

#make_sense

OOP

Object oriented programming

With c#

Created by: Ahmed fares

ما هو مفهوم الـ OOP ؟

هي عبارة عن نمط برمجة متقدم، وفيه يقسم البرنامج إلى وحدات تسمى الكائنات(**Objects**) كل كائن عبارة عن حزمة (تعليق) من البيانات (المتغيرات والثوابت) والدوال ووحدات التنظيم وواجهات الاستخدام. ويتم بناء البرنامج بواسطة استخدام الكائنات وربطها بعضها البعض وواجهة البرنامج الخارجية باستخدام هيكلية البرنامج وواجهات الاستخدام الخاصة بكل كائن.

لماذا نستخدم OOP في عملية بناء البرامج ؟

- الـ OOP تعد من اهم المفاهيم البرمجيه التي نستخدمها عند بناء البرامج وذلك لانها
1. تحافظ على عدم تكرار الأكواد وسهولة التعامل معها
 2. تقسيم البرنامج الى وحدات منفصله يمكن العمل و التعديل المستمر عليها دون التأثير المباشر على عمل الوحدات الاخرى
 3. امكانية عمل انواع جديدة من الـ Data Types تناسب احتياجات البرنامج الذي يتم بناءه

مفاهيم اساسية داخل الـ OOP

Class .1

Object .2

Class relationships .3

Encapsulation .4

Polymorphism .5

Etc.. .6

ما هو الـ Class ؟

Class is a template definition of the method and variable

أي أن الكلاس هو نموذج من المتغيرات والدوال التي نجمعها عن **كيان** نريد تمثيله في البرنامج الخاص بنا ويستخدم لتنظيم للكود

يتكون **الكلاس** من طبقتين :

1. طبقة الداتا وهي التي تحتوى على العناصر و المتغيرات الخاصة بـ **Class**
2. طبقة الاوامر المنطقية وهي التي تحتوى على الدوال التي تستخدم الطبقة الاولى في تنفيذ عملها



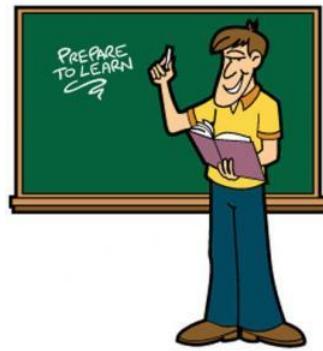
مثال :

إذا أردنا بناء برنامج للمدرسه ... المدرسه تحتوى على عدد العناصر على سبيل **المثال**

1. طلاب
2. مدرسيين
3. موظفين
4. مواد دراسية

لناخذ **المدرس** ك بداية ونقوم ببناء موذج خاص بهم بحتوى على كل الصفات والافعال المشتركة التى تجمع كل مدرس ، فكل مدرس له أسم و عنوان و مواد يدرسها و فصل و الخ و ايضا المدرس يقوم بالشرح و يأخذ الغياب يصحح الامتحانات الخ

لنتحول الكلام السابق عن المدرس الى **class** ، كما قلنا ان الـ **class** يتكون من طبقتين
الطبقه الاولى طبقة الـ **Data** والتى تحتوى على الصفات والخصائص



Name

Address

Subject

Class

Salary

Data layer ←

Teach ()
Correct ()
Check_absent ()

→ **Logic layer**

Let's code

```
0 references
class Teacher
{
    //Data layer
    string name;
    string address;
    string subject;
    string Class;
    int salary;

    //logic layer
    0 references
    void tech()...;

    0 references
    void correct()...;

    0 references
    void check_absent()...;

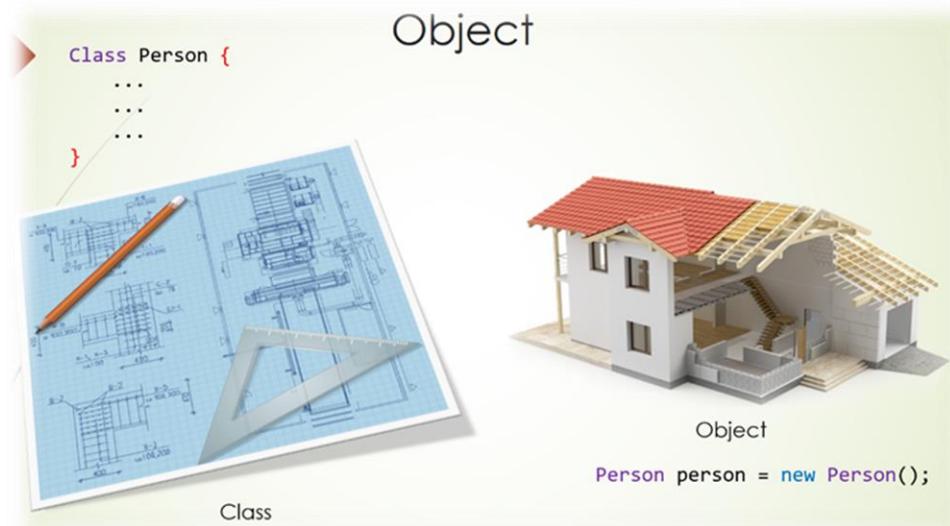
}

}
void check_absent()...
```

في الصوره السابقة تحويل التحليل السابق للـ **C#** الى كود بلغة الـ **class** .
نقوم بتعريف الـ **class** و وضع اسم له وفي هذه الحاله هو **class** خاص بالمدرسین لذاك كمنا بتسمیته **teacher** .
ثم بداخله نعرف المتغيرات التي ستحمل صفات المدرس مثل الاسم و العنوان وتحديد الـ **Data type** الخاص بكل متغير
بعد ذالك ننتقل الى الـ **logic layer** وهي الدوال و الافعال التي يقوم بها كل مدرس مثل الشرح والتصحيح

ما هو ال Object ؟

كما ذكرنا ان ال class هو نموذج لكيان يحتوى على صفات و افعال هذه الكيان اي انه النموذج التخطيطى للكيان اما ال Object هو التنفيذ الحقيقى لهذا النموذج ، ويمكن بناء اكثرا من Object من نفس هذه النموذج



في المثال السابق قمنا ببناء class تحت اسم teacher

```
0 references
class Teacher
{
```

ف class teacher هو مجرد نموذج فقط ولتنفيذ هذا المخطط وعمل نسخه منه تحمل نفس محتوى ال

```

0 references
static void Main(string[] args)
{
    Teacher teacher_one = new Teacher();
}

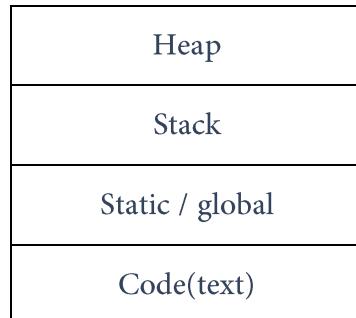
```

داخل الـ **Teacher teacher_one = new Teacher()** قمنا بكتابه **main function**

وتنقسم هذه الجمله الى شقين :

- **Teacher teacher_one**
- **new Teacher()**

لفهم الجزئين يجب ان نفهم اكثر ما يتم خلف الاكواود وما يحدث في الذاكره



memory

عند تنفيذ الكود يكون النظام بتخصيص مساحه من الذاكره خاصه بالبرناموج وتنقسم الى عدد من انواع الذاكره المختلفه

- **Code**
- وهي الذاكره التي يتم فيها تخزين الأكواود نفسها ك **Text**
- **Stack**
- **Heap**
- نوعين مختلفين من الذاكره ولكل واحده منهم استخدام مختلف عن الثاني

نعلم ان كل قيمة نقوم بإدخالها تخزن في الذاكرة، عندما نعرف متغير من النوع int يقوم البرنامج بحجز مكان في الذاكرة مساحته 4 byte وهي المساحة الخاصة به int وكل مكان في الذاكرة address يشير اليه حتى يمكننا الرجوع اليه مره اخره واستخدام القيمة بداخله

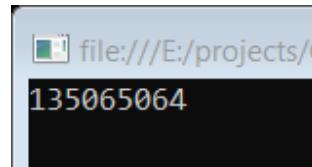
```
using System;
static void Main(string[] args)
{
    unsafe
    {
        int MyNumber = 10;

        int* ptr = &MyNumber;

        Console.WriteLine( (int)ptr );

        Console.ReadKey();
    }
}
```

في الصوره السابقه عرفنا متغير من نوع int بعد ذالك عرفنا متغير من نوع int pointer وهو نوع خاص قادر على حمل عنوانين المتغيرات من نوع int و تستخدمنا * بجانب اسم الـ data type حتى نفرق بينها وبين الـ data type العادي ثم وضعنا بها عنوان المتغير MyNumber في الذاكرة وذلك عن طريق استخدام & قبل اسم المتغير وطبع هذا العنوان



و يعد الـ int من نوع الـ primitive data type وهو نوع بسيط غير قادر الا على حمل قيمة واحدة ويخزن داخل الـ stack

ما هو الـ Stack ؟

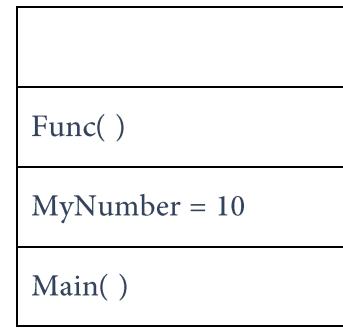
كما قلنا انه نوع من انواع الذاكرة و تختص في تخزين كل ما يتم تنفيذه ويتم تنظيم العمل فيها بشكل تلقائي ولذاك تعمل بشكل اكفاء واسرع

```

0 references
static void Main(string[] args)    // first thing goes to stack
{
    int MyNumber = 10; // second this goes to stack
    func(); // third
    func2(); //fourth

    Console.ReadKey();
}

```



اول شيء يدخل ال stack هى دالة ال **main** وتبقى بداخل ال stack لاننا ما زلنا بداخلها بعد ذالك المتغير ال **MyNumber**

بعد ذالك تدخل دالة ال **func** حتى يتنهى العمل بداخلها فتخرج من ال stack وتدخل دالة **func2** حتى تنتهي

وتخرج من ال stack وبهذا تكون قد انتهت دالة ال **main** فيخرج المتغير ثم تخرج ال **main**

وتعمل ال Stack ب نظام ال **FIFO** او **First in First out** اول شئ دخل ال stack كان ال **main** و هو اخر شئ خرج منه

و في حالة المتغير السابق **MyNumber** يتم تخزين قيمته في ال stack مع العنوان الخاص و يسمى هذا النوع ب ال **value type**

اى انه يحمل قيمته معه

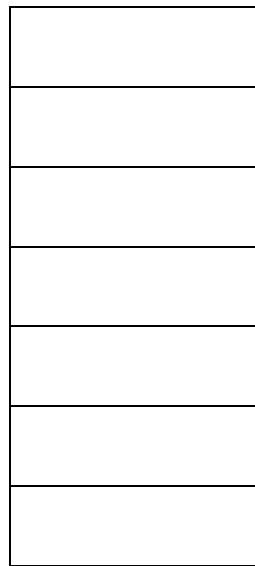
لكن ماذا لو كنا نعرف **class** بداخل ال **class** العديد من المتغيرات و الدوال؟ وفي هذه الحالة يتم فصل القيمه عن العنوان

وهنا تخزن القيمه في ال **heap**

ما هو ال **Heap** ؟

هو نوع اخر من الذاكرة لكنه اكبر من ال **Stack** غير منظم و غير مرتبه و العمل فيها يمكن ان يتم بشكل يدوى ولذالك هى ابطأ في التعامل

مثال



stack



Heap

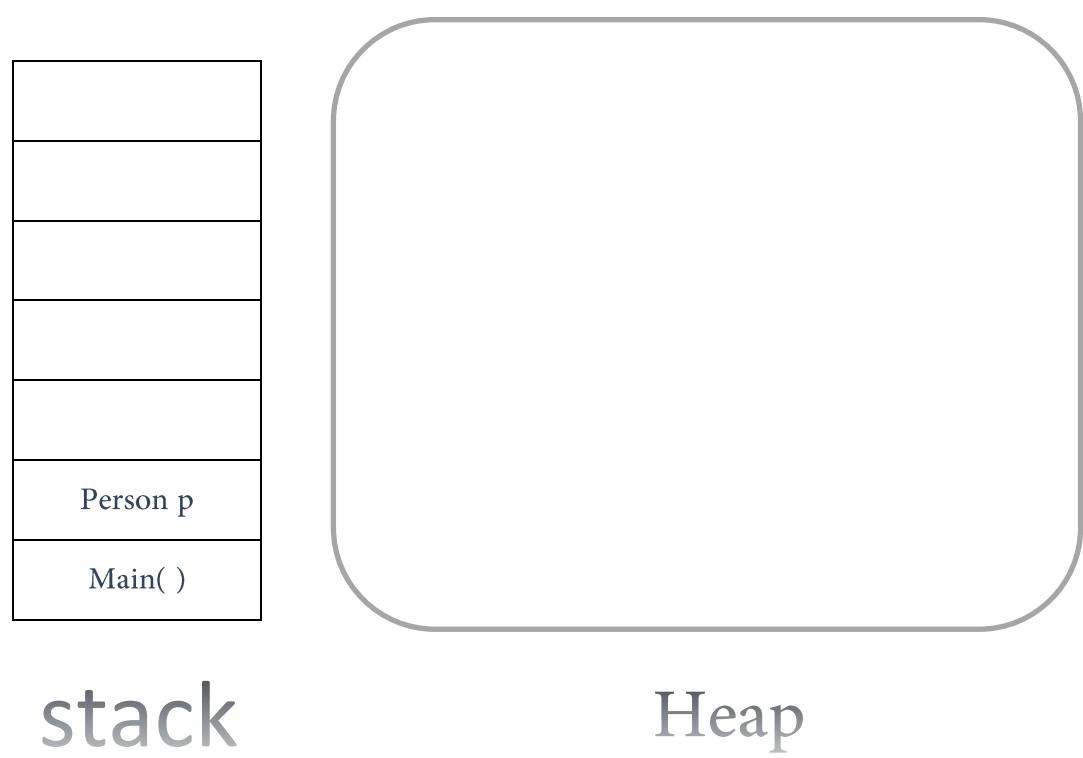
لدينا **heap** و **stack** فارغين

```
references
class person
{
    public string name;
    public int age;
    0 references
    public person(string name , int age)
    {
        this.name = name;
        this.age = age;
    }
}
0 references
class Program
{
    0 references
    static void Main(string[] args)    // first thing goes to stack
    {

        person p;

        Console.ReadKey();
    }
}
```

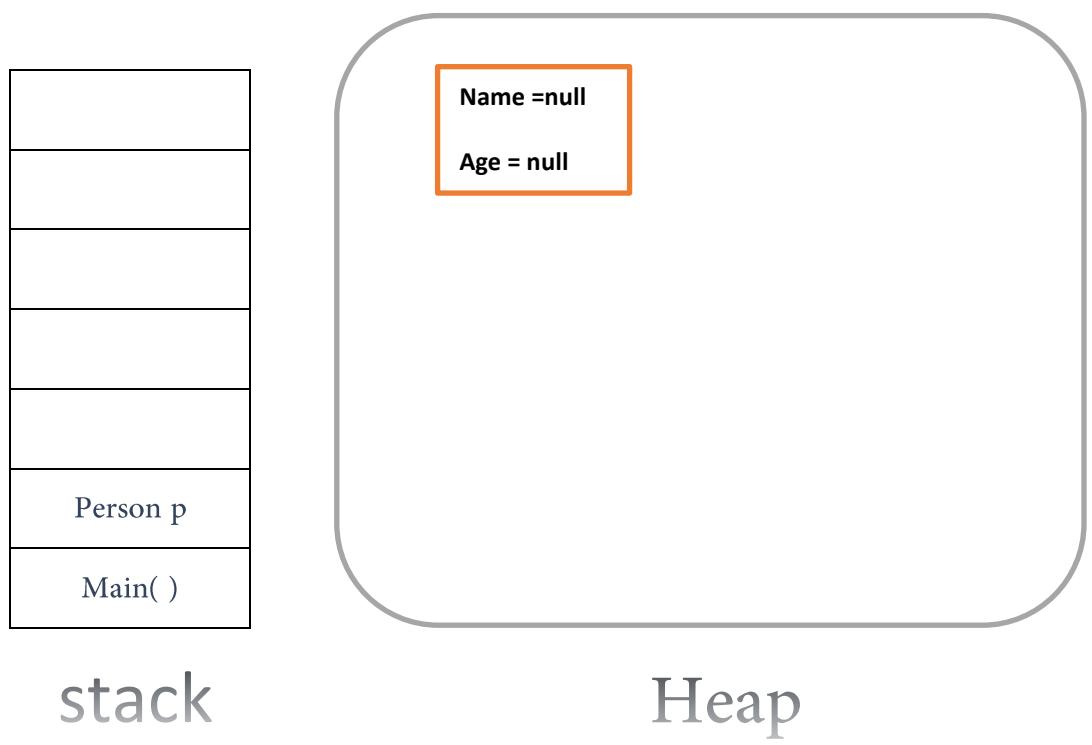
لدينا Class اسمه person وبه متغيرين الاسم وال عمر ولدينا الـ main الاساسى الذى يحتوى على دالة الـ



تدخل دالة الـ main او لا في الـ stack بعد ذلك عرفنا object من نوع class person و يعتبر الـ person من نوع composite data type لأنة يتكون من مجموعه من الـ primitive and composite data type لذاك فإنه يحمل أكثر من قيمة ولا تخزن هذه القيم في الـ stack لكن في الـ Heap ، لذاك يجب تخصيص مساحه داخل الـ Heap لحمل هذه القيم

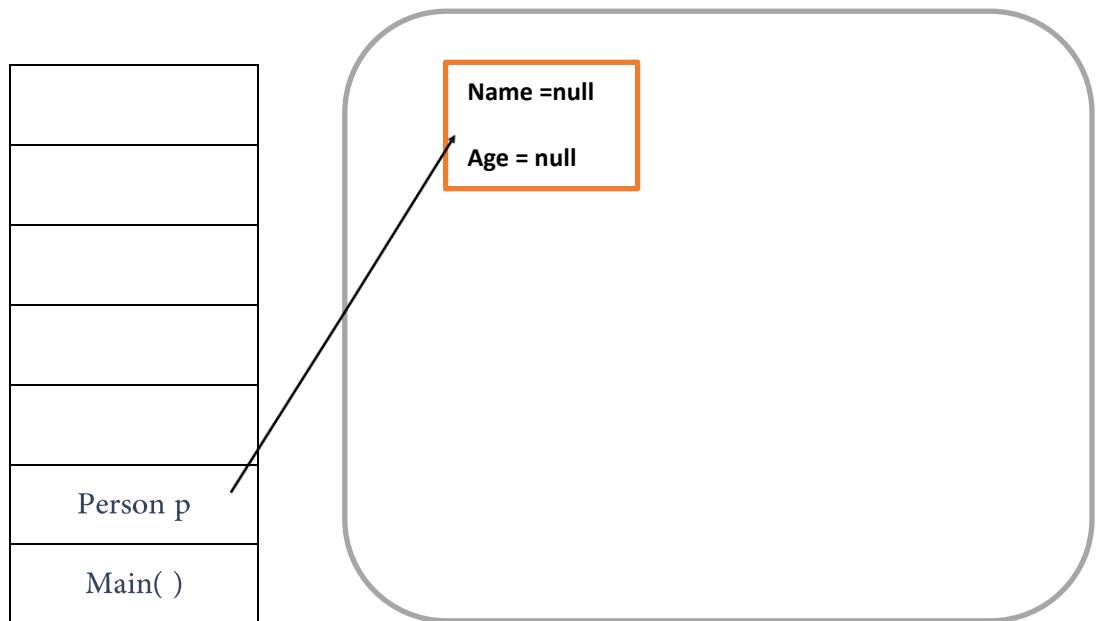
```
person p;
new person();
Console.ReadKey();
```

نستخدم الكلمة المفاتيحية `new` لخضيص مساحه جديد تحمل القيم الخاصه بـ `person`



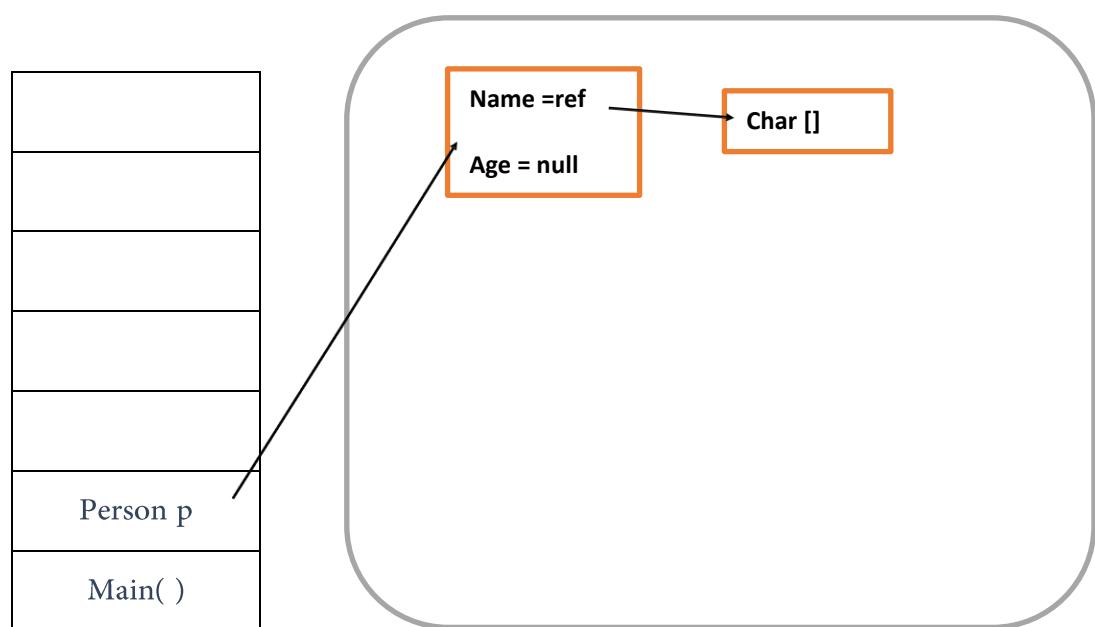
الآن نملك المتغير و عنوانه في الـ `stack` و نملك مساحه قادره على حمل قيم للـ `object person` ولكن يجب ربط الـ `object` بامساحه التي يجب ان يخزن بها المتغيرات الخاصه به

`Person p = new person()`



لأن أصبح لدى الـ **object** عنوان الذاكرة التي يخزن بها القيم والتي تقع في الـ **Heap** ويسمى هذا بالـ **reference** ويسمى هذا النوع من الـ **objects** بالـ **reference type**

في الحاله السابقه الـ **string** يملك بداخله ايضا **composite type** و هي المتغير **name** لانه من النوع **char** وال **array** هو **string** من الـ **primitive data type** وهو ال **char** اي ان المتغير **name** يعمل ملكان اخر في الذاكرة

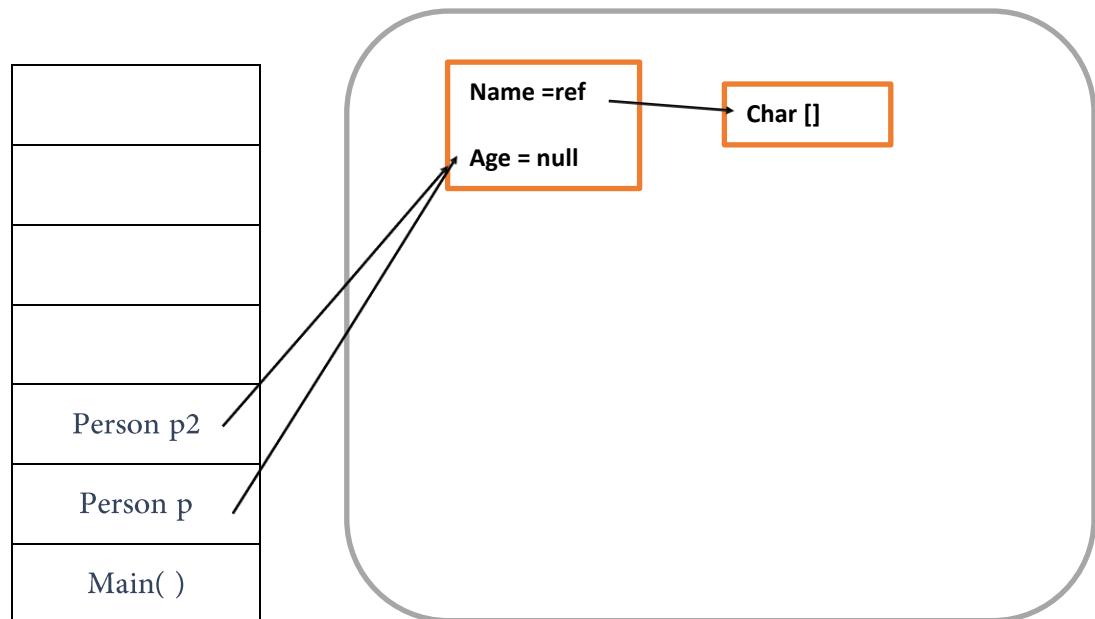


stack

Heap

في المتغير الاول اشرنا الى مساحه جديد **new** في الذاكرة و لكن يمكن ايضا ان نشير الى مساحه موجوده من قبل

```
person p = new person();
person p2 = p;
```



stack

Heap

ف أصبح كلاهما يشيران إلى نفس المساحة من الذاكرة وأى تأثير يجد على أى object يراه الـ الآخر

```

person p = new person();
p.name = "ahmed";
person p2 = p;
Console.WriteLine("p2 name is : " + p2.name);
p2.name = "mohamed";
Console.WriteLine("p name after change : " + p.name);
Console.ReadKey();

```

```
file:///E:/projects/C#/Desktop/Console/Csh
p2 name is : ahmed
p name after change : mohamed
```

ما هو ال parameters ؟

هي نوع المتغيرات يتم تمرير الداتا من خلالها الى الدوال .

مثال :

```
int a = Math.Max(1, 2);
Console.WriteLine(a);
```

في المثال السابق استخدمنا **Math** و هو **class** خاص بالدوال الرياضيه و استخدمنا دالة الـ **max** التي تقارن بين رقمين و تقوم بإرجاع أكبرهم ، أرسلنا اليها رقمين 1 , 2 وفي هذه الحاله تمرر الى الداله عن طريق الـ **Parameters**

```
public static int Max(int val1, int val2);
```

في الصوره السابقه دالة الـ **Max** نفسها في داخل كلاس الـ **math** الداله بداخلها متغيرين **int val1** و **int val2** وعن طريق هذا المتغيرات تقبل الداله تمرير اي داتا من نفس نوع المتغيرات وهو **int**

ما هو ال constructor ؟

هو نوع من خاص الدوال مرتبط بالـ **class** و يستخدم الـ **constructor** في القيم الابتدائيه للمتغيرات او القيم الافتراضيه وذاك لأن الـ **constructor** هو اول ما يتم استدعاءه وتنفيذه من الـ **Class** و يحمل الـ **constructor** نفس اسم الـ **Class** و لا يحمل تعريفه اي نوع من انواع الـ **data type** و بالتالي لا يكون للداله **return**

```
3 references
class car
{
    1 reference
    public car()
    {
        Console.WriteLine("from class constructor ");
    }
}
```

في الصوره السابقة نملك `class` `car` وبداخله أمر طباعة

```
car MyCar = new car();
```

ثم بداخل الـ `main function` قمنا بعمل `object` من الـ `class` `car` وعند تنفيذ الكود

```
file:///E:/projects/C#/Desktop/Console  
from class constructor
```

أي أنه بمجرد عمل `object` من الـ `class` أخذ الـ `object` نسخة من محتوى الـ `constructor` بما فيها الـ `Object` وعند تنفيذ الكود يكون الـ `constructor` هو اول ما يتم استدعاؤه في الـ `Object`

ويستخدم الـ `constructor` في وضع القيم الافتراضيه و الابتدائيه للمتغيرات كما في الصوره التاليه

```
3 references  
class car  
{  
    1 reference  
    public car(string name = "none", int speed = 0)  
    {  
        this.name = name;  
        this.speed = speed;  
    }  
  
    public string name;  
    public int speed;  
}
```

لدينا متغيرين هما `name`, `speed` داخل الـ Class ، لوضع القيم ابتدائيه لهذه المتغيرات نستخدم الـ `constructor` وذلك لأنه اول ما يتم تنفيذه من اكواز الـ `class`

وكما تعلمونا كيفية تمرير الداتا الى الدوال عن طريق الـ `parameters`، بداخل اقواس الـ `construer` قمنا بتعريف متغيرين من نفس نوع الداتا المراد تمريرها

المتغير الاول `string name = "none"` ، أى ان الدالله ستستقبل داتا من نوع `string` في متغير اسمه `name` ولكن قمنا باعطائه قيمة افتراضيه، بمعنى انه اذا لم يتم تمرير داتا ستكون قيمة الـ `name` هي `none` ، اما اذا تم تمرير داتا ستكون `name` هي الداتا التي تم ولكن اذا لاحظت هناك متغيرين يحملان نفس الاسم !

وهنا نتحدث عن مفهوم الـ `scoop`

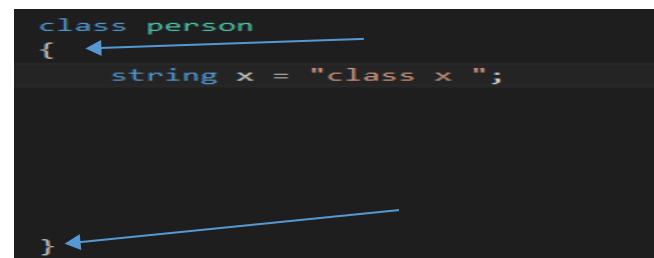
ما هو الـ `scope`؟

تعنى الكلمة `scope` النطاق ، ويقصد بها فالبرمجه النطاق الأولويه في الاستدعاء



```
0 references
class person
{
```

في الصوره السابقه `class` يسمى `person` و هو فارغ تماماً



```
class person
{
    string x = "class x ";
}
```

قمنا بتعريف متغير جديد اسمه `x` من نوع `string` ويمكن استخدام هذا المتغير في نطاق هذه الاقواس { }

```
2 references
class person
{// the beginning of the "class x" scope
    string x = "class x ";

    1 reference
public void func()
{
    string x = "function x";
}
```

بعد ذلك قمنا بتعريف دالة تحت اسم `func` وقمنا بتعريف متغير جديد تحت اسم `x` ايضا الفرق بين الاثنين هو النطاق ف الاولى كما قلنا انه مرأ في نطاق اقواس الـ `class` ويمكن رؤيته حتى داخل الدوال لكن الـ `x` الثانية نطاقها داخل اقواس الدالة فقط ولا يمكن استدعاءها من خارج هذه الدالة

ماذا اذا فنا بـ الاستدعاء المتغير من داخل الـ `function` ؟

في هذه الحاله سيكون هناك خيارين، المتغير داخل الدالة والمتغير داخل الكلاس، لكن ستكون الاولوية في الاستدعاء للمتغير الداخلى ويسمى الـ `local variable` أي انه سيستخدم المتغير الذى يحمل قيمة `x`

ماذا اذا فنا بـ الاستدعاء المتغير من داخل الـ `class` ؟

هنا سيكون هناك خيار واحد فقط وهو المتغير الخاص بالكلاس لأن المتغير الخاص بالدالة غير مرئي خارج اقواس الدالة

```

2 references
class person
{ // the begining of the "class x" scope
    string x = "class x ";
    1 reference
    public void func()
    {
        string x = "function x";
        Console.WriteLine(x);
    }

    1 reference
    public void print()
    {
        Console.WriteLine(x);
    }
} // the end of "class x" scope

```

في الصوره نقوم بطباعة المتغير `x` في حالات مختلفه الاولى داخل داله `func` الثانيه داخل داله اخرى تسمى `print`
 في الحاله الاولى يرى البرنامج المتغيرين لكن الاولوية لمتغير الداله `func`
 الحاله الثانية يرى البرنامج متغير واحد فقط و هو متغير الخاص بالـ `class`

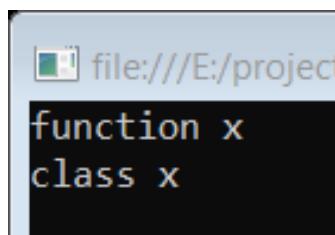
```

0 references
static void Main(string[] args) // first thing goes to stack
{
    person p = new person();
    p.func();
    p.print();

    Console.ReadKey();
}

```

و بعد ذالك صنعنا `object` من الـ `class` واستدعينا دالتي الـ `func` و `print`



لكن ماذا إذا أردنا أن نستخدم المتغير في نفس الوقت داخل دالة الـ `func` ؟

في هذه الحالة نستدعي المتغير الخاص بالـ `class` نفسه حتى يستطيع أن يفهم أننا نريد المتغير الخارجي وليس الداخلي

```
3 references
class person
{
    // the begining of the "class x" scope
    static string x = "class x";
    1 reference
    public void func(string x)
    {
        person.x = x;
        Console.WriteLine(x);
    }
}
```

الـ `parameter` هو المتغير الداخلي الخاص بالدالة و قمنا بستدعاء المتغير الخارجي الخاص بالـ `class` عن طريق اسم الـ

لندع إلى المثال الأساسي وهو وضع قيم الـ متغيرات عن طريق الـ `constructor`

```
3 references
class car
{
    1 reference
    public car(string name = "none", int speed = 0)
    {
        this.name = name;
        this.speed = speed;
    }

    public string name;
    public int speed;
}
```

نفس الحال ولكن المختلف كلمة `this` !

ما هي كلمة `this` ؟

هي كلمة مفاتيحية بسيطة تستخدم للتعبير عن الـ `Object` الحالى ، كما عرفنا انه اذا أردنا استدعاء المتغير الخارجي نقوم بإستدعاوه عن طريق اسم الـ `class` ، لكن في هذه الحالة نحن لا نتعامل مع الـ `class` ولكن مع الـ `Objects` المنسوخة من الـ `Class` أي انه يجب عند عمل `object` جديد أن نستدعي المتغير بنفس الاسم ، وهنا تكمن قوه `this` انها توفر عليك عمل كل هذا بكتابه بدل اسم المتغير وعند التنفيذ سيتم التبديل التلقائى لكلمة `this` باسم المتغير

نستخدم كلمة static سواء قبل أو بعد المتغيرات او الدوال !

ما معنى كلمة Static ؟

ترجمة كلمة Static هي الثابت وفي البرمجة تعد من اهم الكلمات المفتاحية والغرض الاساسى منها الحفاظ على الذاكرة و كفاءة البرنامج

مثال

```
1 reference
class Worker
{
    public string company_name = "HP";
    public string name;
    public int salary;
0 references
public Worker(string name , int salary)
{
    this.name = name;
    this.salary = salary;
}
}
```

في هذه المثال لدينا ثلاثة متغيرات اسم الشركه واسم العامل و المرتب ، اسم العامل و المرتب يختلف من عامل الى اخر ولكن اسم الشركه هو ثابت على فرض انهم يعملوا في نفس الشركه ، وفي هذه الحاله اذا كان لدينا 1000 عامل سيكون لدينا 1000 متغير لهم نفس القيمه ، وهنا تظهر اهميه Static، عند اضافة Static الى تعريف المتغيرسيتم إنشاء متغير واحد فقط .

وبناءً على ذلك لا يمكن استدعاء هذا المتغير من الـ Object لذاك يتم استدعاء عن طريق اسم الـ Class

```

  REFERENCES
class Worker
{
    public static string company_name = "HP";
    public static int NumOFWorkers = 0;
    public string name;
    public int salary;
    3 references
    public Worker(string name , int salary)
    {
        this.name = name;
        this.salary = salary;
        NumOFWorkers++;
    }
}

```

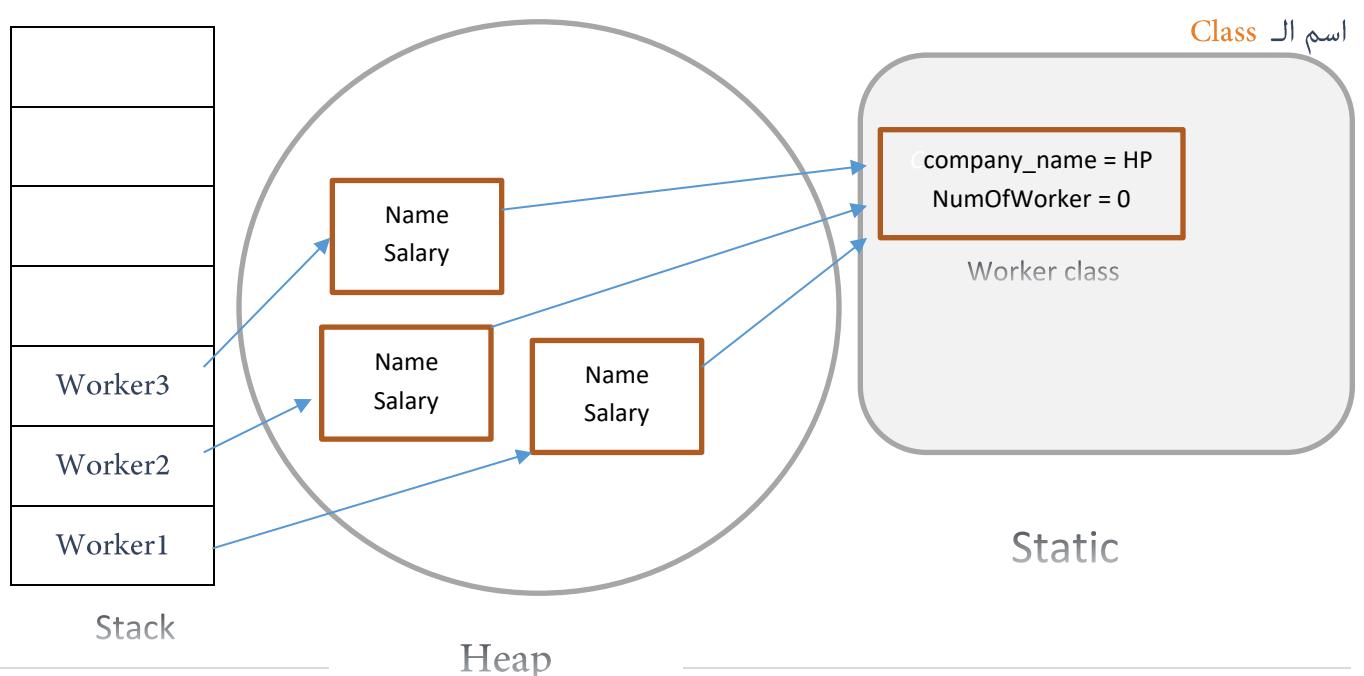
في الصوره السابقة نملك متغيرين من النوع Static اسم الشركه وعدد العاملين و في الـ Constructor قمنا بعمل increment للمتغير الخاص بعدد العمال

```

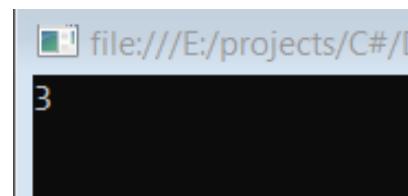
Worker worker1 = new Worker("ahmed", 0);
Worker worker2 = new Worker("islam", 0);
Worker worker3 = new Worker("mohamed", 0);
Console.WriteLine(Worker.NumOFWorkers);

```

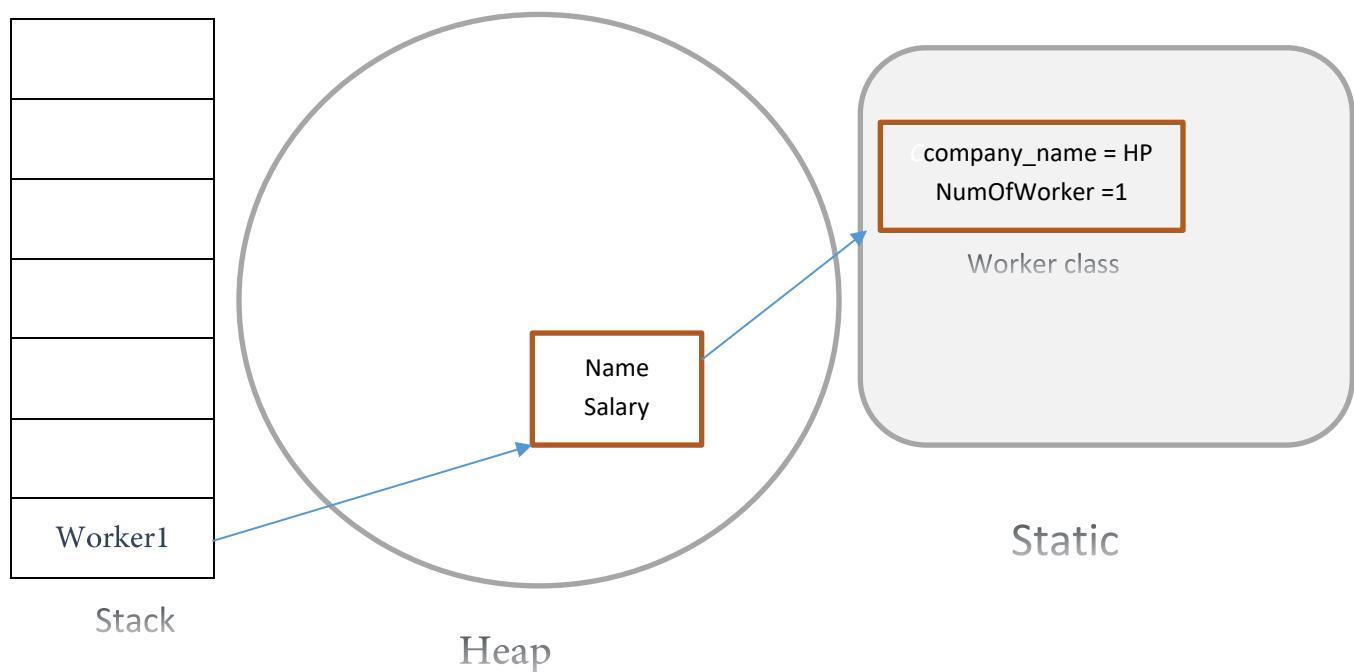
و نسخنا ثلاث Objects من الـ Class أي انه لدينا ثلاثة عمال و قمنا بطباعة المتغير الخاص بعدد العمال بعد استدعاؤه عن طريق



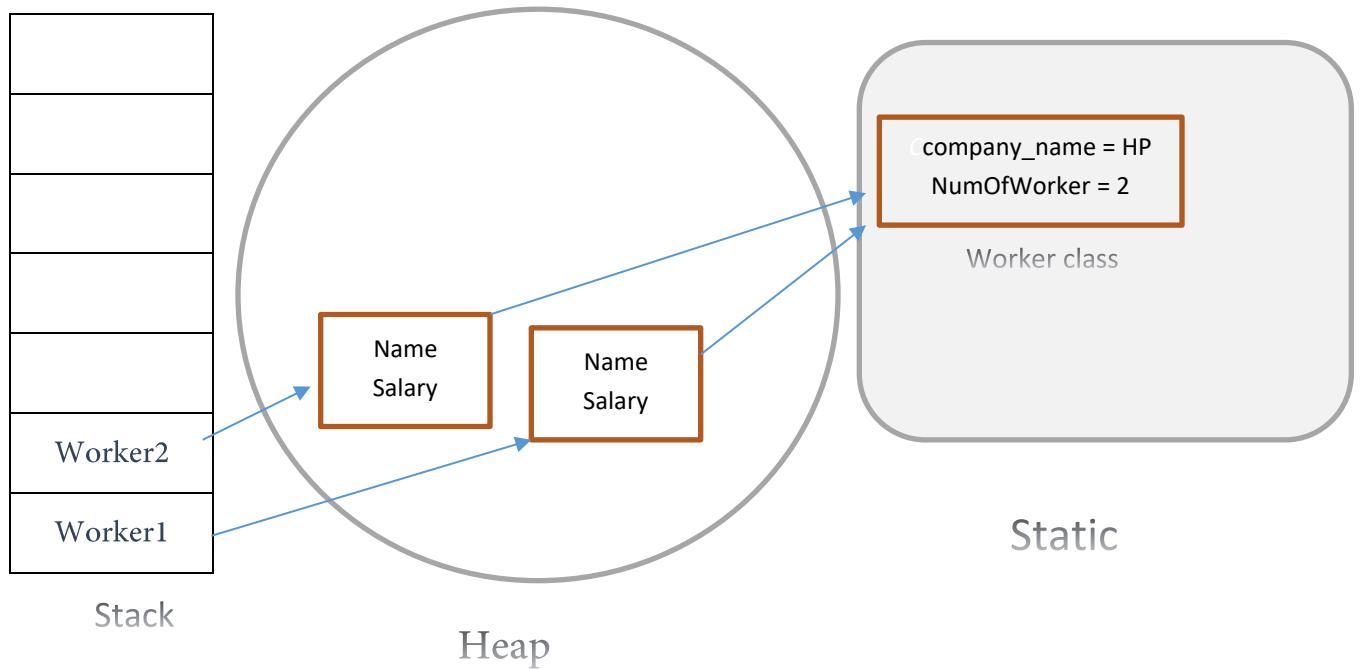
في الرسمه السابقة تعبير عن الثلث **objects** و كيف يتم حفظهم في الذاكره كما شرحنا في السابق ولكن في هذا المثال لدينا متغيرين من النوع **static** وتخزن المتغيرات من نوع **static** في نوع ذاكره خاص بها
ونلاحظ انه تم صنع متغيرات جديدة لكل **object** من المتغيرات العاديه مثل **salary** و **name** ، اما المتغيرين من نوع **static**
تم صنع متغير واحد فقط للكل، أو بمعنى ادق انه تم عمل متغير خاص بالكلas ككل وأى **object** من هذا **class** قادر على رؤية
هذا المتغير والتأثير عليه و لكن لا يمكن استدعاء هذه المتغيرات إلا عن طريق الـ **class** نفسه ونلاحظ ايضا اننا لا نستخدم **this** مع
المتغيرات من نوع **static** لانه لا يتم استدعائها الى عن طريق الـ **class** فقط



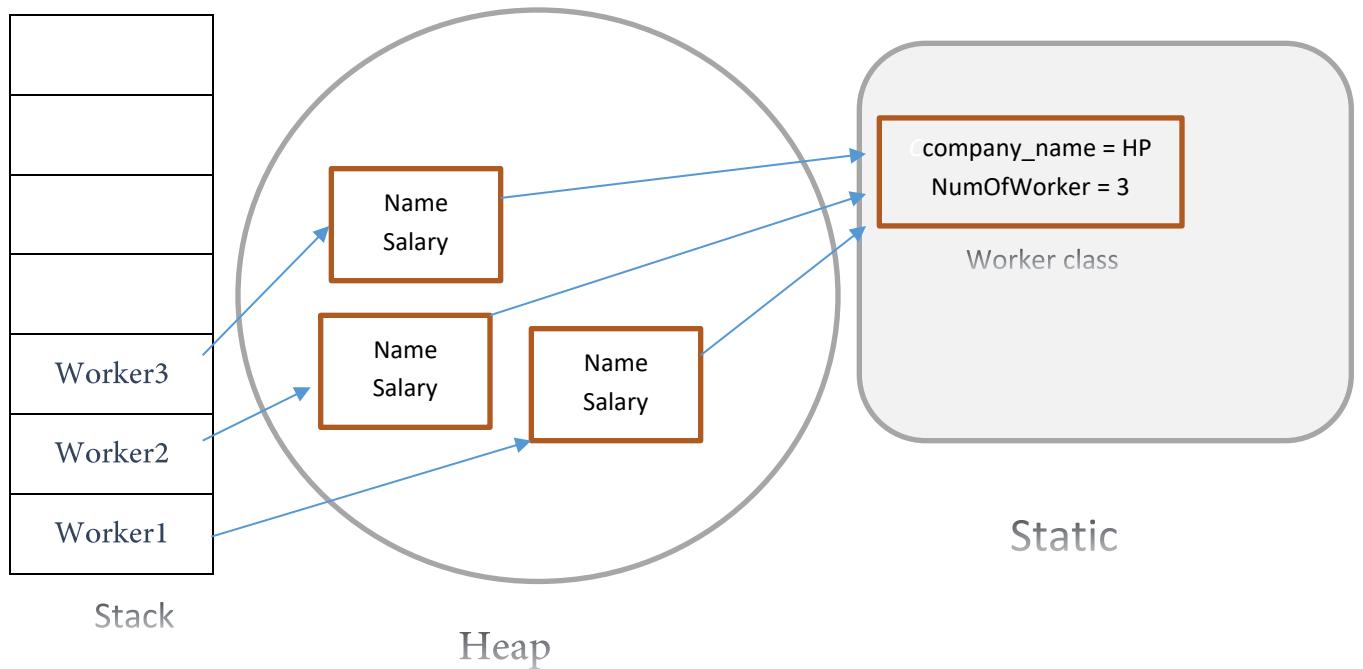
و عند تنفيذ البرنامج نجد انه طبع عدد العمال او بمعنى ادق عدد الـ **objects** التي قمنا بصنعها
لاننا اضفنا فالكلاس انه عند استدعاء الـ **constructor** يضيف واحد الى متغير عدد العمال وعند صنع **object** يتم استدعاء الـ **constructor**
ويزيد المتغير بمقدار واحد



ثم عند صنع ال object الثاني يقوم ال constructor بزيادة المتغير بمقدار واحد



وهكذا مع المتغير الثالث



تعرفنا على المتغيرات من نوع الـ static لكن هل من الممكن ان تكون الدوال من نوع static ؟

نعم ، فدالة الـ main من النوع static و الدوال من هذا النوع تشبه خصائص المتغيرات من نفس النوع حيث انه يتم عمل نسخه واحده من الدالة تكون تابعه للكلاس نفسه وليس للـ objects ولكن هناك خصائص اخرى تتميز بيهها دالة من نوع Static حيث أن دالة الـ static لا تتعامل مع متغيرات او دوال خارجيه الا اذا كانت من نوع static ايضا ولكن يمكن لـ اي دالة عاديه ان تستخدم الدوال من نوع static وذلك عن طريق استدعائها باسم الـ class

```
6 references
class Worker
{
    public static string company_name = "HP";
    public static int NumOfWorker = 0;

    public string name;
    public int salary;
    1 reference
    public Worker(string nameS , int salary)
    {
        name = nameS;
        this.salary = salary;
        NumOfWorker++;
        Worker.Func1();

    }
    1 reference
    public static void Func1()
    {

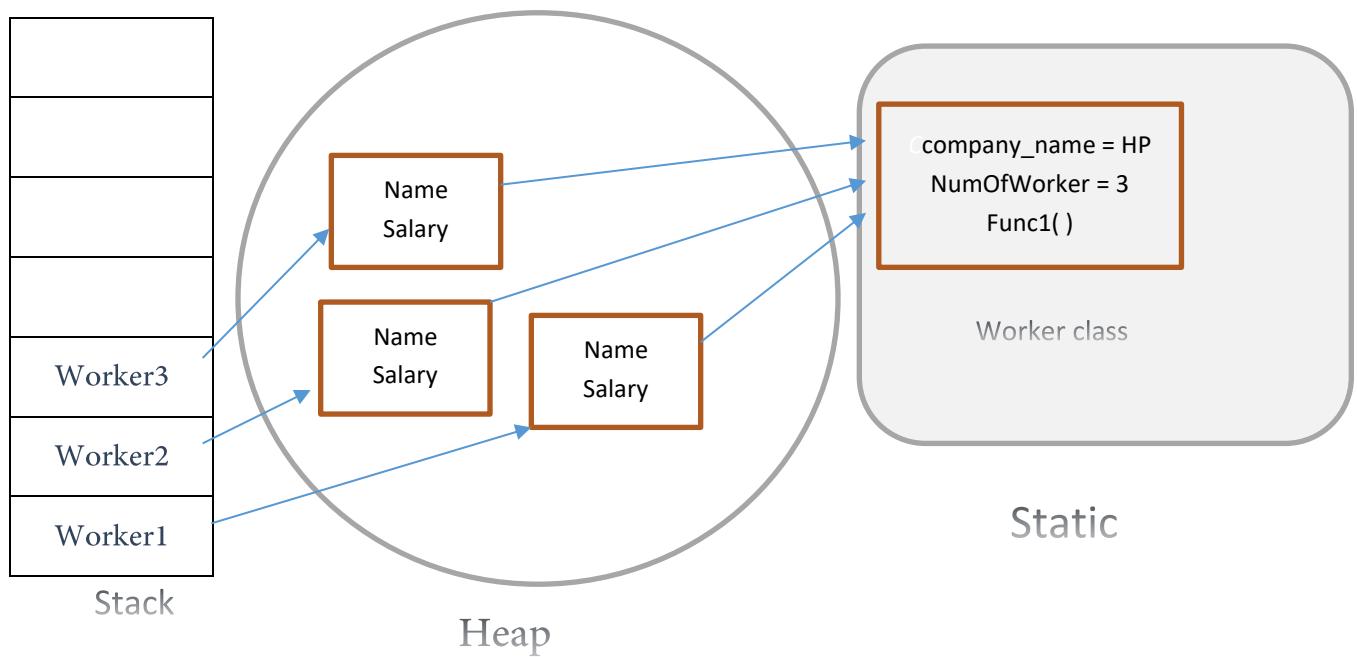
        Console.WriteLine("From static func1");
        Console.WriteLine(Worker.company_name);

    }
}
```

في الصوره السابقه لدينا دالة func1 وهي من النوع static و دالة الـ constructor وهي دالة ليست static

ونقوم دالة الـ constructor بإستدعاء دالة الـ static عن طريق اسم الـ class

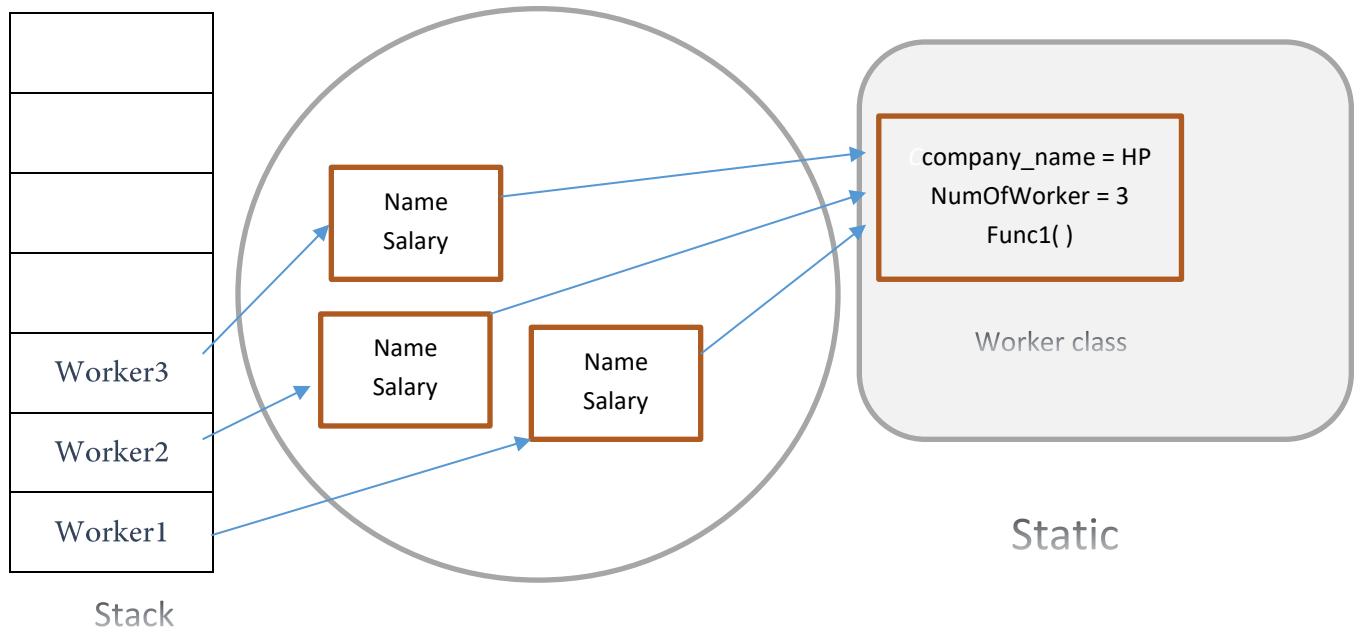
وايضا تقوم دالة الـ static باستخدام متغير من النوع static



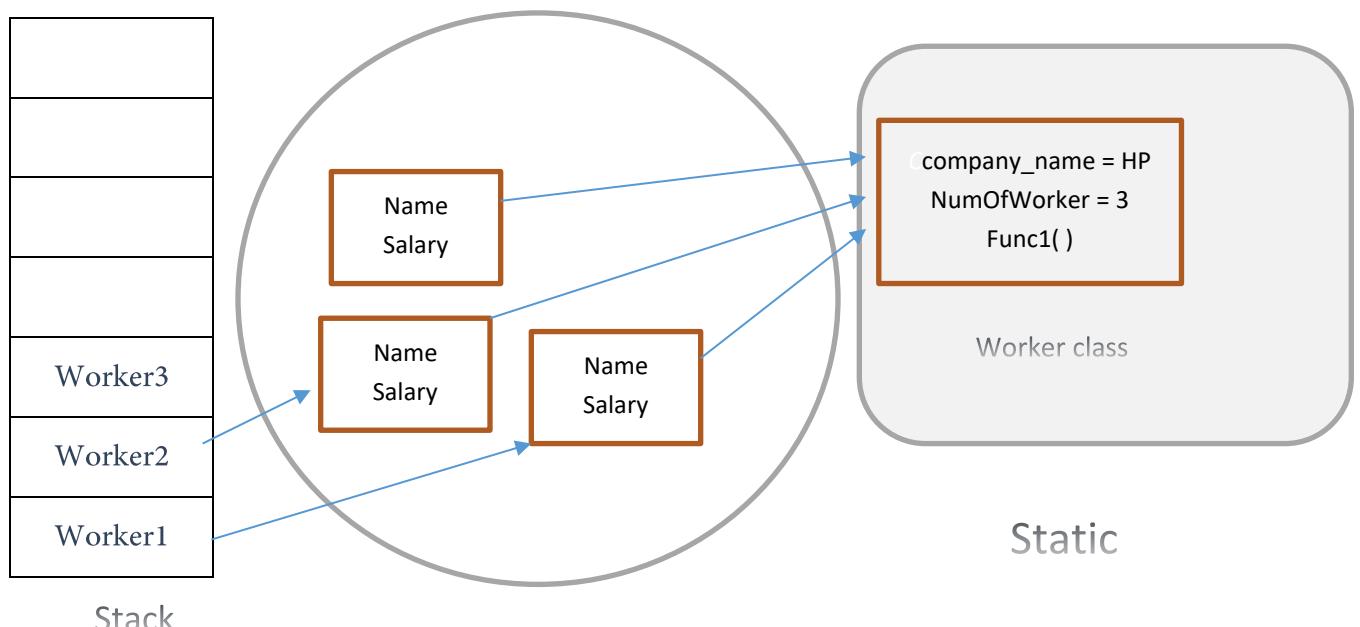
و ذلك يحدث لأنه كما قلنا ان العناصر من نوع `static` تكون منفردة ولا تكون تابعة للـ `objects` لكن للـ `class` ولا يتم استدعاءها الى عن طريق الـ `class` و تخزن في منطقة منفصلة عن باقى العناصر في الـ `class` اما العناصر العاديه لا يتم استدعاءها الا عن طريق الـ `object` ، فإذا تم وضع عنصر عادي داخل دالة من نوع `static` سيحدث اخطاء عند التنفيذ

بعد ان تعرفنا على ما يم داخل الـ `class` ننتقل الى بقى آخر وهو العلاقات بين الـ `classes` وتوجد اكير من طرقه لكي تتعامل الـ `classes` مع بعضها

قبل البدء في شرح العلاقات يجب توضيح بعض المفاهيم الاساسيه مثل `object life cycle` وهذا المفهوم يقصد به دورة حياة الكائن داخل الميموري ومتى يتم اخراجه من الميموري



يتم اخراج الـ **object** من الكائن في حالة فقدان المسار الخاصه به الذي يربط الـ **stack** و الـ **heap**
فـ اذا قمنا بقول ان ; **worker3 = null** في هذه الحاله ينقطع المسار الخاص بالـ **worker3** الذي يربطه مع الـ



ولكن ماذا يحدث للقيم الخاصه به داخل الـ **heap** في لغة الـ **C#** يتم ازالتها بشكل تلقائي عن طريق الـ **garbage collector**

وذلك للحفاظ على المساحة الخاصة بالـ heap لكن في بعض اللغات الأخرى يتم التعامل معها و إزالتها بشكل يدوي وتعد هذه العملية خطيرة و مهمة و ذلك لعدم حدوث ما يسمى memory leak اي اننا فقدنا جزء من الذاكرة بسبب امتلاء الذاكرة بقيم غير مستخدمة

و تتعلق الـ object life cycle بالـ scope ايضا فإذا عرفنا object داخل دالة الـ main بعد انتهاء دالة الـ main يتم إزالة الـ object بشكل تلقائي، اي انه بمجرد خروج الـ main من الـ stack يتم إزالة الـ object وعلى نفس المبدأ اذا تم تعريف متغير داخل دالة عاديه او داخل class اخر فستنتهي حياة هذا المتغير بمجرد انتهاء الدالة او بمجرد موت الـ object الخاص بالـ class الذي قمنا بتعريف متغير داخله

```
2 references
class cls
{
    //some codes
}

0 references
class Program
{
    1 reference
    public static void func()
    {
        cls c = new cls(); // this object lives in this function only
    }
    0 references
    static void Main(string[] args)    // first thing goes to stack
    {
        func();

        Console.ReadKey();
    }
}
```

في الصورة السابقة قمنا بتعريف object من الـ class cls داخل الدالة func وفي هذه الحالة لا يمكن استخدام هذه الـ object الا داخل هذه الدالة فقط لكن بعد انتهاء الدالة لا يكون لهذه الـ object وجود

```
2 references
class cls
{
    //some codes
}
2 references
class cls2
{
    cls c = new cls(); // this object lives in the objects of this class only
}
0 references
class Program
{

    0 references
    static void Main(string[] args)    // first thing goes to stack
    {
        cls2 c2 = new cls2();
        Console.ReadKey();

    }
}
```

في الصوره السابقة قمنا بتعريف `object` من `cls` داخل `cls` ولا يمكن استخدام هذه الـ `object` الى داخل الـ `objects` الخاصه بـ `cls` مثل `c` لكن بعد انتهاء عمل هذه الـ `c` ينتهي معه الـ `c`

نعود للعلاقات بين الـ `classes` تسمى الثلاث علاقات التالية بعلاقات الـ "Has a" و اول علاقه تسمى الـ `association`، في هذه العلاقة لا تعتمد الـ `class` على بعضها بشكل حصرى و لا تتوقف حياة كل `object` من هذه الـ `classes` على الاخر و يستطيع كل منهم العمل في غياب الاخر

مثال

الطيب و المريض ... الطبيب `has a` مريض و ايضا المريض `has` طبيب و لكل منهم `class` خاص به، فالطبيب لا يعالج مريض واحد فقط و المريض قد لا يعالج عند طبيب واحد فقط و في الصوره التاليه `class` الطبيب والمريض وداخل كل منهم متغيرين اسم الطبيب و اسم المريض و داخل `class` الطبيب توجد دالة تسمى المريض الحالى، أى المريض الذى يتم علاجه الان وهنا تحدث علاقة بين الطبيب والمريض، فداخل `class` الطبيب يحتاج معرفه اسم المريض لذلك نريد ارسال ملف المريض الى الـ `class` لأخذ الاسم و نعتبر الملف هنا هو الـ `object` الخاص بالمريض فنقوم بتمريره للدالة

```
1 reference
class Doctor
{
    public string doctor_name;
    public string patient_name;
0 references
public Doctor(string name)
{
    doctor_name = name;
}
0 references
public void current_patient(Patient obj)
{
    patient_name = obj.patient_name;
    obj.doctor_name = doctor_name;
}
}

2 references
class Patient
{
    public string doctor_name;
    public string patient_name;
0 references
public Patient(string name)
{
    patient_name = name;
}
}

}
```

```
0 references
static void Main(string[] args) // first thing goes to stack
{
    Doctor drAhmed = new Doctor("ahmed fares");
    Doctor drmohamed = new Doctor("mohamid aid");

    Patient samy = new Patient("samy ramy");
    Patient karem = new Patient("kaem mohamed");

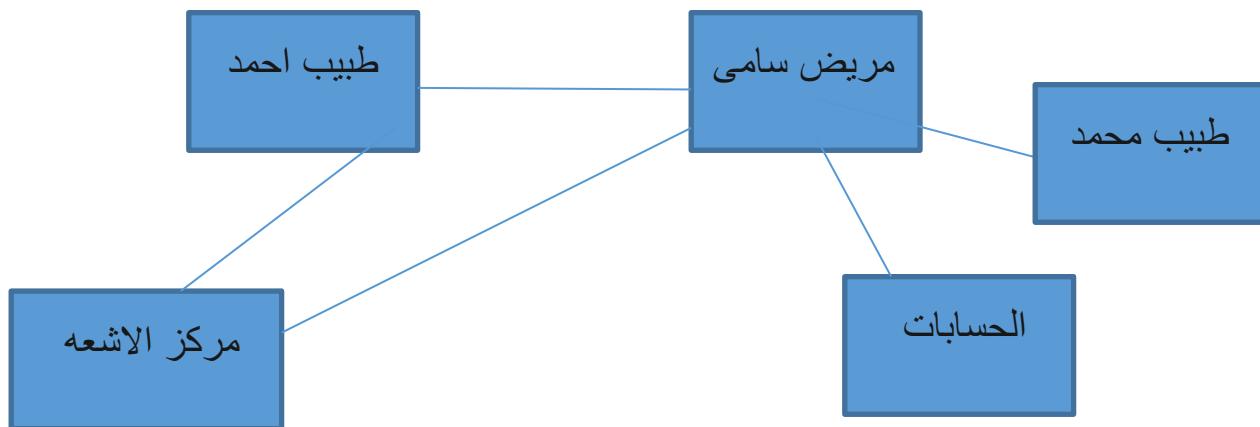
    drAhmed.current_patient(samy);
    drAhmed.current_patient(karem);
    Console.WriteLine(drAhmed.patient_name);

}
```

قمنا بعمل objects خاصه بالطبيب أحمد و الطبيب محمد و ايضا قمنا بعمل objects خاصه بالمريض سامي و كريم لا يوجد علاقه حتى الان لكن استدعينا داله المريض الحالى للطبيب أحمد و ارسلنا المريض سامي او بمعنى أصح ارسلنا ملفه الى object و هنا تم العلاقه حيث سيتم سحب اسم المريض من الملف الخاص به .

بعد ذلك دخل المريض كريم الى الدكتور سامي واذا قمنا بالكشف عن اسم المريض داخل ال object الخاص بالدكتور أحمد سنجده كريم

و بعد خروج كل المرضى الخاصه بالطبيب أحمد يظل بإمكاننا استخدام الطبيب أحمد او ال object الخاص به وكذلك المريض يمكن إستخدامه في objects خاصه بالأشعة او الخ الخ... أيان حياة كل object غير متوقفه على ال objects الاخري



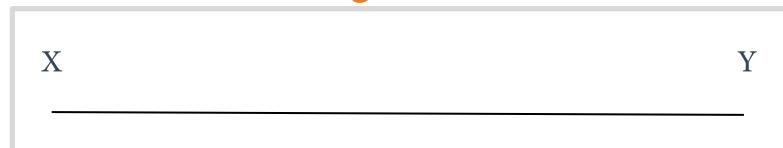
في الصوره السابقه المريض يتعامل ولديه علاقه association مع العديد من الكيانات مثل الطبيب احمد و الطبيب محمد و مركز الاشعة والحسابات و كذلك الطبيب يتعامل مع المريض سامي و مركز الاشعة و في حالة غياب احد الكائنات فإن الكيان المعتمد عليه لا يتوقف

مثال آخر.. الكتاب و المؤلف فإذا مات المؤلف هل ينتهي الكتاب بعد موته ؟ وهل اذا منع الكتاب يوم توفي المؤلف ؟

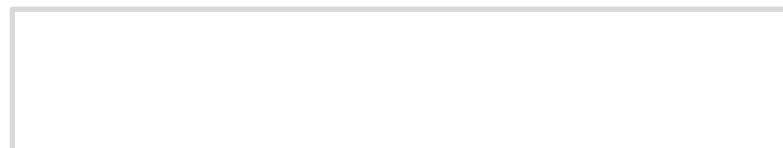
قد يكون الامر غير مفهوم ولكن ننتقل الى نوع علاقه اخرى و هو على العكس التام من العلاقة السابقه و ستساعد في فهم العلاقة السابقه وتسمى هذه العلاقة بال **composition**

في هذه العلاقة يكون احد الـ objects تابعة تماماً لـ object الاخر واذا إنتهت حياتهم معه وتسمى بالـ **Death relationship**

مثال



هذا الخط المستقيم يبدأ عند نقطة تسمى النقطه x و ينتهي عند النقطه y



و اذا قمت ازالة هذا الخط المستقيم سيتم بشكل تلقائي و منطقى ازاله النقطتين y , x اي انه مع انتهاء حياه الخط المستقيم انتهت حياه النقطتين وهذه العلاقة تسمى بعلاقة الـ **composition** في الصوره التاليه class يسمى **point**

```
class Point
{
    private int p;
    0 references
    public void setValue(int p)
    {
        this.p = p;
    }
    0 references
    public int getValue()
    {
        return p;
    }
}
```

بعد ذلك عرفنا **class** يسمى **line**

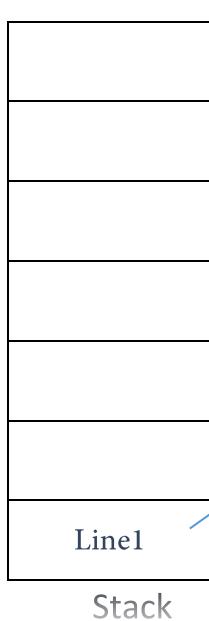
```
class Line
{
    public Point x;
    public Point y;

    0 references
    public Line(int x , int y)
    {
        this.x = new Point();
        this.y = new Point();

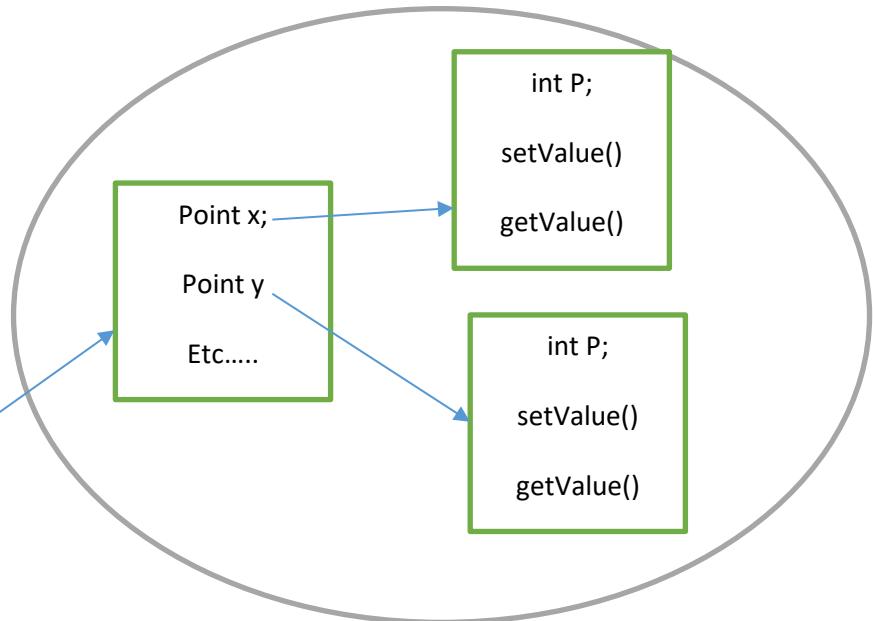
        this.x.setValue(x);
        this.y.setValue(y);
    }

    0 references
    public int length()
    {
        int length = x.getValue() - y.getValue();
        return length;
    }
}
```

داخل هذا الكلاس عرفنا **point objects** من **objects**



Stack

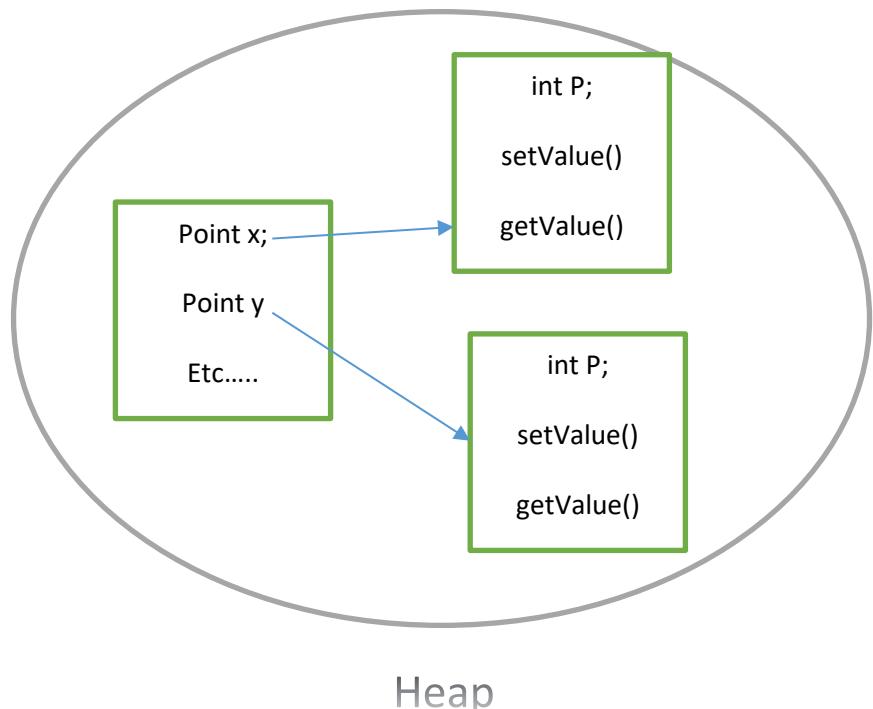


Heap

و دا خل الـ constructor اشرنا الى مساحات جديدة من كلاس point وبالتالي اصبحت الـ `x`, `y` تشير الى مساحات اخرى داخل الـ heap



Stack

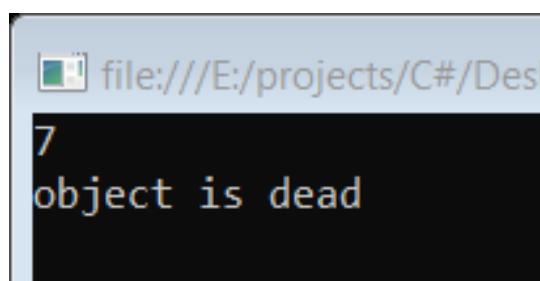


Heap

وبالتالي ستنتهي المساحة الخاصة بالـ `line1` في الـ `heap` و بالتالي سينهی معها الـ `objects` الخاصه بالـ `points` اي ان حياتهم مرتبطة بحياة الـ `object` الخاص بـ `line` وهذا ما يسمى بالـ `composition relationship`

```
0 references
static void Main(string[] args)    // first thing goes to stack
{
    Line line1 = new Line(7, 5);
    Console.WriteLine(line1.x.getValue());
    line1 = null;
    try
    {
        Console.WriteLine(line1.x.getValue());
    }
    catch (Exception e)
    {
        Console.WriteLine("object is dead");
    }
}
```

في الصوره السابقه قمنا بتعريف `object` من `line` وبعد ذلك قمنا بإستدعاء داله `getValue` الخاصه بالنقطه `x` ثم قمنا بقتل الـ `object` الخاص بـ `line` وبالتالي سيتم قتل الـ `objects` الخاصه بالـ `points` ثم قمنا بقتل `x` بعد ذلك



عند تنفيذ البرنامج سيطبع رقم 7 قيمة النقطه `x` بعد قتل الـ `line1` واستدعاء الداله الخاصه بالنقطه `x` مره اخرى، نجد ان الـ `object` الخاص بـ `x` قتل هو ايضا مثال اخر العلاقة بين الغرف و المنزل، اذا هدم المنزل فـ بالطبع لن يكون هناك غرف

ننتقل الى العلاقة الثالثة و هي وسط بين العلاقات السابقتين في هذه العلاقة هناك class يملك الـ class الآخر لكن ليس من اللازم ان يموت الـ object المملوك مع الـ object المالك وتسمى Aggregation

لتوضيح الفرق بين الـ composition and aggregation relationship نأخذ مثال ببرنامج الـ text editor عند فتح البرنامج يقوم البرنامج بحجز مساحه في الميموري تسمى الـ cache و ايضا يقوم بانشاء file جديد عند غلق البرنامج تحرر مساحة الميموري cache مع البرنامج اي ان العلاقة بينهم من نوع composite اما الـ file حياته قد تنتهي او تستمر، فقد نقوم بحفظ الملف و استخدامه مرة اخرى او استخدامه في برامج اخرى او عدم حفظه وتنتهي حياة الملف وهذه علاقة aggregation حيث ان الملفات كلها مملوكة بواسطة البرنامج و البرنامج يتحكم في بقائهما او موتها

```
6 references
class file
{
    public string name;
    public string content;
    1 reference
    public file()
    {
        name = "untitled.txt";
    }

    0 references
    public void write(string c)
    {
        content = c;
    }

}

4 references
class cach
{
    public int size;
    2 references
    public cach()
    {
        size = 50;
    }

}
```

في الصوره السابقة لدينا `class` خاص بال `file` و `cache` و كل كلاس لديه عدد من المتغيرات و في الصوره التاليه `class` خاص بال `program` و داخل الـ `class` قمنا بتعريف `static ArrayList` من النوع `file` حتى تخزن بداخلها كل الملفات التي يقوم البرنامج بصنعها ، و عرفنا `objects` من نوع `file , cache` و صنعنا بعض الدوال `create` لعمل ملف جديد و `open` لفتح ملف سابق و `save` و `unsave`

```

0 references
class program
{
    public static List<file> files = new List<file>();

    public cach fileCach;
    public file newFile;
0 references
    public void creat(string fileName)
    {
        fileCach = new cach();
        newFile = new file();
        newFile.name = fileName;

    }
0 references
    public void open(file obj)
    {
        fileCach = new cach();
        Console.WriteLine(obj.name + " is opend");
        fileCach.size = 50;
    }
0 references
    public void save()
    {
        files.Add(newFile);
        fileCach = null;
    }

0 references
    public void unsave()
    {
        newFile = null;
        fileCach = null;
    }
}

```

```

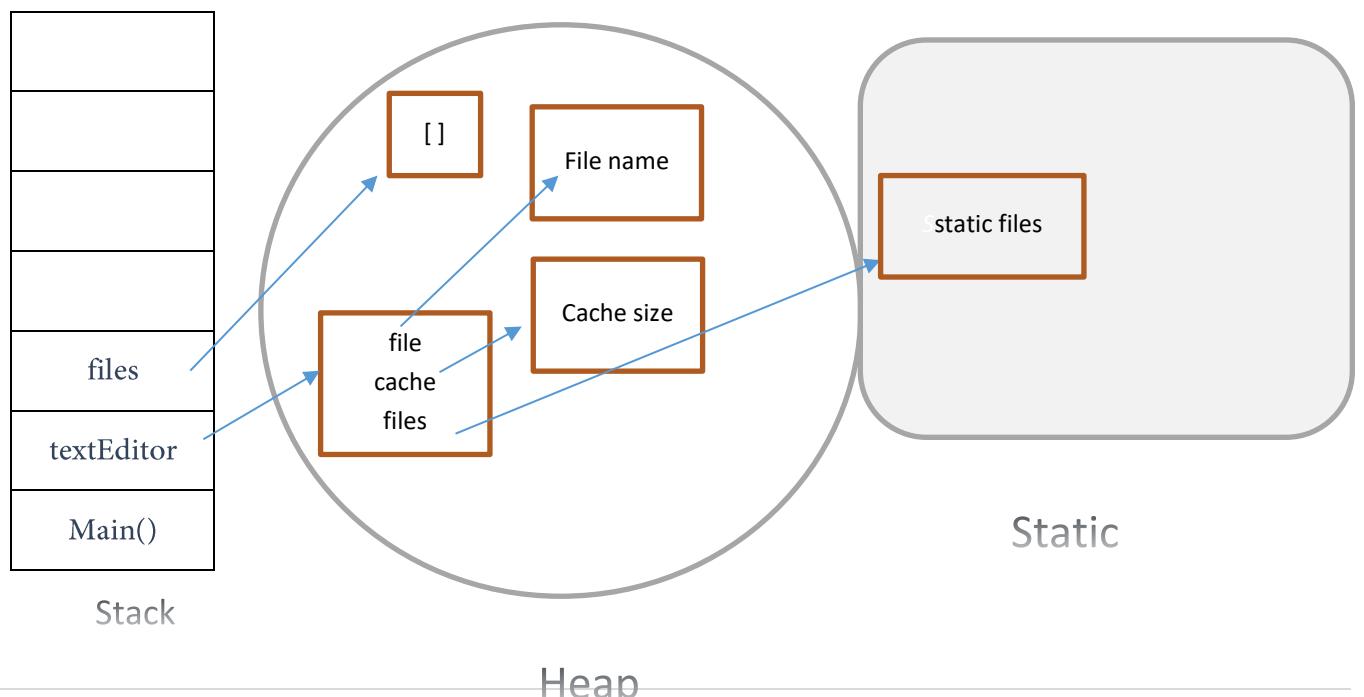
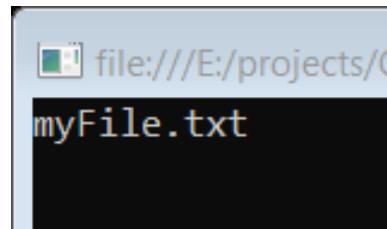
0 references
static void Main(string[] args)    // first thing goes to stack
{
    program textEditor = new program();
    List<file> files = new List<file>();
    textEditor.create("myFile.txt");
    files = textEditor.save();
    textEditor = null;
    foreach (file x in files)
    {
        Console.WriteLine(x.name);
    }

    Console.ReadLine();

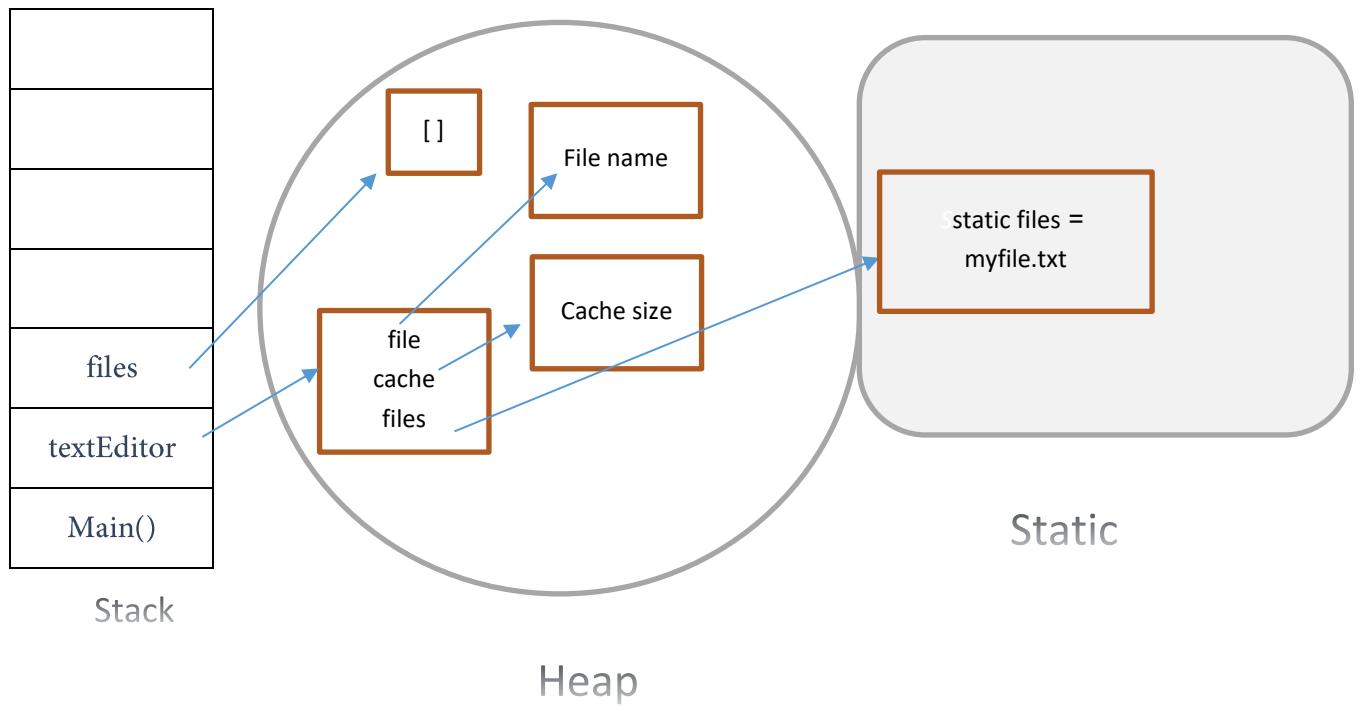
}

```

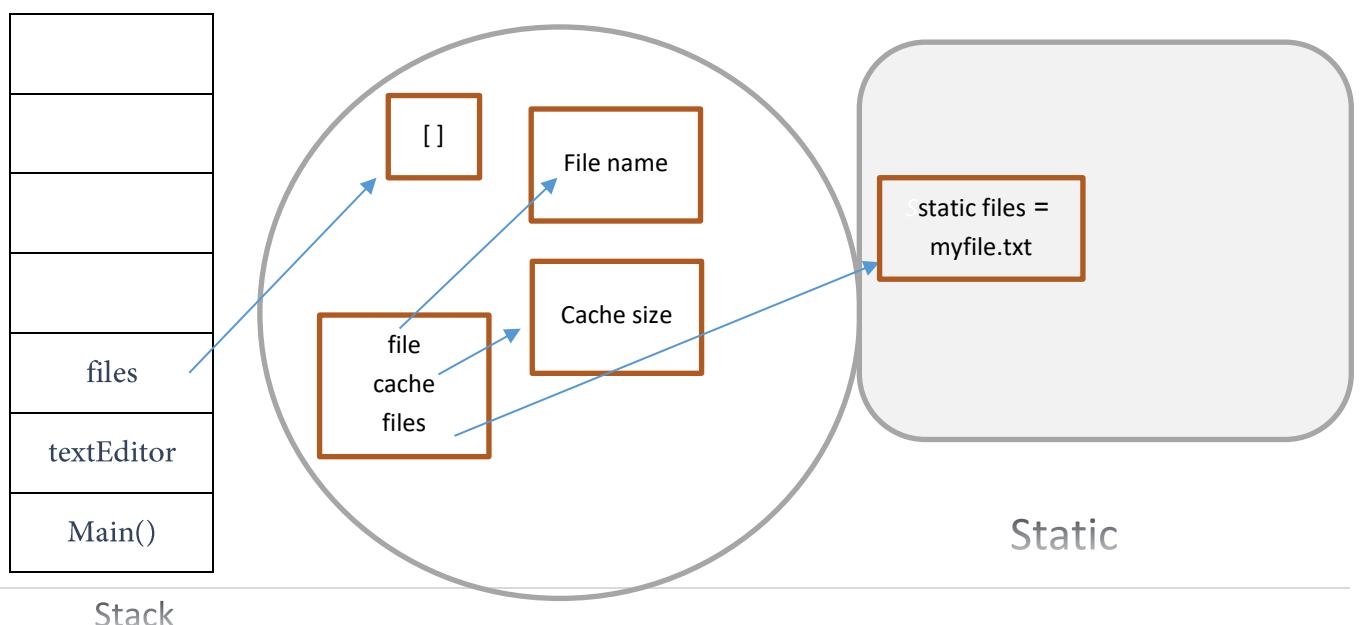
و في دالة الـ `main` قمنا بعمل `object` للبرنامجه و عرفنا `arraylist` لاستقبال الملفات المحفوظه بعد ذلك استدعينا دالة `create` لعمل ملف جديد تحت اسم `myFile.txt` بعد ذلك قمنا باستدعاء دالة `save` لحفظ هذا الملف و ترجع دالة `save` الملفات التي تم حفظها و نستقبلها في الـ `arraylist` التي قمنا بصنعها و اغلقنا البرنامج عن طريق قتل الـ `object` بفصله عن عنوانه في الـ `heap` عن طريق `null` و قمنا بطباعة اسم الملفات التي تم صنعها



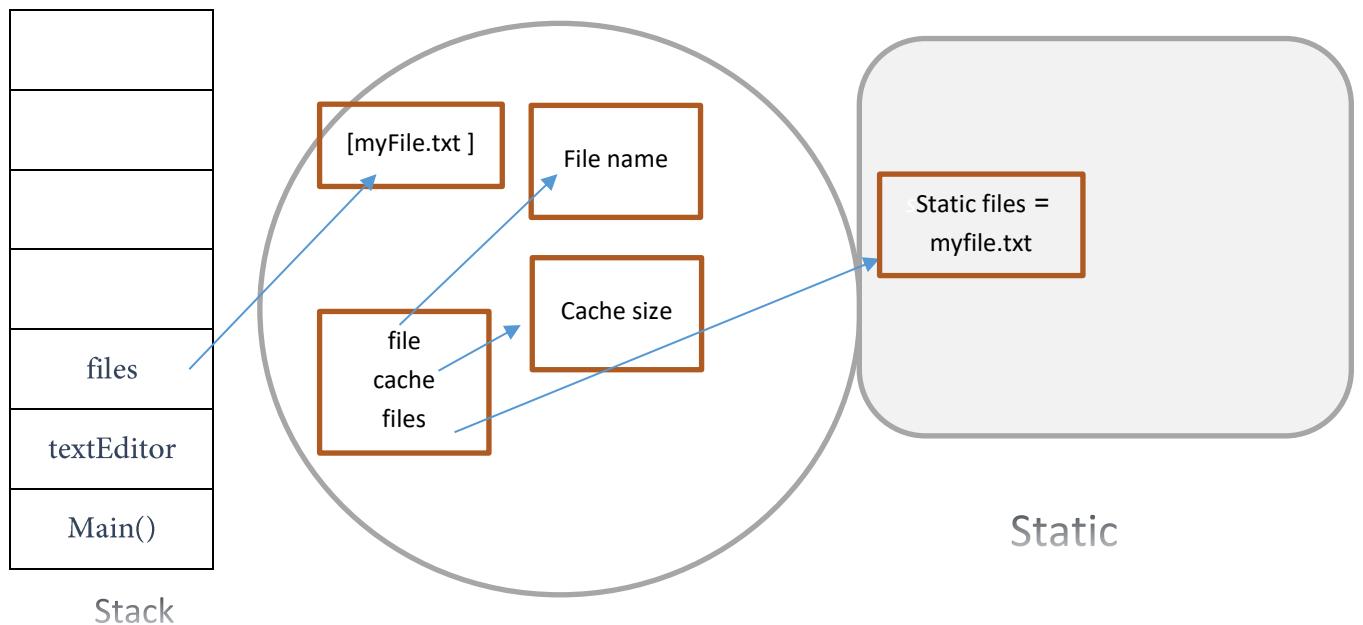
لشرح ما تم داخل الذاكرة حيث تدخل دالة الـ `main` في الـ `stack` من البرنامج ويتم حجز مساحة خاصة به في الـ `heap` وداخله يتم تعريف مساحة جديدة خاص بالـ `file` و `cache` وايضا يتم تعريف `array list` ولكن تجز في الذاكرة الخاصة بالـ `static` وذلك لانه منطقة يخزن بها كل الملفات وليس منطقة خاصة بملف واحد فقط



بعد صنع الملف يمكننا ان نحفظه او لا نحفظه وفي حالة عدم الحفظ وفصل البرنامج عن المنطقة الخاصة به في الـ `heap`



في هذه الحالة يموت الملف مع البرنامج ولكن اذا تم الحفظ سيتم ارسال الملفات الى المساحة داخل الـ heap



في هذه الحالة حتى و ان مالت البرنامج سيظل الملف على قيد الحياة و يمكن استخدامه في اي براماج اخر
حيث قمنا بعمل object ثانی من البرنامج و اعطيته الملف

```
0 references
static void Main(string[] args)    // first thing goes to stack
{
    program textEditor = new program();
    List<file> files = new List<file>();
    textEditor.creat("myfile.txt");
    files = textEditor.save();
    textEditor = null; // the textEditor object dead now

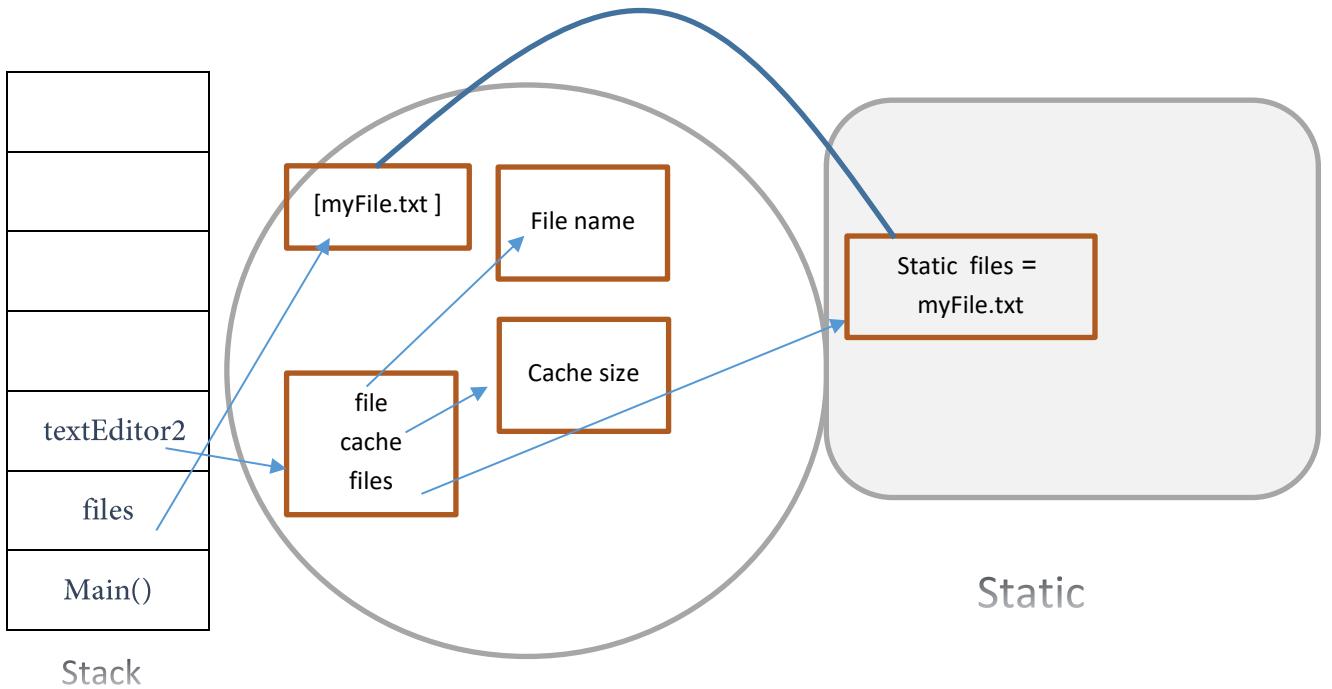
    //files = ["myfile.txt"]

    program textEditor2 = new program(); // onther program
    textEditor2.open(files[0]);

    Console.ReadLine();
}
```

```
file:///E:/projects/C#/Desktop/ConsoleApp1
myFile.txt is open
```

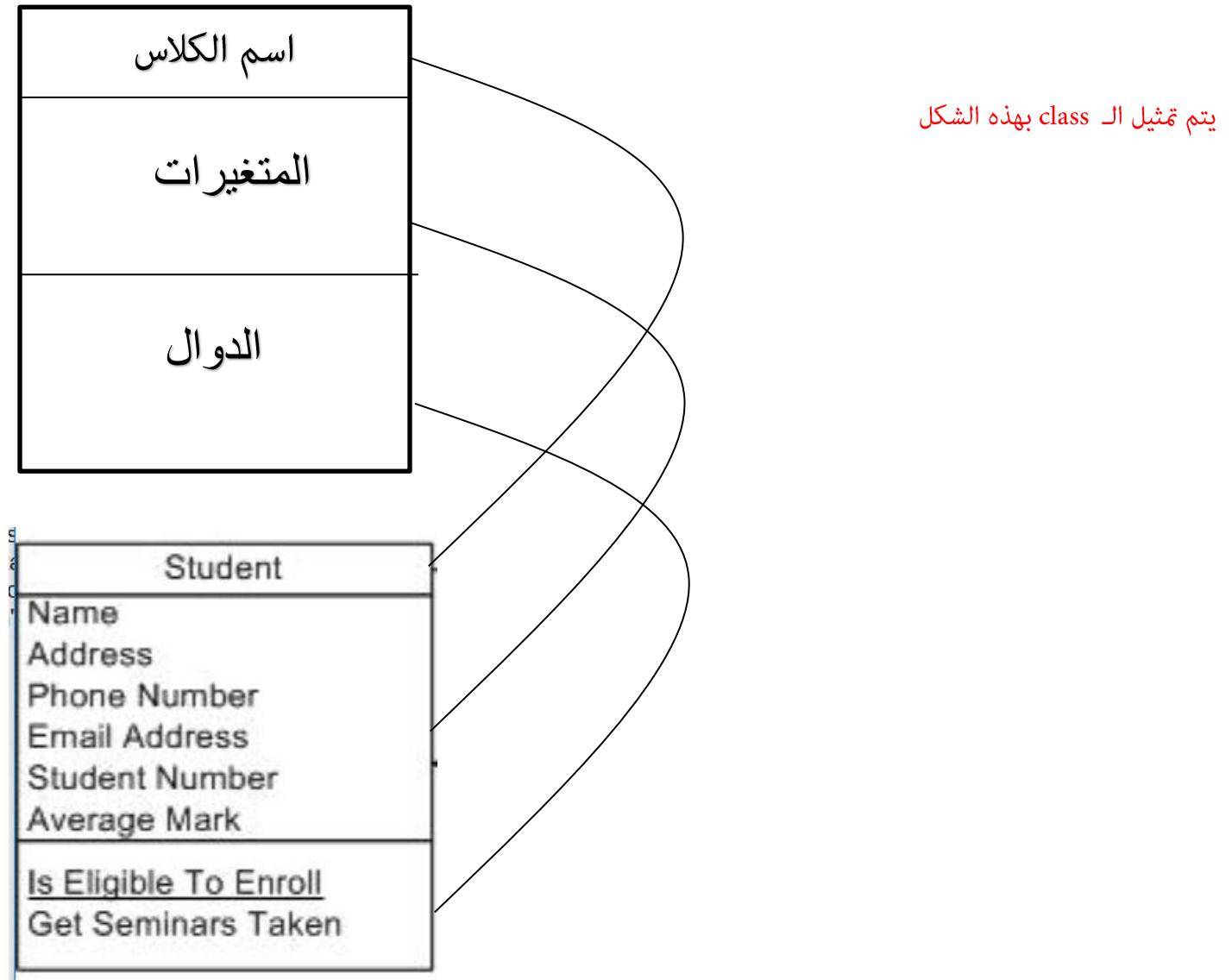
تم فتح الملف الذى صنع عن طريق برنامج اخر انتهى وخرج من الذاكرة



حيث قمنا بارسال الملف الذى صنع عن طريق برنامج اخر انتهى الى البرنامج الجديد و يستطيع العمل عليه
هذه هي علاقه الـ **aggregation** حيث انه هناك **class** مسؤل عن الـ **other class**

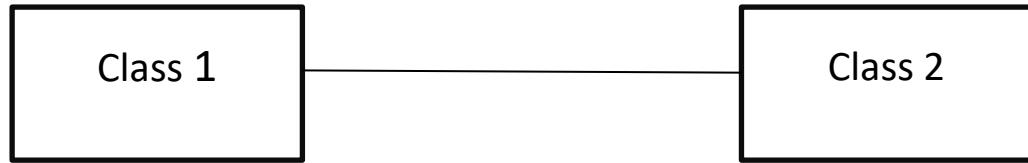
و لكن كيف يتم تمثيل هذه العلاقات عن طريق ما تحويله الى "كود" ؟

يتم تحليل البرنامج و تحديد الـ classes التي سيتم صنعها و تحديد العلاقات بينهم و كيف سيتم التعامل معها

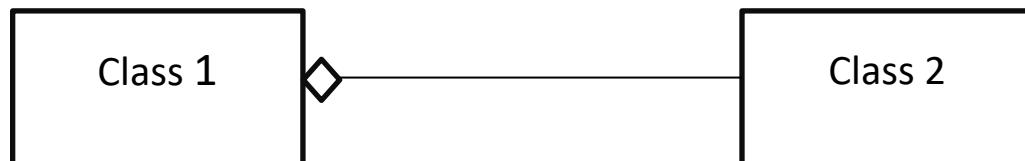


Association relationship

تمثل بخط مستقيم يربط بين كل class



Composition relationship



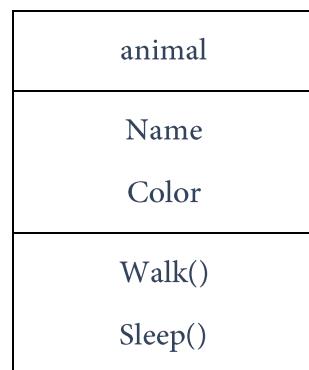
Aggregation relationship



في العلاقات السابقة تحدثنا عن ان العلاقة بينهم هي علاقة الـ **has a** فالطبيب لديه مريض و المريض لديه دكتور و كانت علاقة بين الـ **objects** الخاصه بكل **class** لكن في العلاقة القادمه هي علاقة بين الـ **classes** مباشره حتى قبل صنع **objects** منها و تسمى الـ **is a** و تسمى بعلاقة الـ **inheritance**

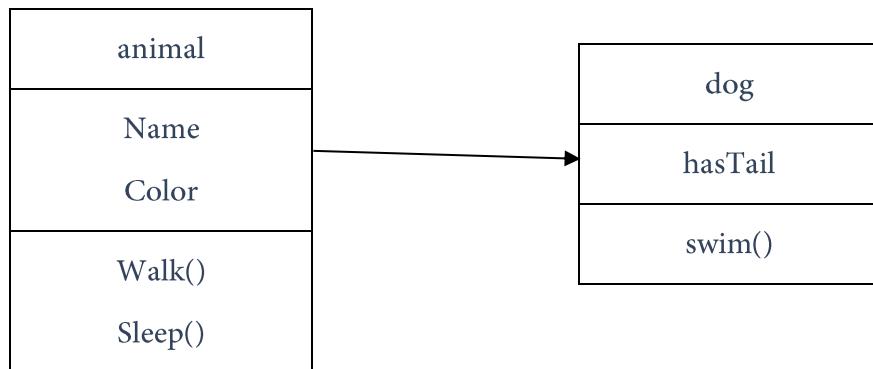
مثال

اذا كنا نريد تمثيل الحيوانات في برنامج فـ ان كل حيوان سيمثل بـ **class** مختلف عن الثاني لوجود صفات مختلفه من حيوان للثاني، فهناك حيوان يستطيع القفز و حيوان اخر لا ، او حيوان يستطيع العوم و اخر لا . لكن ايضا هناك صفات مشتركه فـ كل حيوان له اسم و صوت و نوع اكل و هكذا . و هنا تظهر مشكله تكراريه الكود حيث ان اكواود الصفات و الافعال المشتركه ستكرر لـ كل **class** يتم صنعه . ايضا اذا اردنا التعديل عليها سيطلب الامر التعديل على كل هذا العدد من الـ **class** و من هنا بدأنا **inheritance** ، الصفات المشتركه بين كل هذه الحيوانات تجمع في **animal** و لنفرض اسمه **animal**



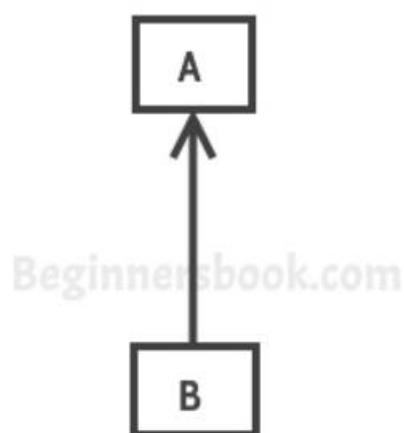
وبالتالي اذا أردنا عمل **class** خاص بـ حيوان جديد و ليكن كلب فقط كل ما نحتاجه ان نقول ان **class** الخاص بالكلب هو نوع من انوع **animal** و هي كما قلنا علاقة الـ **IS A** ، العلاقة تختصر في **dog is an animal**

ولكي تنتقل الصفات الاساسيه في **animal** الى **dog** يحتاج هذا الـ **class** ان يرث الصفات الاساسية من **animal**
ويسمى الـ **super or parent** بالـ **animal** ويسمي الـ **child** هنا بالـ **dog**



انواع الوراثة ؟

في هذه الحالة يكون هناك **child** واحد فقط يرث **parent** واحد فقط مثل المثال السابق



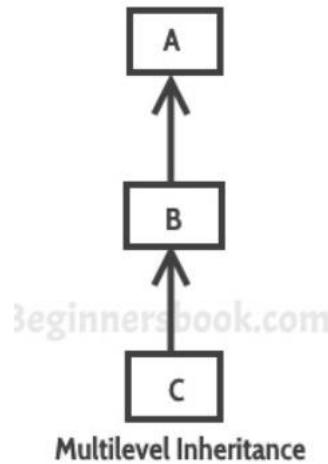
Single Inheritance

Multilevel inheritance

في هذا النوع تكون عباره عن سلسلة من الوراثات المختاليه

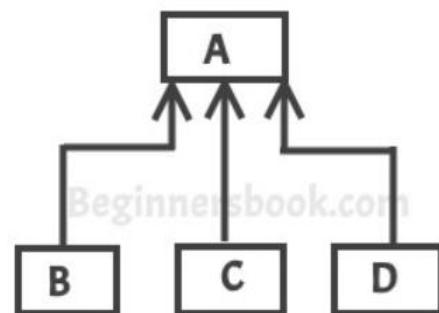
مثال

موبيل iphone 7s يرث صفات iphone 7 و الاخير يرث صفات iphone الاساسية



Hierarchical inheritance

الوراثه الهرميه حيث يرث العديد من الـ `class` صفات و افعال واحد ، في المثال الاول قد يرث الـ `animal` عدد من الحيوانات الأخرى

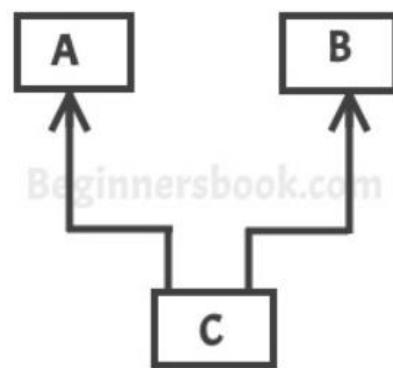


Multiple Inheritance

الوراثة التعددية حيث يمكن لـ class واحد ان يرث صفات و افعال اكثرا من class اخر

مثال

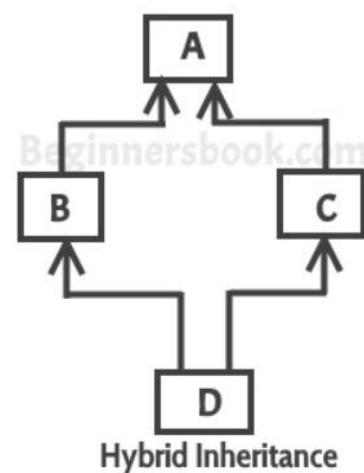
نوع من شاشات شركة سامسونج الجديدة التي تدعم نظام الاندرويد ،
هذا النوع يرث صفات و افعال من كلاسيين مختلفين TV and android ولكن الـ C# لا تدعم الـ Multiple Inheritance ستحدث عنها في وقت لاحق بعض المشاكل اهمها diamond problem



Multiple Inheritance

Hybrid Inheritance

و هي خليط من الانواع السابقة

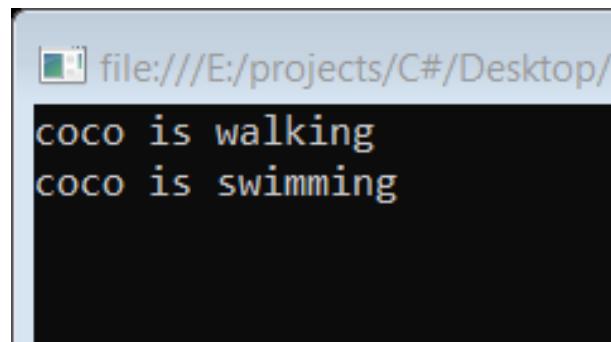


```
2 references
class Animal
{
    public string name;
    public string color;
1 reference
    public Animal(string name , string color)
    {
        this.name = name;
        this.color = color;
    }

1 reference
    public void walk()
    {
        Console.WriteLine(name + " is walking");
    }
}

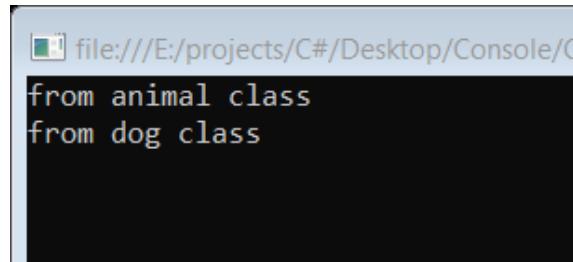
3 references
class Dog : Animal
{
    1 reference
    public Dog(string name, string color ) : base(name, color)
    {
    }
1 reference
    public void swim()
    {
        Console.WriteLine(name + " is swimming");
    }
}
```

```
0 references
static void Main(string[] args)    // first thing goes to stack
{
    Dog myDog = new Dog("coco", "red");
    myDog.walk();
    myDog.swim();
```



في الصورة الأولى قمنا بصنع `animal` و `dog` يرث الـ `animal` و تتم الوراثة عن طريق كتابة : وبعدها اسم الـ `class` الذي يتم وراثته عند عمل `object` من `dog` من `animal` نجد أنها قد ورث دالة `walk` من `animal`

وعند استدعاء الـ `class` الابن يتم استدعاء الـ `constructor` الخاص به ثم يتم استدعاء الـ `constructor` الخاص بالأب



A screenshot of a C# console application window. The title bar says "file:///E:/projects/C#/Desktop/Console/C". The code in the editor is:

```
from animal class
from dog class
```

The console window is empty, showing only the command prompt.

لكن يتم تنفيذ الـ `constructor` الخاص بالأب أولا ثم بعد ذلك الخاص بالابن و ذلك لأن دالة الـ `constructor` للابن يتدخل الى `stack` او لا ثم بعد ذلك الـ `constructor` الخاص بالأب و الـ `stack` تعمل بنظام الـ `FILO` first in last out كما شرحنا من قبل و يتم استدعاء الـ `constructor` الخاص بالأب بشكل تلقائي و يتم تمرير الـ `parameters` الى الأب كما هو موضح في الكود في الصورة الاولى

لكن ماذا لو حدث و تشابه اسم دالة في الأب مع اسم دالة في الابن ، ونتحدث هنا عن مفهوم يدعى الـ `polymorphism`

ما هو مفهوم الـ `polymorphism` ؟

تعنى الكلمة تعدد الوجه او الاشكال، فيمكن ان يكون للدالة اكثر من شكل و فعل و هيئة ويوجد طريقتين لاستخدام الـ `overriding` او `overloading` الاولى تسمى `polymorphism` والثانية الـ `overloading`

ما هو الـ `overloading` ؟

هو عندما يكون هناك اكثرا من دالة تحمل نفس الاسم ولكن تختلف في عدد او نوع الـ `parameters` سواء كانت هذه الدوال في نفس الـ `class` او في الأب و الابن

```
2 references
class Person
{

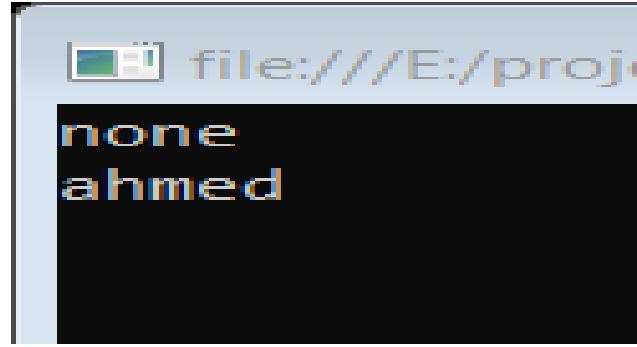
    1 reference
    public void printName()
    {
        Console.WriteLine("none");
    }
    1 reference
    public void printName(string name)
    {
        Console.WriteLine(name);
    }
}
```

```
0 references
class Program
{

    0 references
    static void Main(string[] args)
    {

        Person p = new Person();
        p.printName();
        p.printName("ahmed");

        Console.ReadLine();
    }
}
```



سنجد انه داله `printName` يمكن استخدامها بطريقتان حيث يمكن ارسال اسم اليه، أو عدم ارسال اي متغيرات وهذا ما سبق ذكره بالـ `overloading`

ما هو مفهوم الـ `overriding` ؟

يعنى هذه المفهوم اعادة صنع الداله موجوده من قبل ويعد من اهم مفاهيم الـ `oop`

مثال

لنفرض ان سياره من نوع 2018 ترث صفات سياره من نوع 2017 و من الطبيعي ان يتم التحديث في السياره من نوع 2018 عن 2017

لنفرض ان هناك داله في السياره نوع 2017 تعمل على قياس مستوى البنزين و تعطي انذار عند انخفاض المستوى و توجد نفس الخاصيه ايضا في السياره نوع 2018 لكن مع وجود تعديل بأن تقوم السياره تلقائيا بتحفيض سرعة السياره الى حد معين وهنا نتكلم عن مفهوم الـ `overloading` ف الداله التي توجد في الاب سيقوم الابن بالتعديل عليها و لكن ستظل تحمل نفس الاسم

```

3 references
class car2017
{
    1 reference
    public void FuelAlarm()
    {
        Console.WriteLine("Alert on");
    }

}
2 references
class car2018:car2017
{
    1 reference
    public void FuelAlarm()
    {
        Console.WriteLine("Alert on");
        Console.WriteLine("slow down speed");
    }

}

```

```

car2018 car18 = new car2018();
car2017 car17 = new car2017();
|
car17.FuelAlarm();
Console.WriteLine("\n");
car18.FuelAlarm();

```

```

Alert on

Alert on
slow down speed

```

عند استدعاء الدالة من `car18` سنجد انها عدلت على الدالة التي كانت في الأب أي أنها `override` للدالة داخل الأب و في بعض الأحيان يقوم الابن بالتعديل على اكتر من دالة في الأب او احيانا كل الدوال يتم عمل `override` عليها ولكن اذا كان سيتم التعديل على الاب من الأبناء ملأذ يتم صنع هذه الدوال في الأب سابقا إن كانت لا تستخدم بالفعل؟

وهنا نتحدث عن مفهوم الـ `Encapsulation` ولفهم هذا المفهوم يجب ان نعرف شيء جديد يسمى الـ `abstraction`

ما هو مفهوم الـ **Encapsulation** ؟

يقصد بهذا المفهوم هو التغليف فكلما زاد عدد الـ **classes** زاد تعقيدها و تداخلها لذاك مفهوم التغليف يستخدم كـ نوع من التنظيم الداخلي للـ **classes** و الاعضاء داخل الـ **classes**

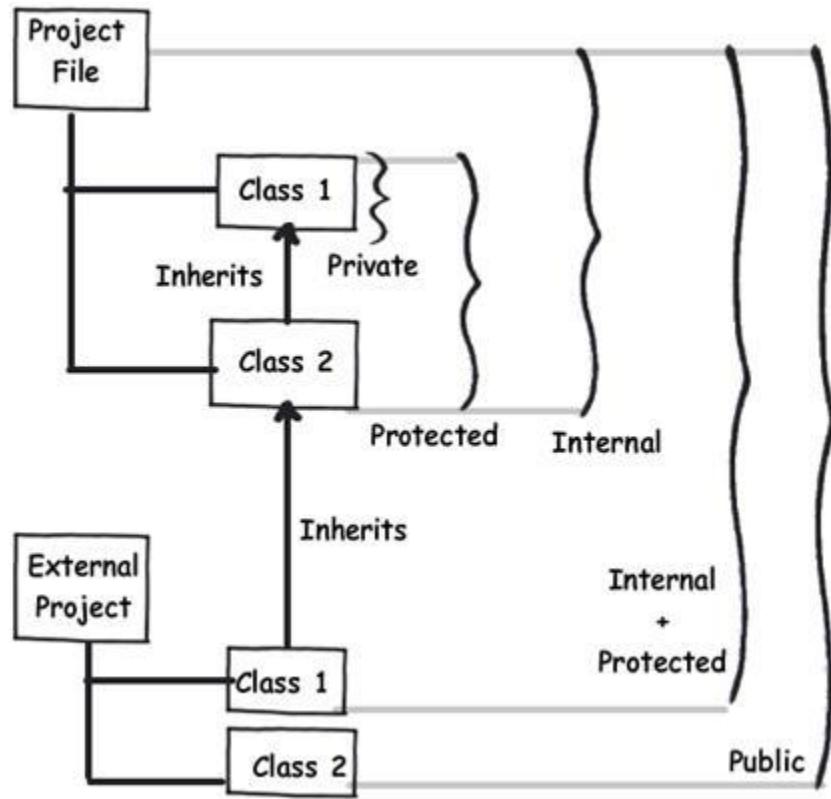
مثال

```
3 references
class worker{
    public string ssn = "159789952";
    public int hourOfWork;
    public int moneyPerHour = 50;
    public int salary;
    1 reference
    public worker(int HOW)
    {
        hourOfWork = HOW;
        salary = moneyPerHour * hourOfWork;

    }
}
0 references
static void Main(string[] args)    // first thing goes to stack
{
    worker worker1 = new worker(15);
    Console.WriteLine(worker1.hourOfWork);
    Console.WriteLine(worker1.moneyPerHour);
    Console.WriteLine(worker1.salary);
    Console.ReadLine();

}
```

في الصورة السابقة داخل الكلاس الخاص بـ **worker** يوجد رقم تحقيق الشخصيه و عدد ساعات العمل و المبلغ المتلقى لكل ساعه و المرتب الكامل و قمنا بعمل **object** للعامل **worker1** نلاحظ اننا يمكننا ان نرى كل ما بداخل هذه الـ **object** رغم عدم الحاجه الى رؤيته و من هنا جاءت فكرة الـ **encapsulation** ان يوضع نطاق للرؤيه لكل عضو من اعضاء الـ **class** و نلاحظ داخل الكلاس ان المتغيرات و الدوال مسبوقة بكلمه **public** وتعنى الكلمه ان هذا العضو يمكن رؤيته و استخدامه في اي مكان و بلا قيود



فكمما نرى في الصوره ان **private** هى اقل مصطلح لتحجيم نطاق رؤية اعضاء الكلاس حيث لا يمكن استخدام او رؤية اعضاء اي كلاس إلا بداخله فقط. **protected** اكثراً افتتاحيه من **private** حيث ان اعضاء الكلاس يمكن رؤيتها في نفس الكلاس او الكلاسات الوراثه منه يمكن ان يرى كل كلاسات المشروع كُلُّ لكن لا يمكن رؤيتها خارج المشروع **internal protected** تحمل صفة **internal** ترى في كامل المشروع و ايضاً **protected** انها ترى في الكلاس الوراثه للكلاس الحامل هذا العضو اما **public** فيمكن ان ترى في اي مكان ولكن كيف يمكن التعامل مع المتغيرات من نوع **private** في حالة اعطاء و سحب القيم منها وهنا تستخدم الـ **setters and getters**

ما هي الـ **setters and getters** ؟

هي دوال من نوع غير **private** تستخدمن لتكون حلقة الوصل لاعطاء و سحب القيم من المتغيرات من النوع **private** كما في الصوره التالية

```

0 references
class Program
{

    2 references
    class worker{
        private int value;

        1 reference
        internal void setValue(int val)
        {
            value = val;
        }
        1 reference
        internal int getValue()
        {
            return value;
        }
    }
    0 references
    static void Main(string[] args)    // first thing goes to stack
    {

        worker worker1 = new worker();
        worker1.setValue(50);
        Console.WriteLine(worker1.getValue());
    }
}

```

بعد اخفاء تنظيم الاكواد و خفاء الداتا الغير مرغوب في اظهارها ننتقل الى مفهوم الـ **abstraction**

ما هو مفهوم الـ **abstraction** ؟

الـ **abstraction** يوجد في كل شئ حولنا اذا نظرت الى السيارات ستجد هناك اشياء مشتركة بينهم سواء كانت سياره قد يه او حديده سواء كانت من نفس الشركه المنتجه او لا، كل السيارات توجد بها كشافات اضاءه و عجلة قياده و كراسي و خزان للوقود لكن تختلف في ما هو وراء هذه الواجهه، هناك سيارات عجلة القيادة بها من النوع الاماتيكي وسهل الحركه و سيارات اخر نوع آخر صعب التعامل وهكذا وهكذا، لكن في النهايه كلها يقوما بنفس الغرض

اي عندما نقول اننا نريد صنع سياره سنكتب كل هذه الاساسيات لكن كيف يتم تنفيذ هذه الاشياء و كيف ستعمل؟ هذا امر يعود الينا وهذا ما يسمى الـ **implementation**

```

1 reference
class car
{
    0 references
    public void drivingWheel()
    {
        //mechanism of the driving wheel
    }
    0 references
    public void accelerator()
    {
        //mechanism of the driving wheel
    }

}
2 references
class mycar:car
{
}

```

Car هو كلاس يحمل كل الصفات و الافعال الاساسية في السياره و الكلاس الثاني هو myCar وهو يرث الصفات الاساسيه من الأب car لكن في المثال السابق انا ارث ايضا تنفيذ هذه الدوال من الأب لذلك إذا أردت أن اقوم بالتعديل و عمل طريقة تنفيذ خاصه بي يجب عمل overriding لهذه الدوال كما شرحنا ومن هنا جاءت فكره الـ abstraction في الكود الدوال الخاصه بالكلاس car سيتم التعديل عليها من أى ابن سيرثها لذلك ليس لها اهميه في الأب لكن في الجانب الآخر يجب ان تكون موجوده حتى يرثها الابن لانها صفات اساسيه لذلك نستخدم بدلا منها ما يسمى بالـ abstract function

ما هي الـ **abstract function** ؟

هي دالة لا تحمل طريقة لعمل implementation مثل الدوال الأخرى و تستخدم كالمثال السابق في الأب حتى يتم عليها الـ overriding

و تستخدم هذا الدوال بداخل الـ class او interface او abstract

ما هو ال **abstract class** ؟

هو نوع خاص من الـ **classes** يستطيع حمل الدوال العاديه و الدوال من النوع **abstract**

```
1 reference
abstract class car
{
    1 reference
    public abstract void drivingWheel();

    0 references
    public void accelerator()
    {
        //mechanism of the driving wheel
    }

}

2 references
class mycar : car
{
    1 reference
    public override void drivingWheel()
    {
        Console.WriteLine("overriding done |");
    }
}
```

المثال السابق بعد استخدام الـ **abstract** نلاحظ ان الكلاس **car** من نوع **abstract** يستطيع حمل دوال عاديه و دوال
و ان دوال **abstract** لا تحمل اكواد لعمل **implementation** وفي الكلاس الابن **mycar** اصبح من الاجبارى أن يتم عمل
overriding
على الداله من نوع **abstract**

في المثله السابقه علمنا ان الـ **abstraction class** يستطيع دوال من النوع العادي و دوال من النوع **abstraction** و تسمى ايضا بالـ **full-abstraction** لكن ماذا اذا اردنا تحقيق **semi-abstraction** وهذا ننتقل الى الـ **interface**

ما هو الـ **interface** ؟

في المفهوم السابق **abstract class** هو في الاصل كلاس عادي لكن بصفات خاصه لكن الـ **interface** ليس **class** و يحقق %100 **abstraction** حيث ان كل الدوال داخله تكون **abstract function** ولذلك فإن جميعهم يجب ان يحدث لهم **overriding** لذا فإن الكلاس لا يرث الـ **interface** لكن وظيفة الكلاس هنا هو عمل **implementation** للـ **interface**

```
1 reference
interface car
{
    1 reference
    void drivingWheel();

}

2 references
class mycar : car
{
    1 reference
    public | void drivingWheel()
    {
        Console.WriteLine("overriding done ");
    }
}
```

و يتميز الـ **interface** بامكانيه حل مشكله عدم وجود **multiple inheritance**

```

1 reference
interface car
{
    1 reference
    void drivingWheel();
}
2 references
interface design
{
    1 reference
    void shape();
}
2 references
class mycar : car , design
{
    1 reference
    public void drivingWheel()
    {
        Console.WriteLine("from car interface");
    }

    1 reference
    void design.shape()
    {
        Console.WriteLine("from shape interface");
    }
}

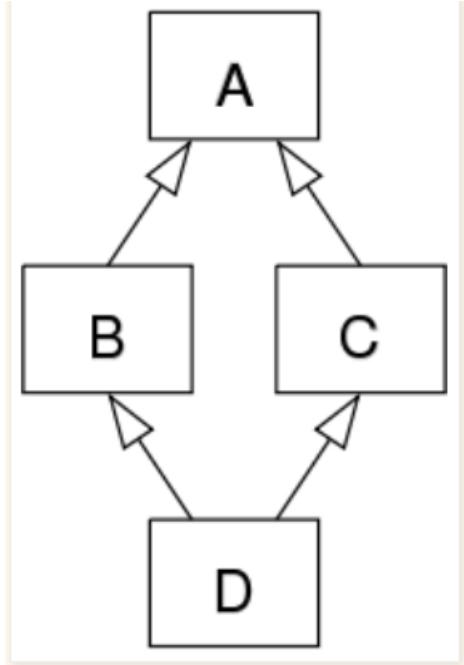
```

كما في الصورة السابقة فـ `class` يقوم بعمل "وراثه" للاثنين من الـ `car interfaces` و `design`

ولكن لماذا يوجد وراثه تعددية لـ `interface` و لا يوجد ذلك لـ `classes` ؟

فـ اكبر الشاكل الوراثه التعددية في الـ `class` هي مشكله تسمى الـ `diamond problem`

ما هي مشكلة ال diamond problem؟



في الصورة السابقة رسمة توضح مشكلة ال diamond problem فاذا كان هناك كلاسات **A** **B** **C** **D** فإذا كان **A** يرث **B** و **C** و **B** و **C** يرثان نفس الدالة **f** اي ان **B** و **C** يملكان نفس الدالة **f** و يمكنهما عمل override على الدالة **f** معا على فرض ان اللغة تدعم الوراثة المتعدد و هنا المشكله كل الكلاسين **B** و **C** ليديهم نفس الدالة اي منهم سيرث **D**؟

قامت بعض اللغات الاخرى بحل المشكله عن طريق ما يسمى بال **virtual function** لاجبار الابن الذي يكون امامه وراثه دالتين متطابقتين باستخدام واحده فقط منهم

اما ال **interface** يدعم الوراثه التعددية لانه في الحاله السابقة ستكون الدالة من النوع **abstract** وسيتم عمل ال **implementation** لها في نفس المكان وهو الكلاس **D**

كانت هذه اخر ما يحتويه الكتيب من برح لـ object oriented programming

و وجب الاشاره ان الكتيب لا يحتوى على كل ما يمكن برحه في هذه الموضوع لكن مجرد بدايه لدخول الى عالم الـ oop ومحاوله تبسيط المفاهيم وربطها بعض

فإن وجديم في طيات هذا الكتاب أخطاء لغوية أو تقنية أو لديكم ملاحظات

واقيراحات لتحسين السلسله فلا ترددوا بمراسلتنا عبر العنوان الالكترونيه

التالية:

Afaresahmed19@gmail.com

<https://www.facebook.com/a7medfares1994>