
    Console.WriteLine("Even numbers: " + string.Join(", ", evenNumbers));
}
```

```
C:\WINDOWS\system3:
Enter a list of numbers separated by spaces:
546 79 123 46 98 53 76 98 2 35 6
Even numbers: 2, 6, 46, 76, 98, 98, 546
Press any key to continue . . .
```

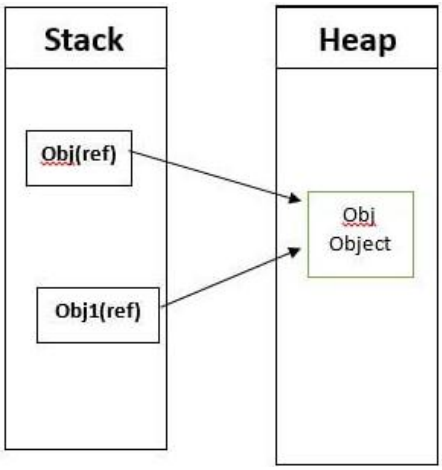
-Function Oriented Programming(FOP):

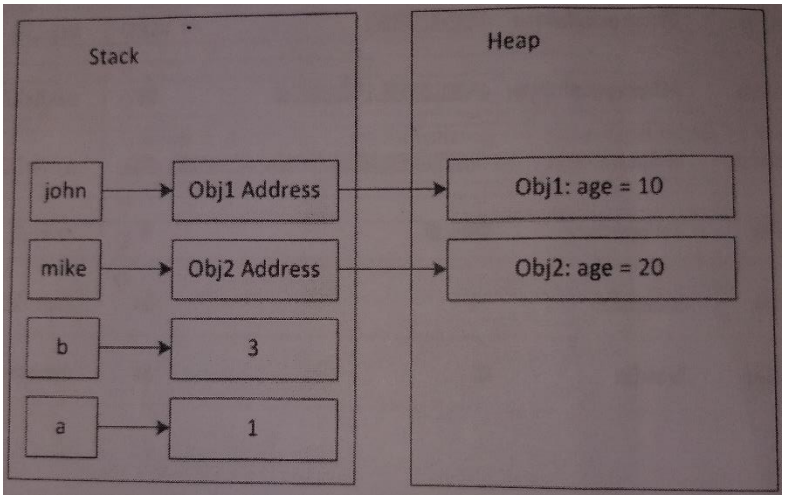
نوعی برنامه نویسی است که مراحل انجام کار را بخش بندی کرده و به توابع تبدیل می شوند. در FOP فقط از توابع و متغیر استفاده میشود. در FOP بخاطر نداشتن side effect, Encapsulation بوجود می آید. وقتی که کدی پتانسیل side effect را دارد به این معناست که ممکن است متغیر تصادفی مقدارش تغییر کند و چون از آنجایی که پارامتر سراسری است و در سرتاسر برنامه قابل دسترس است تاثیرش را در بقیه بلاک های که از آن متغیر استفاده میکنند, میگزارد, به همین دلیل از امنیت پایینی برخوردار است(پس اگر درون خود کلاسی که تابع Main() وجود دارد تمام کد هایتان را نوشتید, شما در حال برنامه نویسی به روش FOP هستید). متغیر ها در FOP به طور مستقیم در stack ذخیره در حالی که در OOP (به دلیل reference type بودن کلاس) وقتی که از کلاس نمونه میگیریم, نمونه ی ما در heap با تمام اطلاعاتش ذخیره میشود و آدرسش در stack ذخیره میشود. کلاس های static که نمیتوان از آنها نمونه گرفت و صرفا با صدا زدن نام کلاس میتوان از آنها استفاده کرد هم در heap ذخیره میشوند با این تفاوت که در قسمتی از heap ذخیره میشوند که به آن high frequency heap گفته میشود که نیازی به garbage collector ندارند. تصاویر زیر نحوه ی ذخیره شدن کلاس در حافظه را نشان میدهد.




```
Public void Method1()
{
 class1 obj= new class1();
 obj.i=10;
 class1 obj1=obj;
obj1.i=20;

}
```





تصویر زیر مثالی از fop پتانسیل side effect را دارد نشان میدهد که پارامتر سراسری است و در همه جای بلاک ها میتوان تغییرش داد:

```
class Program
{
    static double a, b, c;
    1 reference
    static double Average()
    {
        return (a + b + c) / 3;
    }
    0 references
    static void Main()
    {
        Console.WriteLine($"Enter 3 Numbers:");
        a = double.Parse(Console.ReadLine());
        b = double.Parse(Console.ReadLine());
        c = double.Parse(Console.ReadLine());
        Console.WriteLine($"Average: {Average()}");
    }
}
```

```
C:\WINDOWS\system3:
Enter 3 Numbers:
15
18
20
Average: 17.6666666666667
Press any key to continue . . .
```

در #C نام گذاری اصولی دارد و همه از آن استفاده میکنند در صورتی که در ++C نام گذاری قواعد ندارد!
مثلا در #C بهتر است هر کلاس در یک فایل جداگانه باشد و نام فایل با نام کلاس یکسان باشد, همینطور نام پروژه با نام namespace یکسان باشد. حال چون نمیتوانیم از فاصله(space) در نام گذاری استفاده کنیم, سوال پیش می آید که چگونه کلمات را از هم جدا کنیم و چطوری آنها را بدون فاصله بنویسیم. ما برای نامگذاری namespace, class, interface, property, method از Pascal Case استفاده میکنیم و برای object, variable(field & local) از Camel Case استفاده میکنیم. در جدول زیر دیگر گونه های نحوه نامگذاری بدون فاصله را آوردیم:

Naming Convention	
Name	**Example**
Lazy Case(Flat Case)	numberofdonuts = 34
Screaming Case(Upper Flat Case)	NUMBEROFDONUTS = 34
Camel Case(Lower Camel Case)	numberOfDonuts = 34
Pascal Case(Upper Camel Case)	NumberOfDonuts = 34
Snake Case	number_of_donuts = 34
Screaming Snake Case	NUMBER_OF_DONUTS = 34
Camel Snake Case	number_Of_Donuts = 34
Pascal Snake Case	Number_Of_Donuts = 34
Kebab Case	number-of-donuts = 34
Train Case	Number-Of-Donuts = 34
Cobol Case	NUMBER-OF-DONUTS = 34

در رعایت کردن این جور قواعد اجباری وجود ندارد ولی اگر رعایت شوند سرعت کامپایل شدن بیشتر میشود.

در نامگذاری توابع باید کاری که تابع انجام میدهد را نام تابع گذاشت, اما این نام باید چنان واضح باشد تا دیگر برنامه نویسیان بتواند راحت و سریع آن را استفاده یا فراخوانی کنند و وقتی که نام تابع به گوش میرسد بتوان به راحتی ماهیتش(چه کاری انجام میدهد) را حدس زد. اگر در نام گذاری تابع تان دچار سردرگمی شدید یا نامی که انتخاب کردید طویل است این به این معناست که تابع کاری که انجام میدهد واضح نیست یا تابع چندین کار را با هم انجام میدهد, بهتر است آن را به چند تابع دیگر تقسیم کنید. نام گذاری ها نباید زیاد طویل باشند چون سرعت را پایین می آورد ولی زیاد مهم نیست چون بعضی مواقع نام متغیر که طویل است بیشتر از سرعت ارزشمند میشود.

-Abstract Data Type(ADT):

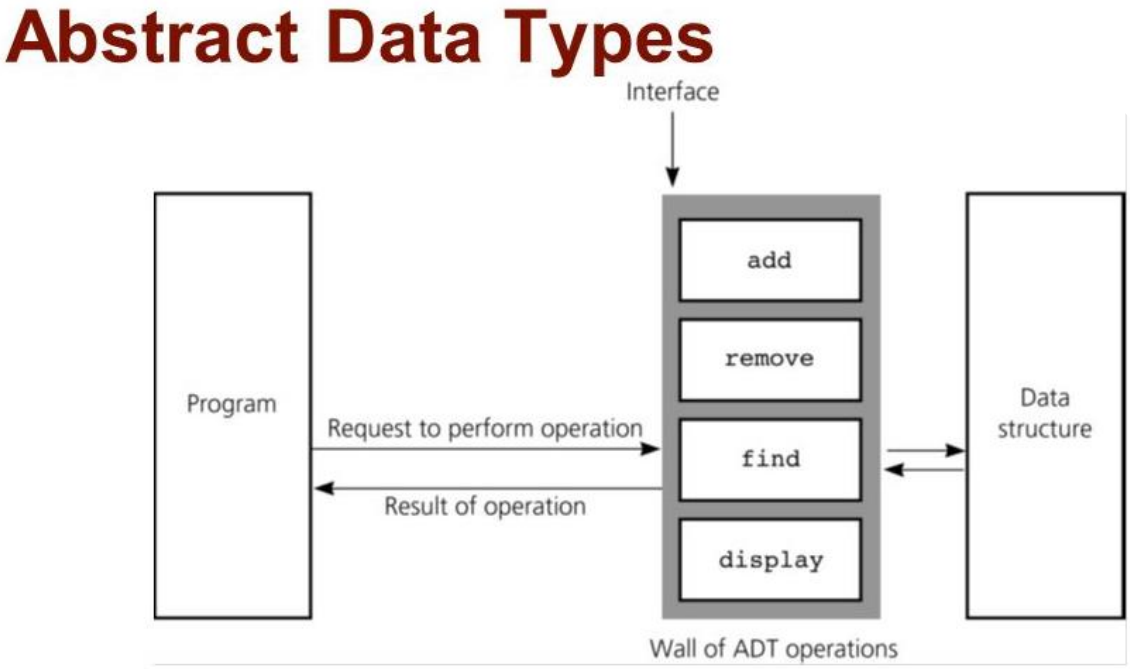
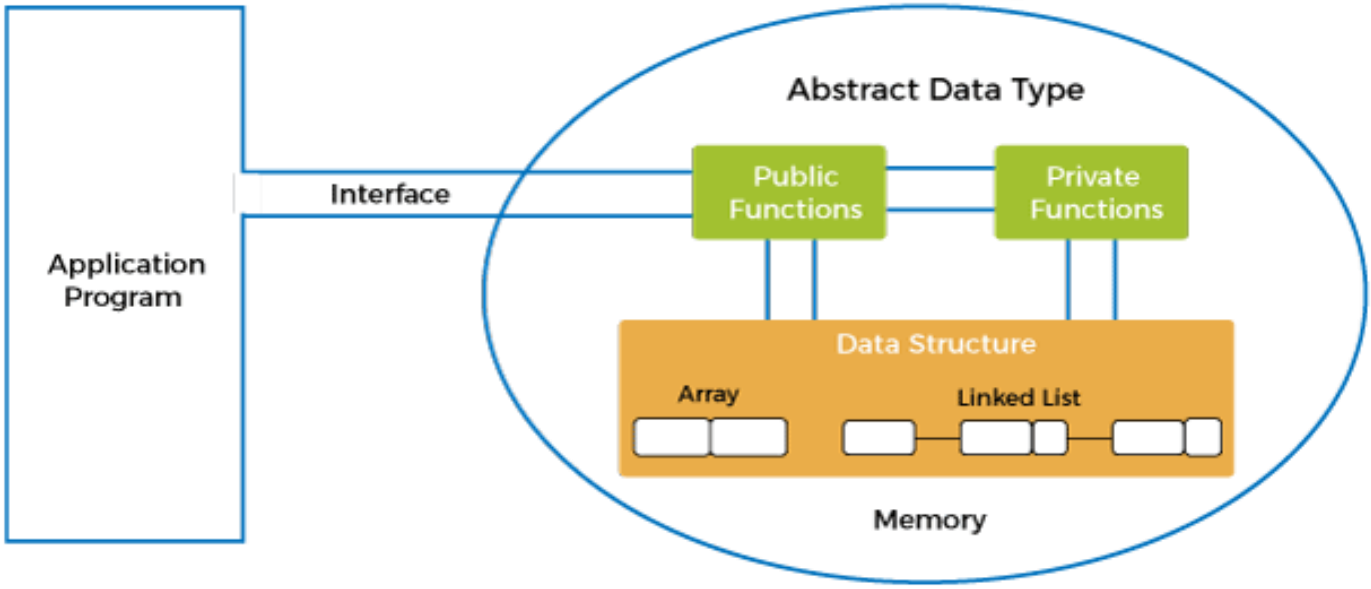


Figure 4-4

A wall of ADT operations isolates a data structure from the program that uses it



در علوم کامپیوتر ADT موجودیت تئوریک است که برای آسان کردن تعریف الگوریتم کد نویسی و طبقه بندی ساختمان دادها استفاده میشود.

در حالی که کلاس یک نقشه ای از موجودیت است که حالت(داده) و رفتارش(توابع) را درونش نگه داری میکند.

فرق ADT با OOP در اینجاست که OOP پیاده سازی شده ی تئوری ADT است یعنی OOP زیر مجموعه ی ADT است به عبارتی دیگر هر کلاسی چون از OOP پیروی میکند, یک ADT است.

Interface در ADT به معنی واسط است که منظورش توابع Setter & Getter است که واسطی بین کاربر و کلاس هستند که داده را از کاربر بگیرند یا داده های کلاس را تغییر دهند (Setter) یا داده را از کلاس بگیرند و به کاربر نشان دهند (Getter).

-Object Oriented Programming(OOP یا برنامه نویسی شئ گرا):

کلاس یک reference datatype است که درونش یک سری اطلاعات(state) و توابع(behavior) است. OOP یعنی بجای آنکه کد هایی که نیاز دارید را درون کلاسی بنویسید که تابع Main() وجود دارد(FOP), آنها را درون یک کلاس جداگانه با سطح های دسترسی متفاوت مینویسید.
State ویژگی یا اطلاعاتی است که ارزش ذخیره شدن در کلاس را دارند یا در آن موجودیتی که کلاس را برای آن تعریف میکنیم آن اطلاعات نیاز است. کلمات مترادف state کلمات روبه رویی هستند: attribute, member-variable, data

Behavior رفتاری است که کلاس از خود نشان میدهد یعنی ممکن است تابعی از داده های داخل کلاس استفاده کند و حتی تغییرش دهد, اگر بخواهیم تمام حالات ممکن توابع داخل کلاس را بررسی کنیم به نمودار زیر خواهیم رسید:

از اطلاعات داخل کلاس استفاده میکند	اطلاعات داخل کلاس را تغییر میدهد	عملیاتی را انجام میدهد	نوع تابع
✓	✓	✓	
✓	✓	✗	Getter & Setter
✓	✗	✓	
✓	✗	✗	Readonly
✗	✓	✓	
✗	✓	✗	Writeonly
✗	✗	✓	
✗	✗	✗	همچین تابعی وجود ندارد

Behavior همان method های کلاس هستند. وقتی که توابع درون کلاس بروند به آنها method میگویند.
اعضای کلاس را class member میگویند که شامل field, property, method میشود.
موجودیتی که فقط state داشته باشد و behavior نداشته باشد نمیتوان از آن کلاس ساخت.
منظور از موجودیت چیزی است که ما در واقعیت به کلاس نسبت میدهیم مانند انسانها, حیوانات, نباتات و حتی جامدات.
اگر بخواهیم انسان را بعنوان کلاس ذخیره کنیم, state رنگ پوست(سفید, زرد, قهوه ای, صورتی, سیاه, کرمی...), رنگ چشم, رنگ مو, نوع مو(فر, صاف...), ملیّت, نژاد(سرخ پوست, سیاه پوست, سفید پوست, فارس, کرد, لر, بلوچ, ترکمن, ترک, عرب, گیلک(رشتی), مازندرانی), قد, وزن, شکل چهره(مستطیلی, دایره...), شکل اندام(هفتی,هشتی...), تاریخ تولد, جنسیت و... میشود و behavior خوردن, خوابیدن, نشستن, ایستادن, راه رفتن, دویدن, خواندن, نوشتن, حرف زدن, اندیشیدن, گریه کردن, خندیدن, لبخند زدن, تعجب کردن و... میشود
زبان های Java, C++, C#, Python, PHP, JavaScript, Ruby, Perl, Objective-C, Dart, Swift, Scala از OOP پیروی میکنند.
در تمامی کلاس ها اگر برای اعضای کلاس access modifier ای تعریف نشود به طور پیش فرض private است و اگر برای خود کلاس access modifier ای تعریف نشود یه طور پیش فرض internal تعریف میشود. برای کلاس ها نمیتوان private, protected, protected internal & private protected تعریف نمود.

برای استفاده از کلاس ها دو راه وجود دارد:

1. **اسم کلاس را صدا بزنیم:**
   فقط به اعضایی دسترسی داریم که static و public باشند.
2. **از کلاس نمونه بگیریم یا obj بسازیم:**
   فقط به اعضایی دسترسی داریم که non-static و public باشند.

درباره ی نحوه ی فراخوانی شدن اجزاء کلاس ها بعدا در قسمت Access Modifier حرف میزنم.

کلاس در #C به صورت زیر تعریف میشود:

```
2 references
class ClassName
{
    string fieldName;
    2 references
    public string PropertyName { get => fieldName; set => fieldName = value; }
    1 reference
    public void MethodName()
    {
        Console.WriteLine("Hey, I'm method of class\n" +
            "You just invoked me!\n" +
            $"My property's value is: {PropertyName}\n");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        ClassName objectName = new ClassName();
        objectName.PropertyName = "Hello OOP";
        objectName.MethodName();
    }
}
```

```
C:\WINDOWS\system3:

Hey, I'm method of class
You just invoked me!
My property's value is: Hello OOP

Press any key to continue . . .
```

انواع کلاس ها در #C:

1. **Regular or Non-Abstract Class:**

کلاس معمولی که همه ی ما میشناسیم.

2. **Abstract Class:**

این کلاس همون طوری که از اسمش (abstract, تجرید, خلاصه کردن) پیداست, کلاسی است که اطلاعاتش یا رفتارش(تابعش) ناقص است و یک جای دیگر باید تعریف شود.

3. **Partial Class:**

همین طور که از Partial یعنی "کلاس قسمتی" که قسمت بندی شده است و در کار گروهی استفاده میشود و برای توسعه دادن چندین نفری کلاس است به این منظور که یک قسمت از کد کلاس را یک نفر میتواند در یک قسمت از پروژه و قسمت بعدی کلاس را فرد دیگری در قسمت دیگر پروژه بنویسد یا حتی در یک فایل cs. دیگری بنویسد. کامپایلر #C این قسمت ها را با هم کامپایل میکند و به هم ربط میدهد و یک کلاس مرجع میسازد. در کار های گروهی استفاده میشود. اگر در کلاس partial یک از افراد توسعه کننده (developer) در هنگام نوشتن اسم کلاس, به اشتباه از صفات abstract یا sealed یا static استفاده کند, کامپایلر هنگام ساختن کلاس مرجع partial آن کلاس را همان صفتی که تعریف شده در نظر میگیرد(یعنی اگه 10 تا partial class داشته باشیم و یکیش به اشتباه static partial class نوشته شده باشد, کلاس مرجع static class است).

4. **Sealed Class:**

همان طور که از اسمش پیداست کلاس Sealed به معنی "مهر و موم شده" که نمیگذارد کسی ازش ارث بری کند. موقعی از این کلاس استفاده میکنیم که نمی خواهیم کسی از آن ارث بری کند تا از ویژگی های کلاسمان استفاده کند!

5. Static Class:

کلاس Static که معنی ایستا, ایستاده یا ساکن میدهد کلاسیس که همه ی اعضای کلاس Static است و نمیتوان از این کلاس ارث بری یا نمونه گرفت و فقط با صدا زدن نام کلاس فراخوانی میشود. Static را میتوان جوری فهمید که به معنی این است نمیتواند در obj حضور پیدا کند و فقط و فقط متعلق کلاس است. چون obj گرفتن همان نمونه گیری یا کپی کردن از کلاس است میتوان گفت که اگر یک چیز static باشد پس نمیتوان ازش کپی گرفت و فقط یک بار در حافظه قرار میگیرد.

اگر تمام اعضای کلاس شما static است پیشنهاد میشود که کلاس هم static باشد. کلاس System.Math یک کلاس static است. به استثنا در این کلاس field ها به شرط const بودن میتوانند static نباشند!

در #C یک reference data type دیگری هم داریم که کلاس ها از آن به عنوان واسط استفاده میکنند که اسمش interface است. interface یک کلاس نیست! و با کلاس ها متفاوت است.

Interface:

OOP از بدر تولد که بوجود آمد شعارش این بود که امنیت را بالا و پیچیدگی را پایین بیآورد ولی همیشه اینطوری نبوده است!. مثلا چون ++C از Multiple Inheritance پیروی میکند, بعد از استفاده ی چندین بار از ارث بری باعث میشود که پروژه دیگر قادر به توسعه دادن نباشد چون به دلیل ویژگی ارث بری (سلسله مراتب, Hierarchy) پیچیدگی کلاس به حدی میرسد که قابل توسعه نمیشود, و به همین دلیل در #C ما درون یک کلاس فقط میتوانیم یک کلاس دیگر ارث بری کنیم که به آن Single Inheritance می گویند. اگر ما بخواهیم از ارث بری چندگانه (Multiple Inheritance) در #C استفاده کنیم و در عین حال در آینده دچار پیچیدگی نشویم, میتوانیم از Interface استفاده کنیم. اعضای Interface بخاطر نداشتن بدنه, دوباره تعریف شدن شان از کلاس وارث, با پیچیدگی کمتری مواجه میشوند. در Interface نمیتوان field تعریف کرد چون field را نمیتوان virtual یا abstract تعریف کرد که بعدا override ش کرد!

دو تعریف برای Interface وجود دارد:

- Interface یک fully abstract class است که باید همه ی اعضایش بدون بدنه و به طور پیش فرض public abstract تعریف میشوند(که در Abstract class اختیار با برنامه نویس است که چه چیزی را abstract و چه چیزی را non-abstract تعریف کند.)
- Interface یک قرار داد است.(چونکه Interface شرط میکند که برای ارث بری باید در کلاس وارث تمام اعضایش پیاده سازی شود(تعریف یا Implement شود), میگویند انگار که یک قرارداد است)

به واسطه ی Interface میتوانیم کلاس هایی که یک نوع توابع دارند ولی ماهیت توابع (ضابطه تابع, کد داخل تابع, implementation of method) با هم متفاوت است را به Interface مختلف نسبت داد یا یک مجموعه از رفتارهایی را تعریف کنید که چندین کلاس مختلف باید پیاده سازی کنند.

. مثلا. به مثال توجه کنید...

به عنوان مثال، فرض کنید شما یک برنامه نویسید که نیاز به ذخیره سازی داده ها در چندین منبع مختلف دارد، مانند پایگاه داده، فایل و حافظه. شما می توانید یک interface تعریف کنید که عملکردهای ذخیره سازی و بازیابی داده ها را تعریف می کند و سپس چندین کلاس را برای پیاده سازی این interface برای هر منبع داده ایجاد کنید. به عبارتی دیگر موجودیت های انسان, سگ و گربه رفتار مشترک مثل غذا خوردن و دویدن و حرف زدن را دارند ولی هیچ وقت مثل همدیگر عمل نمیکنند! یا همه ی اشکال دو بعدی محیط و مساحت دارند ولی فرمولشان با هم یکسان نیست, از interface استفاده میکنیم.

دو تصویر زیر کلاس های string Build-in و Int32 را نشان میدهد که از interface ها در زمینه های مختلف استفاده کرده است(مثال گفته شده ی ذخیره سازی):

```
namespace System
{
    //
    // Summary:
    //     Represents text as a sequence of UTF-16 code units.
    [Serializable]
    [ComVisible(true)]
    [__DynamicallyInvokable]
    public sealed class String : IComparable, ICloneable, IConvertible, IEnumerable, IComparable<string>, IEnumerable<char>, IEquatable<string>
    {
        [NonSerialized]
        private int m_stringLength;

        [NonSerialized]
        private char m_firstChar;

        private const int TrimHead = 0;

        private const int TrimTail = 1;

        private const int TrimBoth = 2;
```

```
namespace System
{
    //
    // Summary:
    //     Represents a 32-bit signed integer.
    [Serializable]
    [ComVisible(true)]
    [__DynamicallyInvokable]
    public struct Int32 : IComparable, IFormattable, IConvertible, IComparable<int>, IEquatable<int>
    {
        internal int m_value;

        //
        // Summary:
        //     Represents the largest possible value of an System.Int32. This field is constant.
        [__DynamicallyInvokable]
        public const int MaxValue = 2147483647;

        //
        // Summary:
        //     Represents the smallest possible value of System.Int32. This field is constant.
        [__DynamicallyInvokable]
        public const int MinValue = -2147483648;
```

تصویر زیر مثال محیط و مساحت را شرح میدهد:

```
using System;

namespace Test
{
    2 references
    interface IShape
    {
        4 references
        void GetPerimeter();
        4 references
        void GetArea();
    }
    3 references
    class Circle : IShape
    {
        double radius;
        const double pi = 3.1415;
        1 reference
        public Circle(double radius)
        {
            this.radius = radius;
        }
        2 references
        public void GetPerimeter()
        {
            Console.WriteLine(2 * radius * pi);
        }
        2 references
        public void GetArea()
        {
            Console.WriteLine(Math.Pow(radius, 2) * pi);
        }
    }
    3 references
    public class Rectangle : IShape
    {
        double length;
        double width;
        1 reference
        public Rectangle(double width, double length)
        {
            this.width = width;
            this.length = length;
        }
        2 references
        public void GetPerimeter()
        {
            Console.WriteLine(2 * (length + width));
        }
        2 references
        public void GetArea()
        {
            Console.WriteLine(length * width);
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main()
        {
            Rectangle obj1 = new Rectangle(4, 3);
            Circle obj2 = new Circle(3);

            Console.Write("Rectangle Perimeter: ");
            obj1.GetPerimeter();
            Console.Write("Rectangle Area: ");
            obj1.GetArea();

            Console.Write("Circle Perimeter: ");
            obj2.GetPerimeter();
            Console.Write("Circle Area: ");
            obj2.GetArea();
        }
    }
}
```

```
C:\WINDOWS\system3.
Rectangle Perimeter: 14
Rectangle Area: 12
Circle Perimeter: 18.849
Circle Area: 28.2735
Press any key to continue . . .
```

interface ها نسبت به class ها انعطاف پذیرتر هستند، چون چندین interface را می توان در یک کلاس پیاده سازی کرد، در حالی که یک کلاس فقط می تواند از یک کلاس دیگر ارث ببرد. این به شما امکان میدهد تا قابلیت های مختلف را در یک کلاس ترکیب کنید.

برای اعضای کلاسی که abstract یا virtual هستند نمیتوان آنها را private تعریف نمود چون بعدا ممکن است باز نویسی(override) شوند. انواع کلاس های گفته شده هر کدام ویژگی های خودشان را دارند, برای همین باید اول انواع توابع را بگوییم که در قسمت Polymorphism

جملات زیر غلطند:

1. هر کلاسی که نمیتوان ازش نمونه گیری کرد, constructor ندارد!
2. هر کلاسی که constructor دارد, destructor هم دارد!

همه ی کلاس ها بجز کلاس static دارای constructor ساده هستند و این کلاس static constructor دارد!
جدول زیر ویژگی های کلاس ها و Interface را نشان میدهد.

Class Type						Interface
انواع / ویژگی و اجزاء آن	Regular	Abstract	Partial	Sealed	Static	
Abstract Field	✗	✗	✗	✗	✗	✗
Non-Abstract Field	✓	✓	✓	✓	✓	✗
Abstract Property	✗	✓	✗	✗	✗	✓
Non-Abstract Property	✓	✓	✓	✓	✓	✓
Abstract Method	✗	✓	✗	✗	✗	✓
Non-Abstract Method	✓	✓	✓	✓	✓	✗
Instantiation	✓	✗	✓	✓	✗	✗
Constructor	✓	✓	✓	✓	✓	✗
Destructor	✓	✓	✓	✓	✗	✗
Single Inheritance	✓	✓	✓	✗	✗	✓
Multiple Inheritance	✗	✗	✗	✗	✗	✓

چرا به آن ها کلاس میگوییم؟ چون در زبان انگلیسی به مرسوم است به گروهی که یک نوع هستن کلاس بگویند در شی گرایی هم به آنها کلاس میگوییم
چرا به آنها شی میگوییم؟ چون نمونه های کلاس ها ویژگی و رفتار دارند و آنها را در نوع موجودیت(در دنیای واقعی) تعریف میکنیم به آنها شی میگوییم
مزایا و معایب استفاده از OOP:
مزایا:

- بهبود بهره وری توسعه نرم افزار(توسعه ی آسان)
- قابلیت استفاده مجدد از کد(Reusability)
- نگهداری آسان تر
- طراحی مدولار(Modularity)
- جداسازی وظایف
- قابلیت گسترش

معایب:

- برنامه ها می توانند بزرگتر و کندتر اجرا شوند
- برای همه انواع وظایف برنامه نویسی مناسب نیست

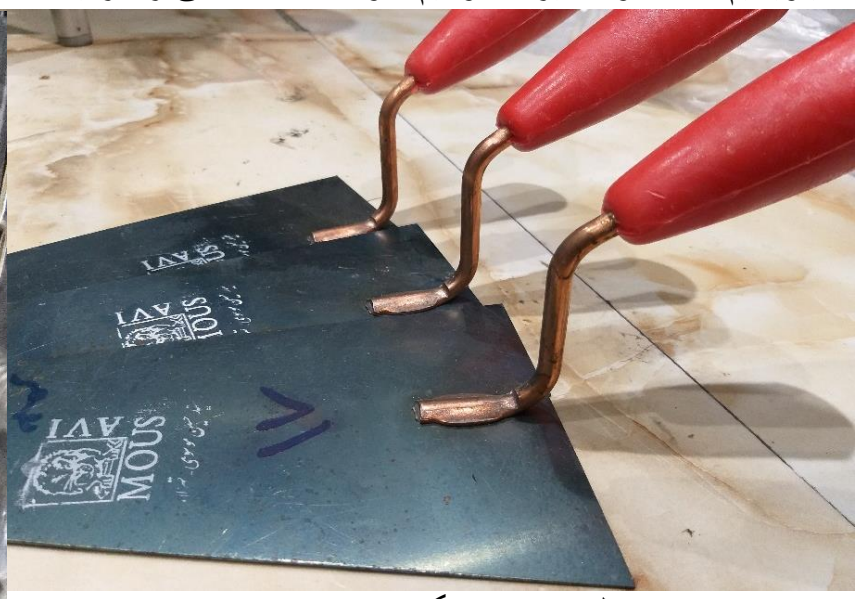
-Abstraction:

خلاصه کردن یا تجرید, یعنی ما به کاربر اطلاعاتی نمیدهیم که به آن هیچ احتیاجی ندارد به طور مثال یک زن خانه دار(کاربر) وقتی که بخواهد قرمه سبزی درست کند به سراغ سبزی های تره, جعفری و شنبلیله میرود یا اگر بخواهد آش رشته بپزد به سراغ سبزی های تره, جعفری, گیشنیز, اسفناج و سیر میرود و سراغ سبزی های دیگر نمیرود چون نیازش ندارد و اگر هم سبزی های دیگر را بهش بدهیم ازش استفاده نمیکند و تا بردن سبزی از سبزی فروش به منزل فقط خودش را خسته کرده است و انرژی الکی هدر داده است, یا مثلا اگر یک شخصی بخواهد لپ تاپی بخرد با توجه به اینکه برنامه نویس یا گرافیست است به همان مقدار لازم سراغ لپ تاپی میرود که ram cpu کارت گرافیک دارد بقیه چیز ها برایش اهمیتی ندارد. این کار یعنی استفاده ی بهینه از حافظه و زمان چون شما فقط اطلاعات مورد نیازتون رو ذخیره میکنید در نتیجه از حافظه استفاده ی حداکثری می برید, از طرفی چون اطلاعات اضافی را نادیده گرفتید, خود را درگیر چیز های بیهوده که تاثیری در سیستم شما ندارد نمیکنید, در نتیجه در مدت زمان ساختن کلاس صرفه جویی کردید و از طرفی چون تمام اطلاعات کلاس را نیاز دارید و هیچ اطلاعات اضافی موجود نیست, مدیریت آن کلاس آسانتر و پیچیدگی کمتر میشود بنابراین عمر برنامه بیشتر میشود.

-Constructor(تابع سازنده):

-Instantiation(نمونه سازی یا نمونه گیری):

کلاس ها برای این هستند که بجای آن که برای تمام موجودیت هایمان کد های جداگانه بنویسیم از آن ها نمونه بگیریم, به طور مثال فرض کنید 10 تا انسان با علایق, سلایق, ظاهر, اطلاعات و رفتار های مختلف را کد بزنیم, چقدر طولانی میشود؟ فکر کنید مقدارش بیشتر می بود, چه میشد؟ نه فقط موجودیت انسان بلکه اگر بخواهیم سگ و گربه را به چندین تعداد بصورت جداگانه کد بزنیم هم وقت و انرژی مان را از ما میگیرد هم اگر قرار باشد اینها درون پروژه باشد, زمان تحویل پروژه را زیاد میکند. برای همین ما همه ی ویژگی هایی که یک موجودیت میتواند داشته باشد را درون کلاس مینویسیم تا با نمونه گیری بتوانیم نمونه های مختلفی را از یک کلاس کپی کنیم. مثال های زیر را توجه کنید:



این دو تصویر مثالی از نمونه گیری در دنیای واقعی است, مثلا در خوار بار فروشی, فروشنده برای آنکه قیمت محصولش را بداند و به مشتری بگوید روی اجناس قیمت را مینویسد ولی اگر قرار باشد بر روی همه ی اجناس قیمت بنویسد, خیلی از او وقت میگیرد بخاطر همین روی یکی از اجناس قیمت را مینویسد. در این تصاویر, جنسی که قیمت رویش است کلاس در نظر گرفته میشود و آنهایی که قیمت ندارند, نمونه یا obj آن کلاس هستند. فروشنده همیشه از جنس هایی که رویش قیمت نیست به مشتری میدهد تا قیمت را گم نکند ولی اگر تعداد نمونه ها (جنس هایی که قیمت ندارند) به صفر رسید, خوده جنسی که رویش قیمت است را به مشتری میدهد که این به معنای اتمام موجودی آن جنس است. اگر جنسی که رویش قیمت است را اول از نمونه ها بفروشد, قیمت نمونه ها که جزئی از ماهیت شان است را گم میکند.

-Constructor:

تابع سازنده اولین تابعی است که همزمان با ساخته شدن obj فراخوانی داده میشود. این تابع توسط برنامه نویس فراخوانی نمیشود. در C++ به اختیار میتوان Constructor برای کلاس گذاشت, در C# هم تعریف کردن این تابع به اختیار برنامه نویس است ولی C# به طور پیش فرض برای هر کلاس آن را در نظر میگیرد (new ClassName()) اما به محض تعریف شدن این تابع در کلاس, compiler تابع تعریف شده را به رسمیت میشناسد و آن را اجرا میکند. در هر کلاس میتوان چندین تابع Constructor داشت. تابع سازنده بیشتر برای مقدار دهی اولیه به کار میرود. مثلا در کلاسی میخواهید بلافاصله بعد از ساخته شدن نمونه, به نمونه مقدار اولیه بدهید و دیگر تغییر داده نشود یا یک کاری را انجام دهد. تابع سازنده همیشه همنام کلاس و با Access Modifier های مختلف تعریف میشود.

دو تصویر زیر را از نظر حجم کد داخل تابع Main() با هم مقایسه کنید:

```
7 references
class Car
{
    string model;
    string color;
    int year;
    3 references
    public Car(string modelName, string modelColor, int modelYear)
    {
        model = modelName;
        color = modelColor;
        year = modelYear;
    }
    3 references
    public void ShowInfo()
    {
        Console.WriteLine($"Car's info is:\nModel: {model}\nColor: {color}\nMade Year: {year}\n");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Car Ford = new Car("Mustang", "Red", 1969);
        Car Opel = new Car("Astra", "White", 2005);
        Car Saipa = new Car("Pride 131E", "White", 1392);
        Ford.ShowInfo();
        Opel.ShowInfo();
        Saipa.ShowInfo();
    }
}
```

```
C:\WINDOWS\system3:

Car's info is:
Model: Mustang
Color: Red
Made Year: 1969

Car's info is:
Model: Astra
Color: White
Made Year: 2005

Car's info is:
Model: Pride 131E
Color: White
Made Year: 1392

Press any key to continue . . .
```

```
6 references
class Car
{
    string model;
    string color;
    int year;
    3 references
    public string Model
    {
        get { return model; }
        set { model = value; }
    }
    3 references
    public string Color
    {
        get { return color; }
        set { color = value; }
    }
    3 references
    public int Year
    {
        get { return year; }
        set { year = value; }
    }
    3 references
    public void ShowInfo()
    {
        Console.WriteLine($"Car's info is:\nModel: {model}\nColor: {color}\nMade Year: {year}\n");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Car Ford = new Car();
        Ford.Model = "Mustang";
        Ford.Color = "Red";
        Ford.Year = 1969;

        Car Opel = new Car();
        Opel.Model = "Astra";
        Opel.Color = "white";
        Opel.Year = 2005;

        Car Saipa = new Car();
        Saipa.Model = "Pride 131E";
        Saipa.Color = "White";
        Saipa.Year = 1392;

        Ford.ShowInfo();
        Opel.ShowInfo();
        Saipa.ShowInfo();
    }
}
```

C:\WINDOWS\system3:

```
Car's info is:
Model: Mustang
Color: Red
Made Year: 1969

Car's info is:
Model: Astra
Color: white
Made Year: 2005

Car's info is:
Model: Pride 131E
Color: White
Made Year: 1392

Press any key to continue . . .
```

-Destructor(Finalizer تابع مخرب):

-Destructor:

Destructor که به آن finalizer هم میگویند. تابع ایست همنام کلاس, بدون Access Modifier و پارامتر ورودی, اول نام تابع علامت ~(Tilde) می آید که برنامه نویس نمیتواند آن را صدا بزند و به طور خودکار توسط Garbage Collector(GC) در آخر برنامه ها(وقتی که برنامه از حالت در حال اجرا به طور کامل بیرون می آید) صدا زده میشود. در هر کلاس یک یا هیچ از این تابع وجود دارد. در Net Framework. یک قسمتی است به نام Common Language Runtime(CLR) که درونش GC وجود دارد که وظیفه اش چک کردن حافظه است, تا حافظه را از متغیر های برنامه خالی کند تا بلا فاصله فضای آزاد در حافظه برای انجام دادن کار های ضروری دیگر بوجود آید. وقتی که GC دارد حافظه را خالی میکند, اگر در حافظه obj ای باشد قبل از پاک کردنش, در صورت وجود Destructor در کلاس obj آن را صدا میزند. در هیچ زبانی نوشتن تابع Destructor اجباری نیست. در این تابع میتوانید کد هم بنویسید تا قبل از حذف شدن obj پیغامی یا کاری را انجام دهد. در بسیاری از زبان ها مانند Java, C#, C++, Python نیازی به نوشتن GC نیست چون آن ها به طور پیش فرض آن را درون خودشان دارند که به آن Built-in Garbage Collector میگویند. به مثال زیر توجه کنید...

```
6 references
class Person
{
    string firstName, lastName;
    2 references
    public Person(string firstName, string lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    0 references
    ~Person()
    {
        Console.WriteLine($"Dear {firstName} {lastName}" +
            $"\nthank you for choosing us.\n");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Person obj1 = new Person("ALi", "Moradzade");
        Person obj2 = new Person("Parviz", "Soorizade");
    }
}
```

```
C:\WINDOWS\system3:
Dear Parviz Soorizade
thank you for choosing us.

Dear ALi Moradzade
thank you for choosing us.

Press any key to continue . . .
```

```
6 references
class Person
{
    string firstName, lastName;
    2 references
    public Person(string firstName, string lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    0 references
    ~Person()
    {
        Console.WriteLine($"Dear {firstName} {lastName}" +
            $"\nthank you for choosing us.\n");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Test();
        Person obj2 = new Person("Parviz", "Soorizade");
    }
    1 reference
    static void Test()
    {
        Person obj1 = new Person("ALi", "Moradzade");
    }
}
```

```
C:\WINDOWS\system3:
Dear Parviz Soorizade
thank you for choosing us.

Dear ALi Moradzade
thank you for choosing us.

Press any key to continue . . .
```

-Dispose:

اگر بخواهیم وقتی که کامپایلر از بلاکی که obj درونش تعریف شده(local var) است بیرون آمد, آن obj را از حافظه پاک کند, یا هر وقت بخواهیم حذفش کنیم, باید یک تابع دیگری را بسازیم که به واسطه ی آن بتوانیم حافظه را از obj خالی کنیم, در غیر این صورت هیچ راه دیگری وجود ندارد. این تابع نامش Dispose هست که برای پیاده سازی (implementation) آن باید از interface IDisposable ارث بری کنیم. این تابع همانند finalizer , میتوان درونش کد نوشت(مثلا خبری را به کاربر بدهید).

در واقع تابع Dispose که درونش کد GC.SuppressFinalize(this); است به GC میگوید که وقتی که خواستی این obj را از حافظه برداری تابع Finalizer رو صدا نزن(نخونش).

تابع Dispose حتما باید public void باشد ولی اگر خواستید public static void بگذارید باید از IDisposable ارث بری نکنید!

تصویر زیر نحوه ی کار کرد Finalizer و Dispose را نشان میدهد:

```
10 references
class Person : IDisposable
{
    string firstName, lastName;
    2 references
    public string FullName => (firstName ?? "Unknown") + ' ' + (lastName ?? "Unknown");
    4 references
    public Person(string firstName, string lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    1 reference
    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Console.WriteLine($"Dispose method invoked\n" +
            $"still I have obj info: {FullName}.\n");
    }
    0 references
    ~Person()
    {
        Console.WriteLine($"Finalize method invoked\n" +
            $"still I have obj info: {FullName}.\n");
    }
}
0 references
class Program
{
    1 reference
    static void Sample1()
    {
        using (Person obj1 = new Person("ALi", "Moradzade"))
        {
            Console.Write("I'm in Sample1.\nNow, ");
        }
    }
    0 references
    static void Main()
    {
        Sample1();
        Sample2();
        Person obj3 = new Person("Parviz", "Soorizade");
        Person obj4 = new Person(null, null);
        Console.Write("I'm in Main.\nNow, ");
    }
    1 reference
    static void Sample2()
    {
        Person obj2 = new Person("Hamed", "Daneshi");
        Console.Write("I'm in Sample2.\nNow, ");
        obj2.Dispose();
    }
}
```

```
C:\WINDOWS\system3:

I'm in Sample1.
Now, Dispose method invoked
still I have obj info: ALi Moradzade.

I'm in Sample2.
Now, Dispose method invoked
still I have obj info: Hamed Daneshi.

I'm in Main.
Now, Finalize method invoked
still I have obj info: Unknown Unknown.

Finalize method invoked
still I have obj info: Parviz Soorizade.

Press any key to continue . . .
```

-Encapsulation(کپسوله سازی یا محصور سازی یا حفاظت):

به ایده ی ندادن دسترسی به کاربر و جلو گیری از side effect در پروژه توسط برنامه نویسان, encapsulation گفته میشود. این کار را چون نمیخواهیم به کاربر همه ی اطلاعات داخل کلاس را نشان دهیم تا کاربر سر در گم شود یا دسترسی بی جا داشته باشد که داده را تغییر دهد, انجام میدهیم. با این روش ما میگوییم چه توابعی را کی کجا میتواند استفاده کند یا داده ها را کی کجا میتواند بخواند یا بنویسد یا حتی چه ویژگی هایی توسط کلاس ها به ارث برده شوند.
برای آنکه بیشتر با این اصول از OOP آشنا شوید باید Access Modifier و Member Modifier ها را بدانید.

-Member Modifier:

Member Modifier میگوید که کدام یک از اعضای کلاس در نمونه ی کلاس وجود داشته باشند و کدام فقط درون کلاس وجود داشته باشند(در نمونه ی کلاس وجود نداشته باشند).
جدول زیر تمام اطلاعات لازم را درباره ی Member Modifier ها میدهد ولی برای مطمئن شدن از درک کامل موضوع, صفحه ی بعد مثالی از تمام حالات جدول زدیم.

Member Modifier / Pros & Cons	regular	void	static	static void
	In Class			
Can define on field variable	✓	✗	✓	✗
Can define on local variable	✓	✗	✗	✗
Have access to members within object of class	✓	✓	✗	✗
	In Method			
Have access to static field variable	✓	✓	✓	✓
Have access to non-static field variable	✓	✓	✗	✗
Have access to static method	✓	✓	✓	✓
Have access to static void method	✓	✓	✓	✓
Have access to void method	✓	✓	✗	✗
Have access to methods within object of class	✓	✓	✗	✗
Return a value - Has datatype	✓	✗	✓	✗

```
5 references
class ClassName
{
    int privateNonStaticField;
    static int privateStaticField;

    public int publicNonStaticField;
    public static int publicStaticField;

    3 references
    public int RegularMethod()
    {
        privateNonStaticField = 1;
        privateStaticField = 1;
        publicNonStaticField = 1;
        publicStaticField = 1;
        RegularMethod();
        StaticMethod();
        VoidMethod();
        StaticVoidMethod();
        return 0;
    }
    5 references
    public static int StaticMethod()
    {
        privateStaticField = 1;
        publicStaticField = 1;
        StaticMethod();
        StaticVoidMethod();
        return 0;
    }
    3 references
    public void VoidMethod()
    {
        privateNonStaticField = 1;
        privateStaticField = 1;
        publicNonStaticField = 1;
        publicStaticField = 1;
        RegularMethod();
        StaticMethod();
        VoidMethod();
        StaticVoidMethod();
    }
    5 references
    public static void StaticVoidMethod()
    {
        privateStaticField = 1;
        publicStaticField = 1;
        StaticMethod();
        StaticVoidMethod();
    }
    0 references
    void Test()
    {
        //static int s; Error: Can't define static to local var
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        ClassName obj = new ClassName();
        obj.publicNonStaticField = 1;
        obj.RegularMethod();
        obj.VoidMethod();

        ClassName.publicStaticField = 1;
        ClassName.StaticMethod();
        ClassName.StaticVoidMethod();
    }
}
```

فعلا در نظر داشته باشید که اگر یک چیزی در کلاس public باشد در تمام جاهای برنامه قابل دسترس است و اگر private باشد فقط درون کلاس قابل دسترس است, بعدا درباره ی آنها(Access Modifier) صحبت میکنیم.

همان طور که گفتیم اعضای کلاس اگر access modifier برایشان تعریف نشود, به طور پیش فرض برای آنها private تعریف میشود که در هر صورت, چه با نمونه گیری چه بدون نمونه گیری (صدا زدن اسم کلاس) نمیتوان به آن دسترسی داشت. اعضایی که static هستند را میتوان بدون نمونه گیری بهشان دسترسی داشت. تصویر روبه رو مفاهیمی که درباره ی static و void, فراخوانی با نام کلاس یا obj هست را به خوبی نشان میدهد:

کد تصویر روبه رو , به دو دلیل جنبه ی اجرایی شدن را ندارن:

1. کاری را انجام نمیدهد و صرفا برای یادگیری است
2. توابع بخاطر صدا زدن خودشان از درون, تابع بازگشتی و بخاطر بی قید و شرط بودن این صدا زدن, حلقه بینهایت را بوجود می آورند که ارور و پیام زیر را میدهد:

```
C:\WINDOWS\system32\cmd.
Process is terminated due to StackOverflowException.
Press any key to continue . . .
```

اگر درون کلاس از خودش نمونه بگیرید هم حلقه تشکیل میدهد و همین خطا را میدهد:

```
4 references
class MyClass
{
    MyClass a = new MyClass();
}
0 references
class Program
{
    0 references
    static void Main()
    {
        MyClass a = new MyClass();
    }
}
```

شاید سوال باشد که ما کی میتوانیم از فراخوانی با کلاس یا با obj استفاده کنیم, مثال زیر آن را شرح میدهد:

در این مثال پارامتر obj counter های ما را میشمارد و هر وقت که به obj دسترسی نداریم یا از بلاک تعریف شده ش آمدیم بیرون, 1 عدد از counter کم میشود.

```
0 references
static class Extention
{
    1 reference
    public static string GenerateNumber(this Random r, uint length)
    {
        if (length == 0) throw new ArgumentOutOfRangeException();
        string s = "";
        for (; length > 0; length--)
        {
            s = s.InsertRandom(r.Next(10).ToString());
        }
        return s;
    }
    1 reference
    public static string InsertRandom(this string s, dynamic text)
    {
        return s.Insert(new Random().Next(s.Length + 1), Convert.ToString(text));
    }
}
8 references
class User : IDisposable
{
    static int counter = 0;
    private string userID, firstName, lastName;
    0 references
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    0 references
    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }
    2 references
    public string FullName => firstName + ' ' + lastName;
    2 references
    public User(string firstName, string lastName)
    {
        userID = new Random().GenerateNumber(10);
        this.firstName = firstName;
        this.lastName = lastName;
        counter++;
    }
    2 references
    public void IntroduceUser() => Console.WriteLine($"User's name with {userID} ID is: {FullName}\n");
    2 references
    public static string GetNumberOfUsers() => $"Users created until now: {counter}\n";
    2 references
    void Bye()
    {
        Console.WriteLine($"User {FullName} with ID of {userID} thank you for being with us.\nFarewell\n");
        counter--;
    }
    0 references
    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Console.WriteLine("I'm Disposer");
        Bye();
    }
    0 references
    ~User()
    {
        Console.WriteLine("I'm Finalizer");
        Bye();
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Sample();
        User u2 = new User("Kamal", "Parizad");
        u2.IntroduceUser();
        Console.WriteLine(User.GetNumberOfUsers());
    }
    1 reference
    static void Sample()
    {
        using (User u1 = new User("Ava", "Tavakoli"))
        {
            u1.IntroduceUser();
            Console.WriteLine(User.GetNumberOfUsers());
        }
    }
}
```

```
C:\WINDOWS\system32\cmd.

User's name with 8975009165 ID is: Ava Tavakoli

Users created until now: 1

I'm Disposer
User Ava Tavakoli with ID of 8975009165 thank you for being with us.
Farewell

User's name with 2651950313 ID is: Kamal Parizad

Users created until now: 1

I'm Finalizer
User Kamal Parizad with ID of 2651950313 thank you for being with us.
Farewell

Press any key to continue . . .
```

-Access Modifier:

سطح دسترسی یا Access Modifier برای آن است که تعیین کنیم کدام یک از اعضای کدام را در چه جاهایی از بلاک برنامه اجازه ی حضور داشته باشند. جدول زیر خلاصه ای از Access Modifier های #C است:

Access Modifier / Caller's location	public	private	protected	internal	protected internal	private protected
Within The Class	✓	✓	✓	✓	✓	✓
Derived Class(Same Assembly)	✓	✗	✓	✓	✓	✓
Non-Derived Class(Same Assembly)	✓	✗	✗	✓	✓	✗
Derived Class(Different Assembly)	✓	✗	✓	✗	✓	✗
Non-Derived Class(Different Assembly)	✓	✗	✗	✗	✗	✗

بعضی Access Modifier ها را تعریف میکنیم...

Public: به معنای دسترسی داشتن از هر جای برنامه میباشد

Private: به معنای دسترسی داشتن فقط از درون کلاس میباشد

Protected: به معنای دسترسی داشتن از داخل کلاس و کلاس فرزندش میباشد

Internal: به معنای دسترسی داشتن فقط از assembly یا namespace ای که همان کلاس درونش است امکان پذیر میباشد

بقیه ی Access Modifier ها را نمیتوان در حد متن توجیه کرد (چون کافی نیست) بخاطر همین تلاش کردم تا در یک تصویر از کد ها همه ی آنها را با هم استفاده کنم تا فرقشان را در حالت های مختلف بگویم. به تصویر زیر با دقت نگاه نمایید:

```
using System;

namespace OtherAssembly2
{
    12 references
    public partial class SuperClass
    {
        5 references
        public static int PublicStaticMethod() { return 0; }
        1 reference
        private static int PrivateStaticMethod() { return 0; }
        3 references
        protected static int ProtectedStaticMethod() { return 0; }
        3 references
        internal static int InternalStaticMethod() { return 0; }
        4 references
        protected internal static int ProtectedInternalStaticMethod() { return 0; }
        2 references
        protected private static int ProtectedPrivateStaticMethod() { return 0; }

        8 references
        public void PublicVoidMethod() { }
        1 reference
        private void PrivateVoidMethod() { }
        3 references
        protected void ProtectedVoidMethod() { }
        3 references
        internal void InternalVoidMethod() { }
        4 references
        protected internal void ProtectedInternalVoidMethod() { }
        2 references
        protected private void ProtectedPrivateVoidMethod() { }

        0 references
        void Access()//Access Within The Class
        {
            //Static
            PublicStaticMethod();
            PrivateStaticMethod();
            ProtectedStaticMethod();
            InternalStaticMethod();
            ProtectedInternalStaticMethod();
            ProtectedPrivateStaticMethod();

            //Void
            PublicVoidMethod();
            PrivateVoidMethod();
            ProtectedVoidMethod();
            InternalVoidMethod();
            ProtectedInternalVoidMethod();
            ProtectedPrivateVoidMethod();
        }
    }
    2 references
    public class DerivedClassSameAssembly : SuperClass
    {
        0 references
        void Access()
        {
            //Static
            PublicStaticMethod();
            //PrivateStaticMethod();
            ProtectedStaticMethod();
            InternalStaticMethod();
            ProtectedInternalStaticMethod();
            ProtectedPrivateStaticMethod();

            //Void
            PublicVoidMethod();
            //PrivateVoidMethod();
            ProtectedVoidMethod();
            InternalVoidMethod();
            ProtectedInternalVoidMethod();
            ProtectedPrivateVoidMethod();
        }
    }
    2 references
    public class NonDerivedClassSameAssembly
    {
        0 references
        void Access()
        {
            //Static
            SuperClass.PublicStaticMethod();
            SuperClass.InternalStaticMethod();
            SuperClass.ProtectedInternalStaticMethod();

            //Void
            SuperClass a = new SuperClass();
            a.PublicVoidMethod();
            a.InternalVoidMethod();
            a.ProtectedInternalVoidMethod();

            //Private, Protected, ProtectedPrivate
        }
    }
}
```

```
using System;
using OtherAssembly2;

namespace OtherAssembly1
{
    2 references
    class DerivedClassDifferentAssembly : SuperClass
    {
        0 references
        void Access()
        {
            //Static
            PublicStaticMethod();
            ProtectedStaticMethod();
            ProtectedInternalStaticMethod();

            //Void
            PublicVoidMethod();
            ProtectedVoidMethod();
            ProtectedInternalVoidMethod();

            //Private, Internal, ProtectedPrivate
        }
    }
    2 references
    class NonDerivedClassDifferentAssembly
    {
        0 references
        void Access()
        {
            //Static
            SuperClass.PublicStaticMethod();

            //Void
            SuperClass a = new SuperClass();
            a.PublicVoidMethod();

            //Private, Protected, Internal, ProtectedInternal, ProtectedPrivate
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main()
        {
            SuperClass a = new SuperClass();
            a.PublicVoidMethod();

            DerivedClassSameAssembly b = new DerivedClassSameAssembly();
            b.PublicVoidMethod();

            NonDerivedClassSameAssembly c = new NonDerivedClassSameAssembly();
            //Nothing to invoke

            DerivedClassDifferentAssembly d = new DerivedClassDifferentAssembly();
            d.PublicVoidMethod();

            NonDerivedClassDifferentAssembly f = new NonDerivedClassDifferentAssembly();
            //Nothing to invoke
        }
    }
}
```

Visual Studio برای درترسی های مختلف علامتی در نظر میگیرد, علامت روی توابع را ببینید:

void SuperClass.PublicVoidMethod()

void SuperClass.PrivateVoidMethod() void SuperClass.ProtectedVoidMethod() void SuperClass.InternalVoidMethod()

void SuperClass.ProtectedInternalVoidMethod() void SuperClass.ProtectedPrivateVoidMethod()

ما هیچ وقت نباید تمام field های کلاسمان را public تعریف کنیم چون باعث میشود کاربر به تمام متغیر های حساس یا غیر حساس دسترسی داشته باشد و بتواند تغییرش دهد از طرفی نمیتوان مطمئن بود که مقدار داخل متغیر جواب درستی است چون ممکن است کاربر متغیری که قرار نبود مقدارش تغییرکند را تغییر داده است. مثلا اگر یه کلاس داشته باشیم که وضعیت تراکنش مالی دانشجو را درونش نگه داری میکند و public باشد, ممکن است کاربر در صورت "بدهکار" بودنش آن متغیر را به "تسویه" عوض کند که باعث میشود با فرض "تسویه" بودن دانشجو کل رفتار کلاس ما عوض شود!

برای همین ما بعضی از متغیر ها را به کاربر دسترسی میدهیم نه همه ی آن را! آن متغیر هایی که مطلقا اجازه دسترسی به کاربر نمیدهند private هستند. در C# مرسوم است که متغیر ها را غیر مستقیم مقدار بدهیم, یعنی تمام field ها را private تعریف کنیم و به همان تعداد Property با دسترسی public تعریف کنیم تا Property آن field را تغییر دهد. در C++ بجای Property از توابع Getter & Setter استفاده میکنند(در C# این روش مرسوم نیست!). به هر تابعی مقدار متغیر های private رو تغییر میدهد, به آنی که مقدار را عوض میکند Set و به آنی که مقدار را نشان میدهد Get میگویند(معمولا از نوع void هستند).

تصویر زیر Property و Getter & Setter را نشان میدهد:

```
2 references
class MyClass
{
    string a;
    2 references
    public string Property //Properties
    {
        get { return a; }  //get => a;
        set { a = value; } //set => a = value;
    }
    2 references
    public string AutoProperty //Auto-Implemented Property (Short Hand)
    {
        get; //get accessor
        set; //set accessor
    }

    int b;
    0 references
    public int WriteOnly
    {
        set { b = value; }
    }
    0 references
    public int ReadOnly
    {
        get { return b; }
    }

    int c;
    0 references
    public void Set(int number) => c = number;
    0 references
    public int Get() => c;
}
0 references
class Program
{
    0 references
    static void Main()
    {
        MyClass a = new MyClass();
        a.Property = "I saved it!";
        a.AutoProperty = "I remember even though if I don't have variable to save!";
        Console.WriteLine(a.Property);
        Console.WriteLine(a.AutoProperty);
    }
}
```

```
C:\WINDOWS\system32\cmd.
I remember even though if I don't have variable to save!
I saved it!
Press any key to continue . . .
```

Property مقدار a را که متغیر Private است تغییر(set) یا نشان(get) میدهد

AutoProperty در C# به صورت اتوماتیک درون خودش داده را ذخیره میکند و داده ی درونش را تغییر میدهد(بودن Get درون Auto-Implemented Properties الزامیست)

ما میتوانیم از طریق Property ها یک مقدار را فقط تغییر(WriteOnly) یا نشان(ReadOnly) دهیم یعنی درون Property فقط قسمت Set یا Get را روی متغیر تعریف کنیم

با ++C مقایسه کنید:

```
class MyClass
{
    string a;
public:
    void Set(string s)
    {
        a = s;
    }
    void Get()
    {
        cout << a;
    }
};

int main()
{
    MyClass a;
    a.Set("I saved it!");
    a.Get();
}
```

```
Microsoft Visual Studio Debu
I saved it!
D:\MyFiles\Programming\C\C++\Projects\Test1\x64\Debug\Test1.exe (process 19412) exited with code 0.
Press any key to close this window . . .
```

اگر بخواهیم درون Property سطح دسترسی بدهیم, فقط میتوانیم برای یکی از get و set ها سطح دسترسی بدهیم و نمتوانیم دسترسی هر دو را یک نوع یا هر دو را متفاوت بگذاریم, اگر هر دو را یک نوع دسترسی بخواهیم بدهیم, به جای get و set باید دسترسی Property را عوض کنیم.
هر دو get و set ها باید پیاده سازی شان یکسان باشند, یعنی اگر یکی از آنها بدنه داشت دیگری هم باید بدنه داشته باشد.

```
1 reference
class SuperClass
{
    string field = "You can't change me from there!";
    3 references
    public string Property { get => field; protected set { field = value; } }
}
2 references
class SubClass : SuperClass
{
    1 reference
    public void ChildChange() => Property = "Child changed its value!";
}
0 references
class Program
{
    0 references
    static void Main()
    {
        SubClass a = new SubClass();
        Console.WriteLine(a.Property);
        a.ChildChange();
        Console.WriteLine(a.Property);
    }
}
```

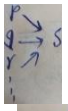
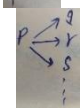
```
C:\WINDOWS\system3:
You can't change me from there!
Child changed its value!
Press any key to continue . . .
```

-Inheritance(وراثت یا ارث بری):

در ارث بری اگر بخواهیم کلاس جدیدی را بر پایه کلاس موجود در پروژه بسازیم که اعضای (class members) آن کلاس را دارا باشد و هم اگر همه ی اعضایی که ما میخواستیم را نداشت, خودمان درون کلاس جدید تعریف کنیم, از ارث بری یا در اصل از ویژگی ارث بری (سلسله مراتب, Hierarchy) استفاده میکنیم. با این کار دیگر نیازی نیست که وقتمان را روی تعریف کردن آن اعضا روی کلاسمان بگزاریم چون از طریق ارث بری داریم از اعضای تعریف شده در کلاس والد استفاده میکنیم که به آن reusability میگویند(کلا به استفاده ی مجدد reusability گفته میشود که inheritance, class, subprogram از آن پیروی میکنند) از طرفی ارث بری یعنی تمرکز بر روی ویژگی مشترک کلاس ها, مثل تعریفی که برای Interface داشتیم, شیر و پلنگ و ببر همه شان گربه سانانند ولی با ظاهر و ویژگی منحصر به فرد.

از بعضی ها هم بر این باورند که ارث بری برای دزدیدن اطلاعات از کلاس استفاده میشود, چون بعضی مواقع برای بعضی از اعضای کلاس, سازنده ی کلاس سطح دسترسی ای را تعریف میکند که فقط از طریق کلاس وارث قابل دسترسی میباشد, یعنی با ارث بری کردن ما میتوانیم بفهمیم که چه اعضایی درون کلاس والد پنهان شده و فقط مختص کلاس وارث تعریف شده است(مثل Protected) که حتی با نمونه گیری هم نمیتوانیم از آن اعضاء استفاده کنیم.

روابط بین کلاس ها با ارث بری یا انواع ارث بری(Inheritance Relationships or Types):

1. **Single Inheritance:**
   یک کلاس فقط از یک کلاس ارث بری میکند.(یک به یک: p→q)
2. **Multi-Level Inheritance:**
   یک کلاس فقط از یک کلاس به صورت پشت سرهم ارث بری میکند. (یک به یک n تایی:
   p→q→r→s→...)
3. **Multiple Inheritance:**
   یک کلاس از چند تا کلاس مختلف ارث بری میکند.(n به یک:) 
4. **Hierarchical Inheritance:**
   چند کلاس مختلف از یک کلاس ارث بری میکنند.(یک به n:) 
5. **Hybrid Inheritance:**
   وقتی که در ارث بری از ترکیب دو یا بیشتر از دو روابط بالا استفاده کنیم, به آن Hybrid Inheritance گفته میشود.

:Multi-Path Inheritance

در این روش یک کلاس به چند کلاس و چند کلاس به یک کلاس ارث بری داده میشود, چون که کلاس آخر دارد از اعضای تکراری کلاس جد را به ارث میبرد, باعث میشود که پیچیدگی و ابهام بیشتری داشته باشد.

:Loop

در این روش کلاس ها جوری ارث بری میکنند که یک حلقه بین خود میسازند. این روش را میتوانیم کد بزنیم اما نمیتوانیم اجرایی کنیم چون موقع اجرا شدنش کامپایلر خطا میدهد. برای این روش نامی داده نشده است ولی ما نامش را برای اشاره کردن بهش Loop گذاشتیم.

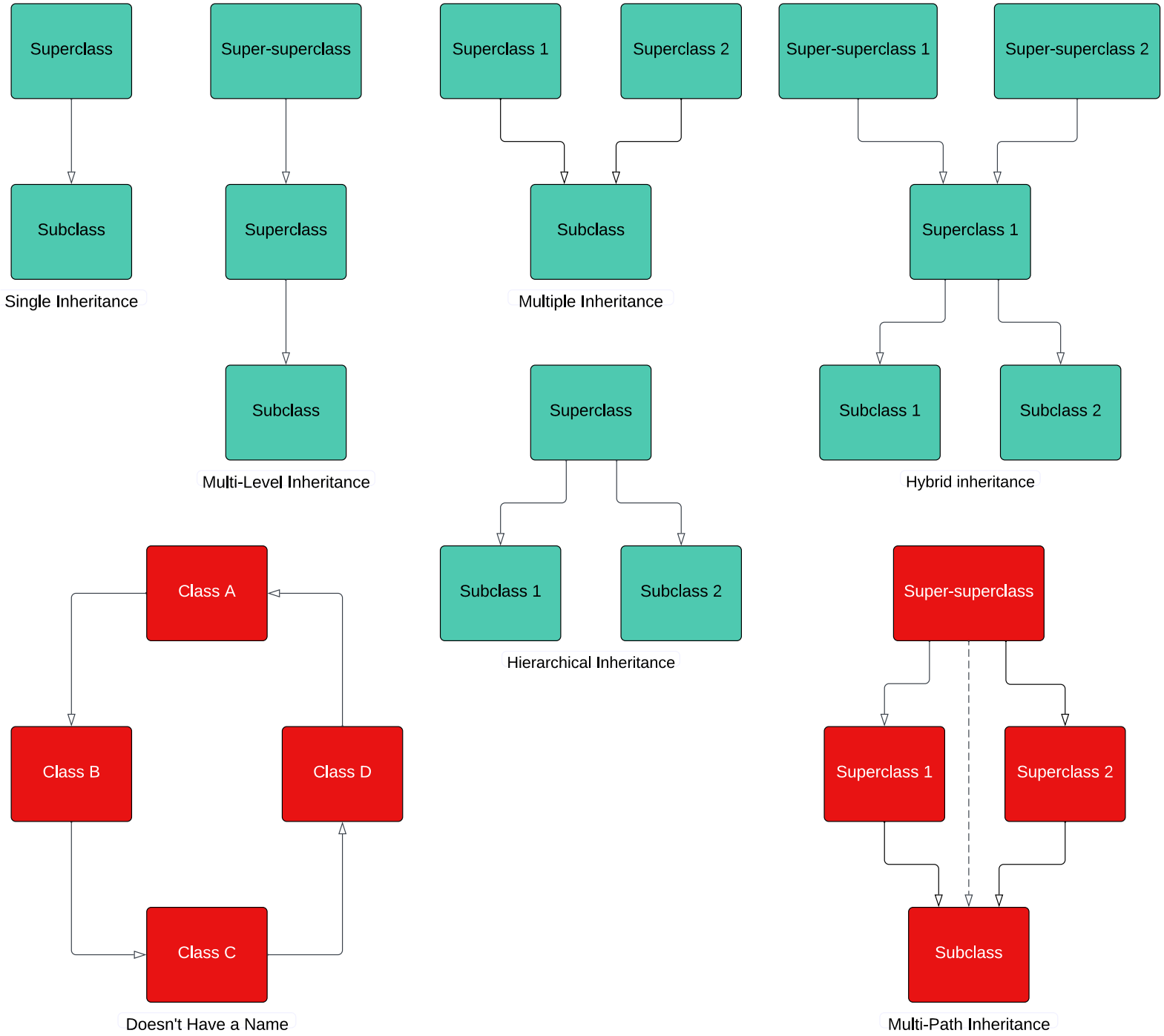
چون در Inheritance باید روابط کلاس ها جوری باشند که حلقه تشکیل ندهند که به آن میگویند "ارث بری باید گراف غیر چرخه ای جهت دار(Directed Acyclic Graph (DAG)) باشد". بخاطر همین Multi-Path را بخاطر ابهامش هنگام تعریف وLoop را بخاطر نقض کردن DAG در روابط ارث بری به رسمیت نمیشناسند و جزء Inheritance Relationships نیستند, بخاطر همین جدا از آن 5 مورد ذکر کردیم.

در C# کلاس ها از single inheritance و interface ها از multiple inheritance پیروی میکنند. بنابراین C# از ارث بری Multiple و Hybrid در کلاس هایش پشتیبانی نمیکند.

جدول زیر لقب هایی برای کلاس والد و فرزند هستند:

class being inherited = کلاسی که ارث میدهد	class that inherits = کلاسی که ارث بری میکند
Base Class = Superclass = Parent Class	Derived Class = Subclass = Child Class
کلاس اشتقاق شده = کلاس والد	کلاس مشتق شده = کلاس فرزند

تصویر زیر نمودار کلی 7 مورد ذکر شده را نشان میدهد:



تصاویر زیر روش Loop و Multi-Path را نشان میدهد:

```cpp
#include <iostream>

using namespace std;

class SupersuperClass
{
public:
    int a;

};
class SuperClass1 : SupersuperClass
{
public:
    int b;
    void M()
    {
        a = 5;
        b = 5;
    }
};
class SuperClass2 : SupersuperClass
{
public:
    int c;
    void M()
    {
        a = 5;
        c = 5;
    }
};
class SubClass : SuperClass1, SuperClass2
{
public:
    int d;
    void M()
    {
        a = 5; //Error!
        b = 5;
        c = 5;
        d = 5;
    }
};

int main()
{
}
```

(field) int SupersuperClass::a
Search Online
"SubClass::a" is ambiguous
Search Online

```
1 reference
class ClassA : ClassD
{
}
```

class Test.ClassA
CS0146: Circular base type dependency involving 'ClassD' and 'ClassA'

```
1 reference
class ClassB : ClassA
{
}
```

class Test.ClassB
CS0146: Circular base type dependency involving 'ClassA' and 'ClassB'

```
1 reference
class ClassC : ClassB
{
}
```

class Test.ClassC
CS0146: Circular base type dependency involving 'ClassB' and 'ClassC'

```
1 reference
class ClassD : ClassC
{
}
```

class Test.ClassD
CS0146: Circular base type dependency involving 'ClassC' and 'ClassD'

```
0 references
class Program
{
    0 references
    static void Main()
    {
    }
}
```

کلاس زیر از Loop به Multi-Level تبدیل شده است:

```
1 reference
class ClassA
{
}
1 reference
class ClassB : ClassA
{
}
1 reference
class ClassC : ClassB
{
}
0 references
class ClassD : ClassC
{
}
0 references
class Program
{
    0 references
    static void Main()
    {
    }
}
```

اصول شی گرایی در ترکیب با ارث بری, ویژگی هایشان بیشتر میشود. مثلا اگر سه کلاس که توابع Constructor, Destructor و Dispose داشته باشند و به صورت Multi-Level از همدیگر ارث بری کنند داشته باشیم, ترتیب فراخوانی توابع به شرح زیر میباشد:

```
7 references
class A : IDisposable
{
    2 references
    public A() => Console.WriteLine("I'm Constructor of Class A");
    0 references
    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Console.WriteLine("I'm Dispose Method of Class A");
    }
    0 references
    ~A() => Console.WriteLine("I'm Desstructor of Class A");
}
7 references
class B : A, IDisposable
{
    2 references
    public B() => Console.WriteLine("I'm Constructor of Class B");
    0 references
    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Console.WriteLine("I'm Dispose Method of Class B");
    }
    0 references
    ~B() => Console.WriteLine("I'm Desstructor of Class B");
}
6 references
class C : B, IDisposable
{
    2 references
    public C() => Console.WriteLine("I'm Constructor of Class C");
    0 references
    public void Dispose()
    {
        GC.SuppressFinalize(this);
        Console.WriteLine("I'm Dispose Method of Class C");
    }
    0 references
    ~C() => Console.WriteLine("I'm Desstructor of Class C");
}
0 references
class Program
{
    0 references
    static void Main()
    {
        using (A a = new A())
        {
        }
        Console.WriteLine();
        using (B b = new B())
        {
        }
        Console.WriteLine();
        using (C c = new C())
        {
        }
        Console.WriteLine();
        A d = new A();
        Console.WriteLine();
        B e = new B();
        Console.WriteLine();
        C f = new C();
        Console.WriteLine();
    }
}
```

```
C:\WINDOWS\system3:
I'm Constructor of Class A
I'm Dispose Method of Class A

I'm Constructor of Class A
I'm Constructor of Class B
I'm Dispose Method of Class B

I'm Constructor of Class A
I'm Constructor of Class B
I'm Constructor of Class C
I'm Dispose Method of Class C

I'm Constructor of Class A

I'm Constructor of Class A
I'm Constructor of Class B

I'm Constructor of Class A
I'm Constructor of Class B
I'm Constructor of Class C

I'm Desstructor of Class C
I'm Desstructor of Class B
I'm Desstructor of Class A
I'm Desstructor of Class B
I'm Desstructor of Class A
I'm Desstructor of Class A
Press any key to continue . . .
```

همان طور که میبینید وقتی که از کلاس فرزند نمونه گیری میکنیم, کامپایلر کلاس والدش را اول اجرایی میکند.

حال بگذارید روی توابع Constructor بیشتر متمرکز شویم. در تصویر بالا دیدیم که کلاس والد در صورت داشتن تابع Constructor اول تابع والد اجرایی میشود بعد فرزندش, حال سوال مطرح میشود که اگر تابع Constructor در ورودی پارامتر بگیرد چگونه تابع Constructor در کلاس فرزند, که باید از تابع کلاس والد تبعیت کند, را بنویسیم.
به تصویر زیر دقت کنید!(توابع Constructor همنام Polymorphism هستند که بعدا رویش متمرکز میشویم)

```
4 references
class SuperClass
{
    0 references
    public SuperClass()
    {
        Console.WriteLine("I'm in SuperClass zero argument.");
    }
    0 references
    public SuperClass(int a)
    {
        Console.WriteLine($"I'm in SuperClass Int32 argument. My argument is: {a}");
    }
    0 references
    public SuperClass(double b)
    {
        Console.WriteLine($"I'm in SuperClass Deouble argument. My argument is: {b}");
    }
}
9 references
class SubClass : SuperClass
{
    1 reference
    public SubClass()
    {
        Console.WriteLine("I'm in SubClass zero argument.");
    }
    1 reference
    public SubClass(int a)
    {
        Console.WriteLine($"I'm in SubClass Int32 argument. My argument is: {a}");
    }
    1 reference
    public SubClass(double b)
    {
        Console.WriteLine($"I'm in SubClass Double argument. My argument is: {b}");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        SubClass a = new SubClass();
        Console.WriteLine();
        SubClass b = new SubClass(5);
        Console.WriteLine();
        SubClass c = new SubClass(1.5);
        Console.WriteLine();
    }
}
```

```
C:\WINDOWS\system32\cmd.

I'm in SuperClass zero argument.
I'm in SubClass zero argument.

I'm in SuperClass zero argument.
I'm in SubClass Int32 argument. My argument is: 5

I'm in SuperClass zero argument.
I'm in SubClass Double argument. My argument is: 1.5

Press any key to continue . . .
```

هر باری که برنامه نویس تابع Constructor دیگری را در کلاس فرزند با نمونه گیری صدا میزند, کامپایلر می آید اول Constructor کلاس والد را صدا میزند ولی همیشه تابعی که ورودی ندارد را صدا میزند. حال اگر ما خواستیم کامپایلر به تابع مشخصی که ورودی دارد ارجاع داده شود چه کنیم؟

در تصویر زیر کامپایلر وقتی که دارد از کلاس فرزند نمونه میگیرد چون تابع سازنده درونش است آن را فراخوانی میکند اما چون ما با base() : تابع سازنده ی فرزند را به تابع مشخص شده ی سازنده ی والد نسبت دادیم اول سازنده ی والد را فراخوانی میکند بعد کد های درون سازنده ی فرزند را اجرایی میکند. اگر base() : را ننویسیم یا درونش آرگمان نگذاریم, به طور پیش فرض کامپایلر تابع سازنده ای که پارامتر ورودی ندارد را فراخوانی میکند(چرایی تکرار شدن یک تابع در تصویر قبل)

```
7 references
class SuperClass
{
    1 reference
    public SuperClass()
    {
        Console.WriteLine("I'm in SuperClass zero argument.");
    }
    1 reference
    public SuperClass(int a)
    {
        Console.WriteLine($"I'm in SuperClass Int32 argument. My argument is: {a}");
    }
    1 reference
    public SuperClass(double b)
    {
        Console.WriteLine($"I'm in SuperClass Deouble argument. My argument is: {b}");
    }
}
9 references
class SubClass : SuperClass
{
    1 reference
    public SubClass() : base()
    {
        Console.WriteLine("I'm in SubClass zero argument.");
    }
    1 reference
    public SubClass(int a) : base(a)
    {
        Console.WriteLine($"I'm in SubClass Int32 argument. My argument is: {a}");
    }
    1 reference
    public SubClass(double b) : base(b)
    {
        Console.WriteLine($"I'm in SubClass Double argument. My argument is: {b}");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        SubClass a = new SubClass();
        Console.WriteLine();
        SubClass b = new SubClass(5);
        Console.WriteLine();
        SubClass c = new SubClass(1.5);
        Console.WriteLine();
    }
}
```

```
C:\WINDOWS\system32\cmd.
I'm in SuperClass zero argument.
I'm in SubClass zero argument.

I'm in SuperClass Int32 argument. My argument is: 5
I'm in SubClass Int32 argument. My argument is: 5

I'm in SuperClass Deouble argument. My argument is: 1.5
I'm in SubClass Double argument. My argument is: 1.5

Press any key to continue . . .
```

یعنی اگر کلاس والد تابع سازنده بدون پارامتر ورودی نداشته باشد حق ندارید base را جلوی تابع سازنده نگذارید یا بدون آرگمان تعریف کنید.
تصویر روبه رو را ببینید.

```
4 references
class SuperClass
{
    1 reference
    public SuperClass(int a)
    {
        Console.WriteLine($"I'm in SuperClass Int32 argument. My argument is: {a}");
    }
    1 reference
    public SuperClass(double b)
    {
        Console.WriteLine($"I'm in SuperClass Deouble argument. My argument is: {b}");
    }
}
2 references
class SubClass : SuperClass
{

    0 references
    public SubClass()
    {
    }
    0 references
    public SubClass(int a) : base()
    {
    }
}
```

حال به مثال زیر که در واقعیت زده شده است و با قابلیت بالا ادغام شده است توجه کنید. کلاس والد چون سازنده اش پارامتر ورودی دارد تمام کلاسای فرزند هم باید base ای داشته باشند که آرگمان به داشته باشد و به آن مقدار بدهند بخاطر همین پیشنهاد میشود سازنده ی فرزندان را هم همچون سازنده ی والد بهشان پارامتر های ورودی بدهید.

```
0 references
static class Extention
{
    2 references
    public static string GenerateNumber(this Random r, uint length)
    {
        if (length == 0) throw new ArgumentOutOfRangeException();
        string s = "";
        for (; length > 0; length--)
        {
            s = s.InsertRandom(r.Next(10).ToString());
        }
        return s;
    }
    1 reference
    public static string InsertRandom(this string s, dynamic text) => s.Insert(new Random().Next(s.Length + 1), Convert.ToString(text));
}
5 references
abstract class User
{
    private string id, firstName, lastName, birthDate;

    0 references
    public string ID => id;
    0 references
    public string FirstName => firstName;
    0 references
    public string LastName => lastName;
    2 references
    public string FullName => firstName + ' ' + lastName;
    0 references
    public string BirthDate => birthDate;

    2 references
    public string Age
    {
        get
        {
            int a = int.Parse(birthDate.Remove(4));
            a = (a >= 2023 ? DateTime.Now.Year : int.Parse(DateTime.Now.ToString("yyyy"))) - a;
            return a.ToString();
        }
    }
    2 references
    public User(string id, string firstName, string lastName, string birthDate)
    {
        this.id = id;
        this.firstName = firstName.Trim();
        this.lastName = lastName.Trim();
        this.birthDate = birthDate;
    }
}
3 references
class Seller : User
{
    private string sellerID;
    1 reference
    public string SellerID => sellerID;
    1 reference
    public Seller(string id, string firstName, string lastName, string birthDate) : base(id, firstName, lastName, birthDate) => sellerID = new Random().GenerateNumber(10);
}
3 references
class Client : User
{
    private string clientID;
    1 reference
    public string ClientID => clientID;
    1 reference
    public Client(string id, string firstName, string lastName, string birthDate) : base(id, firstName, lastName, birthDate) => clientID = new Random().GenerateNumber(10);
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Seller objSeller = new Seller("0048043513", "Parviz", "Lucas", "1372/12/08");
        Client objClient = new Client("0130747831", "Poneh", "Suarez", "1368/03/17");
        Console.WriteLine($"Client {objClient.FullName} 's ID is {objClient.ClientID} who's {objClient.Age} years old.");
        Console.WriteLine($"Seller {objSeller.FullName} 's ID is {objSeller.SellerID} who's {objSeller.Age} years old.");
    }
}
```

```
C:\WINDOWS\system32\cmd.
Client Poneh Suarez 's ID is 7708033269 who's 34 years old.
Seller Parviz Lucas 's ID is 3066924367 who's 30 years old.
Press any key to continue . . .
```

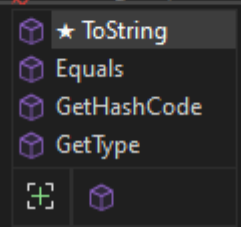
-Hierarchy(سلسله مراتب):

وقتی که شما از کلاسی ارث بری میکنید, علاوه بر داشتن اعضای منحصر به فرد کلاس فرزند, بعضی از اعضای کلاس والد را هم دارید. به اعضایی که از کلاس والد به کلاس فرزند توسط ارث بری داده میشود(ارثیه) Hierarchy , که به معنی سلسله مراتب است, میگویند. ویژگی ارث بری Hierarchy است.

در C# Default Hierarchy داریم که برای هر کلاسی توابع ,ToString(), Equals(), GetHashCode() GetType() را به آن کلاس ارث میدهد.

```
2 references
class MyClass
{

}
0 references
class Program
{
    0 references
    static void Main()
    {
        MyClass a = new MyClass();
        a.ToString();
    }
}
```



تصویر روبه رو توابع پیش فرض یا همان Default Hierarchy را که C# برای کلاس در نظر گرفته است را نشان میدهد.

تصویر زیر کلاس سیستمی Object را نشان میدهد, Default Hierarchy از کلاس تصویر زیر می آیند.

```
namespace System
{
    [Serializable]
    [ClassInterface(ClassInterfaceType.AutoDual)]
    [ComVisible(true)]
    [__DynamicallyInvokable]
    public class Object
    {
        [ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
        [NonVersionable]
        [__DynamicallyInvokable]
        public Object()
        {
        }

        [__DynamicallyInvokable]
        public virtual string ToString()
        {
            return GetType().ToString();
        }

        [__DynamicallyInvokable]
        public virtual bool Equals(object obj)
        {
            return RuntimeHelpers.Equals(this, obj);
        }

        [__DynamicallyInvokable]
        public virtual int GetHashCode()
        {
            return RuntimeHelpers.GetHashCode(this);
        }

        [MethodImpl(MethodImplOptions.InternalCall)]
        [SecuritySafeCritical]
        [__DynamicallyInvokable]
        public extern Type GetType();
```

اگر این توابع را بدون بازنویسی فراخوانی کنیم, کامپایلر همان کد های پیش فرض درونشان را اجرایی میکند. تصویر زیر آن را نشان میدهد:

```
0 references
static void Main()
{
    MyClass a = new MyClass();
    Console.WriteLine(a.ToString());
}
```

```
C:\WINDOWS\system3
Test.MyClass
Press any key to continue . . .
```

ما میتوانیم با Override کردن این توابع کد های خودمان را درونشان بگذاریم و کد های پیش فرض را باطل کنیم (بعدا درباره ی چگونگی این کار در Polymorphism حرف میزنیم)

هنگام ارث بری گرفتن هم Default Hierarchy C# را به کلاس والد میدهد:

```
1 reference
class SuperClass
{
}
0 references
class SubClass : SuperClass
{
    0 references
    void Test()
    {
        base.MemberwiseClone();
    }
}
0 references
class Program
{
    0 references
    static vo
    {

    }
}
```

Equals
GetHashCode
GetType
MemberwiseClone
ToString

-Hierarchy Relationships:

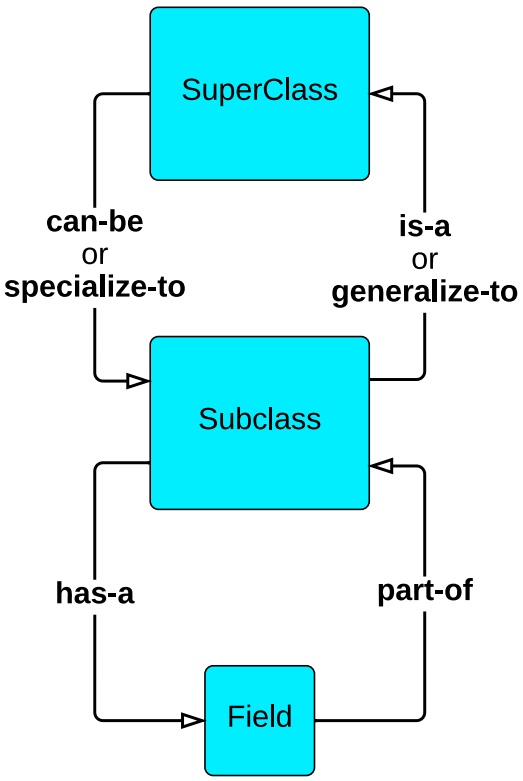
برای آنکه در یک جمله بتونیم بگیم کدوم کلاس از کدوم کلاس ارث بری میکنه یا کدوم کلاس چه متغیر یا اطلاعاتی رو تو خودش ذخیره میکنه از hierarchy relationship استفاده میکنیم.

مثال زیر را توجه کنید:

Canine = سگ سانان – Pelage = خز روباه – Feline = گربه سانان – Gender = جنسیت

1. Canine Superclass
   Fox Subclass [Fox (is a/generalize to) Canine]
   [Canine (can be/specialize to) Fox]
   Pelage Color Field Variable [Fox has a Pelage Color]
   [Pelage Color is part of Fox]
2. Feline Superclass
   Lion Subclass [Lion (is a/generalize to) Feline]
   [Feline (can be/specialize to) Lion]
   Gender Field Variable [Lion has a Gender]
   [Gender is part of Lion]

خلاصه ی مثال بالا را به شکل نمودار در تصویر روبه رو آورده ایم:



همانطور که میبینید...

can-be, specialize-to, has-a از بالا به پایین و-is-a, generalize

to, part-of از پایین به بالا هستند

-Polymorphism(چند ریختی):

Polymorphism کلمه ی یونانی که از دو کلمه ی Poly به معنای چند تایی (مثل کلمه ی Polymer = بسیار) و Morph به معنای حالت(ریخت) تشکیل شده است. به طور مثال کلمه ای که چند تا معنی دارد و معنیش به جمله بستگی دارد (که بهش میگن Homograph) مثل کلمه ی bow , حال ممکن است field یا method های کلاس هم, همنام باشند. بگزارید اول از همه از آسان ترینشان یعنی field شروع کنیم...

-Polymorphism in Fields:

ما میدانیم که نمیتوانیم از یک پارامتر تعریف شده از قبل در جاهای دیگر با data type دیگر استفاده کنیم ولی اگر دو کلاس داشته باشیم که درون هر کدام دو پارامتر همنام داشته باشیم, در فراخوانی آن پارامتر ها اول کدام صدا زده میشود؟ به مثال زیر توجه کنید:

```
3 references
public class Parent
{
    public string field = "Parent: I have a data";
    //public string field = "ramdom text"; Error: Parent already contains definition for field
    1 reference
    public void ParentChanged()
    {
        field = "Parent: I changed my data!";
    }
}
2 references
public class Child : Parent
{
    public string field = "Child: I'm hiding Parent's data from you!"; // hiding base.field
    //public new string field = "Child: I'm hiding Parent's data from you!";
    1 reference
    public void ChildChanged()
    {
        field = "Child: I changed my data!";
        base.field = "Parent: Child HACKED ME!";
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Parent objParent1 = new Parent();
        Child objChild1 = new Child();
        Console.WriteLine(objParent1.field);
        Console.WriteLine(objChild1.field);
        objParent1.ParentChanged();
        objChild1.ChildChanged();
        Console.WriteLine(objParent1.field);
        Console.WriteLine(objChild1.field);
    }
}
```

```
C:\WINDOWS\system3:
Parent: I have a data
Child: I'm hiding Parent's data from you!
Parent: I changed my data!
Child: I changed my data!
Press any key to continue . . .
```

همان طور که میبینید درون کلاس والد نمیتوانیم دو تا پارامتر همنام تعریف کنیم چون به ما خطا میدهد ولی وقتی که کلاس فرزند ارث بری میکند و پارامتر ها همنام هستند, پارامتر کلاس فرزند پارامتر کلاس والد را پنهان(hide) یا غیر قابل دسترس میکند و تنها راه دسترسی داشتن به آن از طریق کلمه ی base است(این کلمه اگر پرانتز داشته باشد به کامپایلر میگوید که Constructor را صدا بزن ولی اینجا کلاس والد را صدا میزند) که فقط میتواند در local کلاس فرزند صدا زده شود تا به کامپایلر بگوید که منظورمان پارامتر درون کلاس والد است(بیرون از کلاس فرزند قابل قابل دسترسی نیست).

وقتی که یک پارامتر یا تابعی تکرار شده باشد و باعث شود پارامتر یا تابع کلاس والد hide شود, بهتر است از کلمه ی new قبل از data type استفاده کنیم(به کامنت زیر پارامتر کلاس فرزند در تصویر دقت کنید)

حال که فهمیدیم تکراری بودن پارامتر در کلاس ها باعث میشود دیگری hide شود, یک سوال بوجود می آید, اگر نمونه های کلاس های والد و فرزند را درون یکدیگر بریزیم چه اتفاقی می افتد؟ طبق Hierarchy , ویژگی های کلاس والد باید درون فرزند برود ولی اگر ما نمونه ی فرزند را درون نمونه ی والد بریزیم یا بالعکس چه میشود؟

```
5 references
public class Parent
{
    public string field = "Parent: I have a data";
    5 references
    public int ParentMethod() { return 0; }
    1 reference
    public void ParentChanged()
    {
        field = "Parent: I changed my data!";
    }
}
3 references
public class Child : Parent
{
    public new string field = "Child: I'm hiding Parent's data from you!";
    2 references
    public int ChildMethod() { return 0; }
    1 reference
    public void ChildChanged()
    {
        field = "Child: I changed my data!";
        base.field = "Parent: Child HACKED ME!";
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Parent objParent = new Parent();
        Child objChild = new Child();

        Parent objParentToParent = objParent;
        //Child objParentToChild = (Child)objParent; Run-Time Error
        Parent objChildToParent = objChild;
        Child objChildToChild = objChild;

        //Hierarchy of Objects
        objParent.ParentMethod();
        objChild.ParentMethod();
        objChild.ChildMethod();
        objParentToParent.ParentMethod();
        //objParentToChild
        objChildToParent.ParentMethod();
        objChildToChild.ParentMethod();
        objChildToChild.ChildMethod();

        Console.WriteLine(objParent.field);
        Console.WriteLine(objChild.field);
        Console.WriteLine(objParentToParent.field);
        //Console.WriteLine(objParentToChild.field);
        Console.WriteLine(objChildToParent.field);
        Console.WriteLine(objChildToChild.field);

        Console.WriteLine();

        objParent.ParentChanged();
        objChild.ChildChanged();

        Console.WriteLine(objParent.field);
        Console.WriteLine(objChild.field);
        Console.WriteLine(objParentToParent.field);
        //Console.WriteLine(objParentToChild.field);
        Console.WriteLine(objChildToParent.field);
        Console.WriteLine(objChildToChild.field);
    }
}
```

```
C:\WINDOWS\system3:

Parent: I have a data
Child: I'm hiding Parent's data from you!
Parent: I have a data
Parent: I have a data
Child: I'm hiding Parent's data from you!

Parent: I changed my data!
Child: I changed my data!
Parent: I changed my data!
Parent: Child HACKED ME!
Child: I changed my data!
Press any key to continue . . .
```

همان طور که میبینید...
ما هیچ وقت حق نداریم که نمونه ی کلاس والد را به فرزند بدهیم(نمونه ی objParentToChild خطا میدهد).
ویژگی های نمونه ی کلاس فرزند, در صورت تبدیل کردنش به کلاس والد, درون نمونه ی تبدیل شده دیگر ویژگی های کلاس فرزند نیستند(زیر کامنت Hierarchy) که این یعنی نمونه ی کلاس فرزند وقتی که پارامتر کلاس والد را پنهان میکرد بعد از تبدیل شدن به کلاس والد دیگر نمیتواند پارامتر کلاس والد را پنهان کند.
ما در این مثال فقط دو نمونه از کلاس های والد و فرزند گرفتیم و آن را به نمونه های دیگر کپی کردیم, پس پارامتر همه ی نمونه ها باید برابر باشند ولی وقتی که پارامتر های دو نمونه اصلی را عوض میکنیم, پارامتر های تمام نمونه های کپی شده با اینکه به آنها این دستور را نداده ایم عوض میشوند. این بخاطر reference data type بودن کلاس ها است. یعنی وقتی که یک پارامتری به یک داده ای لینک میشود, آن داده را هر کسی هر جای برنامه تغییرش دهد روی تمام پارامتر ها اعمال میشود(آرایه, کلاس, interface و delegate از همین نوع data type هستند).

-Polymorphism in Methods:

انواع چند ریختی(Types of Polymorphism):

ما دوتا حالت در پروسه ی اجرا شدن برنامه داریم...

```
0 references
class Program
{
    0 references
    static void Main()
    {
        string s = 564;
    }
}
```

struct System.Int32
Represents a 32-bit signed integer.

CS0029: Cannot implicitly convert type 'int' to 'string'

Compile Time وقتیست که برنامه میخواهد اجرا شود, همان موقعی که کد های برنامه نویس به کدهای Binary تبدیل میشود, اگر برنامه خطا داشته باشد کامپایل نمیشود ولی به محض رفع آن خطا ها برنامه کامپایل میشود

Run Time وقتیست که برنامه از حالت کامپایل در آمده است و در لحظه عملیات را انجام میدهد ,مثلا یک تابعی که تمام data type ها را میگیرد و یک عملیاتی را انجام میدهد, به خطاهایی که برنامه در حال اجرا است اتفاق میوفتد خطا های Run Time میگویند .

```
0 references
class Program
{
    0 references
    static void Main()
    {
        Console.WriteLine(Console.ReadLine().Substring(5));
        Console.WriteLine(Console.ReadLine().Substring(5));
    }
}
```

```
C:\WINDOWS\system3:
123456789
6789
12

Unhandled Exception: System.ArgumentOutOfRangeEx
ception: startIndex cannot be larger than length
 of string.
```

حال در Polymorphism دو نوع توابع داریم که شامل این دو حالت میشوند....

1. Compile-Time Method:

شامل توابعی میشود که ورودی ها و بدنه(Implementation) مختلفی دارند ولی Member Modifier, return type, identifier های یکسانی دارند. به این توابع overloaded methods گفته میشود. بسته به ورودی تابع رفتار تابع عوض میشود(این کار را کامپایلر تشخیص میدهد که طبق ورودی داده شده کدام تابع را صدا بزند).

```
using System;
using System.Web.UI.DataVisualization.Charting;

namespace OtherAssembly1
{
    5 references
    public class Math
    {
        3 references
        public static double Factorial(double number)
        {
            return (number > 0) ? new Chart().DataManipulator.Statistics.GammaFunction(number + 1) : double.NaN;
        }
        2 references
        public static int Factorial(int number)
        {
            if (number < 0) return 0;

            int b = 1;
            for (int i = 2; i <= number; i++)
            {
                b *= i;
            }
            return b;
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main()
        {
            Console.WriteLine(Math.Factorial(0.5));
            Console.WriteLine(Math.Factorial(5));
            Console.WriteLine(Math.Factorial(5.001));
            Console.WriteLine(Math.Factorial(5.999));
            Console.WriteLine(Math.Factorial(6));
        }
    }
}
```

```
C:\WINDOWS\system3:
0.886226925452754
120
120.204919767127
718.652912302178
720
Press any key to continue . . .
```

```
Math.Factorial()
double Math.Factorial(double number) (+ 1 overload)
CS1501: No overload for method 'Factorial' takes 0 arguments
Show potential fixes (Alt+Enter or Ctrl+.)
```

در تصویر روبه رو نشان میدهد که تابع Factorial(double number) بجز خودش یک دونه Overload دیگر به شکل Factorial(int number) هم دارد.

```
Console.WriteL (double number):double (5.999)
Console.WriteL (int number):int (6));
Math.Factorial(,)
▲ 1 of 2 ▼ double Math.Factorial(double number)
```

```
Console.WriteL (double number):double
Console.WriteL (int number):int
Math.Factorial(,)
▲ 2 of 2 ▼ int Math.Factorial(int number)
```

2. Run-Time Method:

شامل توابعی که ورودی, Access & Member Modifier, return type, identifier های یکسانی دارند. به این توابع overrided methods گفته میشود. چون توابع virtual و abstract قابلیت override شدن را دارند, ما نمیتوانیم آنها را private تعریف کنیم. فقط prop و method میتوانند virtual باشند و virtual field نداریم.

اگر نمونه های کلاس والد و فرزند که تابع همنام دارند را به همدیگر بدهیم چه میشود؟ Hierarchy یکسانی با مبحث قبلمان(field) دارد ولی در موضوع hide کردن فرق پیدا خواهد کرد!

```
5 references
public class Parent
{
    3 references
    public void DoWork() => Console.WriteLine("Parent");
}
3 references
public class Child : Parent
{
    2 references
    public new void DoWork() => Console.WriteLine("Child");
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Parent objParent = new Parent();
        Child objChild = new Child();

        Parent objParentToParent = objParent;
        //Child objParentToChild = (Child)objParent;
        Parent objChildToParent = objChild;
        Child objChildToChild = objChild;

        objParent.DoWork();
        objChild.DoWork();
        objParentToParent.DoWork();
        objChildToParent.DoWork();
        objChildToChild.DoWork();
    }
}
```

```
C:\WINDOWS\system3
Parent
Child
Parent
Parent
Child
Press any key to continue . . .
```

همچنان نمیتوانیم نمونه ی والد را به فرزند بریزیم.

```
5 references
public class Parent
{
    6 references
    public virtual void DoWork() => Console.WriteLine("Parent");
}
3 references
public class Child : Parent
{
    6 references
    public override void DoWork() => Console.WriteLine("Child");
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Parent objParent = new Parent();
        Child objChild = new Child();

        Parent objParentToParent = objParent;
        //Child objParentToChild = (Child)objParent;
        Parent objChildToParent = objChild;
        Child objChildToChild = objChild;

        objParent.DoWork();
        objChild.DoWork();
        objParentToParent.DoWork();
        objChildToParent.DoWork();
        objChildToChild.DoWork();
    }
}
```

```
C:\WINDOWS\system3
Parent
Child
Parent
Child
Child
Press any key to continue . . .
```

با نگاه کردن به این دو تصویر میفهمیم که کامپایلر بسته به چه نوع بودن تابع واکنشش را عوض میکند!

تصویر زیر(صفحه ی بعد) اولیوت هر کدام از توابع به هنگام تبدیل شدن به کلاس های دیگر را نشان میدهد....

```
3 references
public class Virtual
{
    13 references
    public virtual void Invoke() => Console.WriteLine("virtual");
}
5 references
public class Override : Virtual
{
    13 references
    public override void Invoke() => Console.WriteLine("override");
}
6 references
public class SealedOverride : Override
{
    13 references
    public sealed override void Invoke() => Console.WriteLine("sealed override");
}
6 references
public class New : SealedOverride
{
    2 references
    public new void Invoke() => Console.WriteLine("new");
}

0 references
class Program
{
    0 references
    static void Main()
    {
        Virtual virtualMethod = new Virtual();
        Override overrideMethod = new Override();
        SealedOverride sealedOverrideMethod = new SealedOverride();
        New newMethod = new New();
        virtualMethod.Invoke();
        overrideMethod.Invoke();
        sealedOverrideMethod.Invoke();
        newMethod.Invoke();

        Console.WriteLine();

        virtualMethod = new Override();
        overrideMethod = new Override();
        virtualMethod.Invoke();
        overrideMethod.Invoke();

        Console.WriteLine();

        virtualMethod = new SealedOverride();
        overrideMethod = new SealedOverride();
        sealedOverrideMethod = new SealedOverride();
        virtualMethod.Invoke();
        overrideMethod.Invoke();
        sealedOverrideMethod.Invoke();

        Console.WriteLine();

        virtualMethod = new New();
        overrideMethod = new New();
        sealedOverrideMethod = new New();
        newMethod = new New();
        virtualMethod.Invoke();
        overrideMethod.Invoke();
        sealedOverrideMethod.Invoke();
        newMethod.Invoke();
    }
}
```

C:\WINDOWS\system3:

```
virtual
override
sealed override
new

override
override

sealed override
sealed override
sealed override

sealed override
sealed override
sealed override
new
Press any key to continue . . .
```

خب تا اینجا ما فهمیدیم overload تابع ما را سر بارگذاری میکند و override کد تابع قبلی ما را باطل میکند.
Overload برای مواقعیست که رفتار ما بسته به نوع دریافت ورودی ما از کاربر عوض میشود.
Override برای مواقعیست که تابع ما یک حالت پیش فرض دارد که ممکن است در ارث بری گرفته شدن عوض بشود.
حال میخواهیم درباره ی overload کردن عملیات ها صحبت کنیم, عملیات هایی که بین دو نمونه انجام میشود...
جدول های زیر منظورمان از عملیات ها چیست و کدام عملیات ها قابل بازنویسی هستند را میگویند...

Overloadable Operators	Non-Overloadable Operators
+x, -x, !x, ~x, ++, --, true, false	x && y, x || y
x + y, x – y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x >>> y	a[i], a?[i]
x == y, x != y, x < y, x > y, x <= y, x >= y	(T)x
	+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=
	^x, x = y, x.y, x?.y, c ? t : f, x ?? y, ??= y, x..y, x->y, =>, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, switch, typeof, with

ما میتوانیم Overloadable Operators را بین obj ها بر قرار سازیم تا کار مان آسانتر شود. ما برای مثال ماتریس را بر حسب کلاس نوشتیم. در آن از Overloadable Operators و Override ToString() هم استفاده کردیم...

```
45 references
public class ComplexNumber
{
    double real;
    double image;
    10 references
    public ComplexNumber(double real = 0, double imaginary = 0)
    {
        this.real = real;
        image = imaginary;
    }
    2 references
    public ComplexNumber(ComplexNumber cN)
    {
        real = cN.real;
        image = cN.image;
    }
    0 references
    public double Real { get => real; set { real = value; } }
    0 references
    public double Imaginary { get => image; set { image = value; } }

    4 references
    public double Absolute() => Math.Sqrt(Math.Pow(real, 2) + Math.Pow(image, 2));
    3 references
    public ComplexNumber Conjugate() => new ComplexNumber(real, -image);
    1 reference
    public ComplexNumber[] SquareRoot()
    {
        double abs = Absolute(), r = Math.Sqrt((abs + real) / 2), i = Math.Sqrt((abs - real) / 2);
        return new ComplexNumber[2] { new ComplexNumber(r, i), -new ComplexNumber(r, i) };
    }

    0 references
    public static ComplexNumber operator +(ComplexNumber a) => a;
    3 references
    public static ComplexNumber operator +(ComplexNumber a, ComplexNumber b)
    {
        return new ComplexNumber(a.real + b.real, a.image + b.image);
    }

    2 references
    public static ComplexNumber operator -(ComplexNumber a)
    {
        return new ComplexNumber(-a.real, -a.image);
    }
    2 references
    public static ComplexNumber operator -(ComplexNumber a, ComplexNumber b) => a + -b;

    3 references
    public static ComplexNumber operator *(ComplexNumber a, double b)
    {
        return new ComplexNumber(a.real * b, a.image * b);
    }
    3 references
    public static ComplexNumber operator *(double a, ComplexNumber b) => b * a;
    6 references
    public static ComplexNumber operator *(ComplexNumber a, ComplexNumber b)
    {
        return new ComplexNumber(a.real * b.real - a.image * b.image, a.real * b.image + a.image * b.real);
    }

    1 reference
    public static ComplexNumber operator /(ComplexNumber a, double b) => a * (1 / b);
    0 references
    public static ComplexNumber operator /(ComplexNumber a, ComplexNumber b) => a * (b ^ -1);

    5 references
    public static ComplexNumber operator ^(ComplexNumber a, short power)
    {
        if (power == 0) return new ComplexNumber(1, 0);
        else if (power == 1) return a;
        else if (power > 1)
        {
            ComplexNumber tmp = new ComplexNumber(a);
            short odd = (short)(power % 2), even = (short)(power / 2);
            if (even > 0)
            {
                for (short i = 0; i < even; i++)
                {
                    tmp *= tmp;
                }
            }
            if (odd == 1) tmp *= a;

            return tmp;
        }
        else
        {
            ComplexNumber tmp = new ComplexNumber(a);
            tmp = tmp.Conjugate() / Math.Pow(tmp.Absolute(), 2);
            return tmp ^ (short)-power;
        }
    }

    1 reference
    public override string ToString()
    {
        string s = "";
        s += real != 0 ? real.ToString() : "";
        if (image > 0) s += '+';
        s += image != 0 ? image.ToString() + 'i' : "";
        return s != "" ? s : "0";
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        ComplexNumber x = new ComplexNumber(2, 5), y = new ComplexNumber(3, 4);
        Console.WriteLine("x = {0}", x);
        Console.WriteLine("y = {0}", y);
        Console.WriteLine("x' = {0}", x.Conjugate());
        Console.WriteLine("y' = {0}", y.Conjugate());
        Console.WriteLine("|x| = {0}", x.Absolute());
        Console.WriteLine("|y| = {0}", y.Absolute());
        Console.WriteLine("√x = {0}", string.Join(", ", x.SquareRoot().Select(z => z.ToString())));
        Console.WriteLine();
        Console.WriteLine("x + y = {0}", x + y);
        Console.WriteLine("x - y = {0}", x - y);
        Console.WriteLine("x * y = {0}", x * y);
        Console.WriteLine("-2y * 3x = {0}", -2 * y * 3 * x);
        Console.WriteLine("-2y + 3x = {0}", -2 * y + 3 * x);
        Console.WriteLine("x^2 = {0}", x ^ 2);
        Console.WriteLine("y^-1 = {0}", y ^ -1);
        Console.WriteLine("(xy)^3 - x: {0}", ((x * y) ^ 3) - x);
    }
}
```

C:\WINDOWS\system32\cmd.

```
x = 2+5i
y = 3+4i
x' = 2-5i
y' = 3-4i
|x| = 5.3851648071345
|y| = 5
√x = 1.9216093264676+1.30099285300391i, -1.9216093264676-1.30099285300391i

x + y = 5+9i
x - y = -1+1i
x * y = -14+23i
-2y * 3x = 84-138i
-2y + 3x = +7i
x^2 = -21+20i
y^-1 = 0.12-0.16i
(xy)^3 - x: 19472+1352i
Press any key to continue . . .
```

تصویر قبلی صرفا جنبه ی تدریسی داشت!

اصولی نیست که ما ریاضیات(مثل ماتریس و اعداد موهومی) یا موجودیت هایی را که اطلاعات اولیه شان برایمان مهم است, به عنوان کلاس تعریفشان کنیم چون اگر یک تابع Destructor به این کلاس اضافه کنیم, که این یعنی تمام obj هایی را که ساختیم (با اینکه از بلوکشان خارج شدیم) درون حافظه, تا بسته شدن برنامه باقی بمانند و سرعت را کند نماید!

به طوری که اگر یک obj را بتوان 5 ببریم(a^5) و اگر کلاس تابع Destructor داشته باشد, تابع 5 بار فراخوانی میشود!

بهتر است به جای class از struct که از نوع value datatype است استفاده کنیم!

ما در انواع کلاس ها انواع توابع داریم که آنها را اینجا میگوییم...

انواع توابع(Types of Methods):

1. **Static Method:**
   توابعی هستند که همیشه مقداری را بر میگردانند
2. **Intance Method(void):**
   توابعی هستند که هیچ وقت مقداری بر نمیگردانند
3. **Virtual Method:**
   توابع مجازی یا توابعی که بعدا ممکن است کد های داخل بدنه شان عوض شود(توابع پیش فرض)
4. **Extention Method:**
   نوعی توابع بازگشتی هستند که فقط درون کلاس Static میتوان تعریف نمود و همیشه یکی از پارامتر هایشان با this شروع میشود که به معنی این است که بعد از this هر data type ای بود روی آن Extention را اعمال کن
5. **Abstract Method:**
   توابع بدون بدنه که فقط در کلاس abstract و interface ها میتوان تعریف نمود
6. **Partial Method:**
   این توابع فقط درون کلاس Partial تعریف میشوند

جدول زیر خلاصه ی این است که کدام کلاس ها چه نوع توابعی را میتوانند داشته باشند:

	Class Type					Interface
انواع / ویژگی توابع کلاس	Regular	Abstract	Partial	Sealed	Static	
Extension Method	✗	✗	✗	✗	✓	✗
Static Method	✓	✓	✓	✓	✓	✗ 7.3
Instance Method(void)	✓	✓	✓	✓	✗	✗
Abstract Method	✗	✓	✗	✗	✗	✓
Partial Method	✗	✗	✓	✗	✗	✗
Virtual Method	✓	✓	✓	✗	✗	✗

-Examples:

Class Inheritance Example:

```
6 references
class Animal  //Base class (parent)
{
    6 references
    public virtual void AnimalSound() //Polymorphism Method
    {
        Console.WriteLine("The animal makes a sound");
    }
}
6 references
class Dog : Animal //Class Declaration
{
    private string name; //Field Declaration (Attribute)
    3 references
    public string Name1 //Property Declaration
    {
        get { return name; }
        set { name = value; }
    }
    2 references
    public string Name2 //Automatic Property Declaration (Short Hand)
    { get; set; }
    5 references
    public override void AnimalSound() //override the base class polymorphism method
    {
        Console.WriteLine($"The {name ?? Name1 ?? Name2 ?? "[unnamed dog]"} says: huff-huff!");
    }
    1 reference
    public void Bark() //Method Declaration (non-static)
    {
        Console.WriteLine($"{name ?? Name1 ?? Name2 ?? "[unnamed dog]"} said: Wow-wow!");
    }
    2 references
    public Dog() //Constructor Declaration (peremeterless empty constructor)
    {
    }
    0 references
    public Dog(string name) //Another Constructor Declaration
    {
        this.name = name;
    }
    0 references
    ~Dog() //Destructor Declaration
    {
        Console.WriteLine("END!");
    }
}
1 reference
class Cow : Animal  //Derived class (child)
{
    4 references
    public override void AnimalSound() //override the base class polymorphism method
    {
        Console.WriteLine("The cow says: Moooow!");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Animal myAnimal = new Animal(); //Create a Animal object
        Animal myCow = new Cow();       //Create a Cow object
        Animal myDog1 = new Dog();      //Create a child Dog object
        Dog myDog2 = new Dog();         //Create a Dog object

        myAnimal.AnimalSound();
        myCow.AnimalSound();
        myDog1.AnimalSound();           //It doesn't let to have access to the Dog's Method & Property!
        myDog2.Name1 = "Stewie";
        myDog2.AnimalSound();
        myDog2.Bark();
    }
}
```

```
C:\WINDOWS\system3:
The animal makes a sound
The cow says: Moooow!
The [unnamed dog] says: huff-huff!
The Stewie says: huff-huff!
Stewie said: Wow-wow!
END!
END!
Press any key to continue . . .
```

Abstract Class Example:

```
1 reference
abstract class Animal //Abstract class
{
    private string name;
    3 references
    public string Name //Property Declaration
    {
        get { return name; }
        set { name = value; }
    }
    3 references
    public abstract void AnimalSound(); //Abstract method (does not have a body)
    1 reference
    public void Sleep() //Regular method
    {
        Console.WriteLine("Zzz");
    }
}
2 references
class Dog : Animal //Derived class (inherit from Animal)
{
    3 references
    public override void AnimalSound()
    {
        //The body of AnimalSound() is provided here
        Console.WriteLine($"{Name ?? "[unnamed dog]"} said: Wow-wow!");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Dog myDog = new Dog(); //Create a Dog object
        myDog.AnimalSound();  //Call the abstract method
        myDog.Name = "Brian";
        Console.WriteLine($"Dog name is {myDog.Name} now!");
        myDog.AnimalSound();
        myDog.Sleep();  //Call the regular method
    }
}
```

```
C:\WINDOWS\system3:
[unnamed dog] said: Wow-wow!
Dog name is Brian now!
Brian said: Wow-wow!
Zzz
Press any key to continue . . .
```

Interface Example:

```
1 reference
interface IAnimal //interface
{
    3 references
    string Name
    { get; set; }
    2 references
    void AnimalSound(); //interface method (does not have a body)
}
2 references
class Chick : IAnimal //Chick "implements" the IAnimal interface
{
    3 references
    public string Name
    { get; set; }
    2 references
    public void AnimalSound()
    {
        // The body of AnimalSound() is provided here
        Console.WriteLine($"The {Name ?? "chick"} says: gik gik!");
    }
}
0 references
class Program
{
    0 references
    static void Main()
    {
        Chick chick = new Chick();
        chick.Name = "Ashkan";
        chick.AnimalSound();
    }
}
```

```
C:\WINDOWS\system3:
The Ashkan says: gik gik!
Press any key to continue . . .
```