

LESSON 09

I. TỪ KHÓA STATIC TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

❖ Nội dung học:

- Đặc điểm của thành viên tĩnh.
- Biến tĩnh.
- Phương thức tĩnh
- Lớp tĩnh.
- Phương thức khởi tạo tĩnh

❖ Đặc điểm của thành viên tĩnh:

Bình thường các thuộc tính, phương thức sẽ có đặc điểm:

- Chỉ có thể sử dụng sau khi khởi tạo đối tượng.
- Dữ liệu thuộc về riêng mỗi đối tượng (xét cùng 1 thuộc tính thì các đối tượng khác nhau thì thuộc tính đó sẽ mang các giá trị khác nhau).
- Được gọi thông qua tên của đối tượng.

Đôi lúc người lập trình mong muốn 1 thuộc tính nào đó được dùng chung cho mọi đối tượng (chỉ được cấp phát 1 vùng nhớ duy nhất). Từ đó khái niệm **thành viên tĩnh** ra đời.

Đặc điểm của thành viên tĩnh:

- Được khởi tạo **1 lần duy nhất** ngay khi biên dịch chương trình.
- Có thể **dùng chung** cho mọi đối tượng.
- **Được gọi thông qua tên lớp.**
- Được **huỷ khi kết thúc** chương trình

Có 4 loại thành viên tĩnh chính:

- Biến tĩnh (**static variable**).
- Phương thức tĩnh (**static method**).
- Lớp tĩnh (**static class**).
- Phương thức khởi tạo tĩnh (**static constructor**).

Để khai báo 1 thành viên tĩnh ta sử dụng từ khoá **static** đặt trước tên biến, tên phương thức hoặc tên lớp.

❖ Biến tĩnh

➤ Cú pháp

<phạm vi truy cập> static <kiểu dữ liệu> <tên biến> = <giá trị khởi tạo>;

➤ Trong đó:

- **<phạm vi truy cập>** là các phạm vi đã trình bày .
- **static** là từ khoá để khai báo thành viên tĩnh.
- **<tên biến>** là tên biến do người dùng đặt và tuân thủ các quy tắc đặt tên biến.
- **<giá trị khởi tạo>** là giá trị ban đầu mà biến tĩnh này chứa. Nếu bạn không khai báo giá trị này thì C# thì tự gán giá trị mặc định và đưa ra 1 cảnh báo khi bạn biên dịch chương trình.

➤ Ý nghĩa:

- Một biến dùng chung cho mọi đối tượng thuộc lớp.
- Nó được khởi tạo vùng nhớ 1 lần duy nhất ngay khi chương trình được nạp vào bộ nhớ để thực thi và huỷ khi kết thúc chương trình.

Ngoài biến tĩnh ra thì hằng số cũng có thể được gọi thông qua tên lớp và không cần khởi tạo đối tượng nhưng biến tĩnh linh hoạt hơn đó là có thể thay đổi giá trị khi cần thiết.

Về cách sử dụng thì bạn thao tác hoàn toàn giống 1 biến bình thường chỉ cần lưu ý là phải gọi biến này thông qua tên lớp.

➤ Ví dụ: (Mở vd + chứa)

```
namespace Lesson09
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.OutputEncoding = ASCIIEncoding.UTF8;
            // Biến tĩnh
            Box box01 = new Box();
            Box box02 = new Box();
            Box box03 = new Box();
            Box box04 = new Box();

            Console.WriteLine("Số lượng box đã tạo ra là: " + Box.CountBox);
            Console.ReadKey();
        }
    }

    10 references
    class Box
    {
        //Thuộc tính
        public double Dai;
        public double Rong;
        public double Cao;
        public double DTXQ;
        public double TheTich;

        public static int CountBox = 0;
    }
}
```

❖ Phương thức tĩnh

➤ Cú pháp

```
<phạm vi truy cập> static <kiểu trả về> <tên phương thức>
{
    // nội dung phương thức
}
```

➤ Trong đó:

- **<phạm vi truy cập>** là các phạm vi đã trình bày .
- **static** là từ khoá để khai báo thành viên tĩnh.
- **<kiểu trả về>** là kiểu trả về của phương thức.
- **<tên phương thức>** là tên do người dùng đặt và tuân thủ các quy tắc đặt tên

➤ Ý nghĩa:

- Hàm tĩnh là 1 hàm dùng chung của lớp. Được gọi thông qua tên lớp và không cần khởi tạo bất kỳ đối tượng nào, từ đó tránh việc lãng phí bộ nhớ
- Hỗ trợ trong việc viết các hàm tiện ích để sử dụng lại.

Về sử dụng thì bạn thao tác hoàn toàn giống 1 phương thức bình thường chỉ cần lưu ý là phải gọi phương thức này thông qua tên lớp.

➤ Ví dụ: (Mở vd + chứa)

```
// Phương thức tĩnh
Box.Selfie();
Console.ReadKey();

}

}

11 references
class Box
{
    //Thuộc tính
    public double Dai;
    public double Rong;
    public double Cao;
    public double DTXQ;
    public double TheTich;

    public static int CountBox = 0;

    1 reference
    public static void Selfie()
    {
        Console.WriteLine("Xin chào! Tôi là họ nhà Box đẹp zai.");
    }
}
```

❖ Lớp tĩnh

➤ Cú pháp

```
<phạm vi truy cập> static class <tên lớp>
{
    // các thành phần của lớp
}
```

➤ Trong đó:

- <phạm vi truy cập> là các phạm vi đã trình bày .
- **static** là từ khoá để khai báo thành viên tĩnh.
- **class** là từ khoá để khai báo lớp.
- <tên lớp> là tên do người dùng đặt và tuân thủ các quy tắc đặt tên

➤ Đặc điểm:

- Chỉ chứa các thành phần tĩnh (biến tĩnh, phương thức tĩnh)
- Không thể khai báo, khởi tạo 1 đối tượng thuộc lớp tĩnh.
- Mọi thứ đều được truy cập thông qua tên lớp.

➤ Ví dụ:

Lớp **Math** chứa:

- Các hằng số như PI, E.
- Các phương thức hỗ trợ tính toán như: sin, cos, tan, sqrt, exp, . . .

II. KẾ THỪ TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG.

❖ Khái niệm kế thừa.

Trong thực tế, **kế thừa** là việc thừa hưởng lại những gì mà người khác để lại. Ví dụ: con kế thừa tài sản của cha, . . .

Trong lập trình cũng vậy, **kế thừa trong lập trình** là cách 1 lớp có thể **thừa hưởng** lại những **thuộc tính**, **phương thức** từ 1 lớp khác và sử dụng chúng như là của bản thân mình.

Một định nghĩa trừu tượng hơn về kế thừa: là một đặc điểm của ngôn ngữ hướng đối tượng dùng để biểu diễn mối quan hệ **đặc biệt hoá – tổng quát hoá** giữa các lớp.

❖ Ưu điểm kế thừa.

- Cho phép xây dựng 1 lớp mới từ lớp đã có.
 - Lớp mới gọi là **lớp con (subclass)** hay **lớp dẫn xuất (derived class)**.
 - Lớp đã có gọi là **lớp cha (superclass)** hay **lớp cơ sở (base class)**.
- Cho phép chia sẻ các thông tin chung nhằm tái sử dụng và đồng thời giúp ta dễ dàng nâng cấp, dễ dàng bảo trì.
- Định nghĩa sự tương thích giữa các lớp, nhờ đó ta có thể chuyển kiểu tự động.

➤ Cú pháp:

```
Class <tên lớp con> : <tên lớp cha>
{
}
}
```

➤ Trong đó:

- **class** là từ khoá để khai báo lớp.
- **<tên lớp con>** là tên do người dùng đặt và tuân theo các quy tắc đặt tên
- **<tên lớp cha>** là tên lớp mà ta muốn kế thừa các đặc tính của nó

➤ Sử dụng:

```
1 reference
class Animal
{
    protected double Weight;
    protected double Height;
    protected static int Legs;
    1 reference
    public void Info()
    {
        Console.WriteLine(" Weight: " + Weight + " Height: " + Height + " Legs: " +
            Legs);
    }
}

3 references
class Cat : Animal
{
    1 reference
    public Cat()
    {
        //Lớp Cat kế thừa lớp Animal
        Weight = 500;
        Height = 20;
        Legs = 2;
    }

    1 reference
    public void Speak()
    {
        Console.WriteLine("I am Cat. meo meo meo...");
    }
}
```

```
0 references
static void Main(string[] args)
{
    Cat meomeo = new Cat();
    meomeo.Info();
    meomeo.Speak();
    Console.ReadKey();
}
```

❖ Các vấn đề trong kế thừa.

🚧 Vấn đề về phương thức khởi tạo, phương thức hủy bỏ.

Phương thức khởi tạo mặc định của lớp cha luôn luôn được gọi mỗi khi có 1 đối tượng thuộc lớp con khởi tạo. Và được gọi trước phương thức khởi tạo của lớp con.

Nếu như lớp cha có phương thức khởi tạo có tham số thì đòi hỏi lớp con phải có phương thức khởi tạo tương ứng và thực hiện gọi phương thức khởi tạo của lớp cha thông qua từ khoá **base**.

Giả sử lớp **Animal** có thêm 1 **constructor** có tham số như sau:

```
//Vấn đề khởi tạo  
0 references  
public Animal(double w, double h, int l)  
{  
    Weight = w;  
    Height = h;  
    Legs = l;  
}
```

Khi đó lớp con kế thừa từ lớp **Animal** phải có **constructor** có các tham số tương ứng:

```
/*  
public Cat() //vấn đề khởi tạo => ko dùng đk.  
{  
    //Lớp Cat kế thừa lớp Animal  
    Weight = 500;  
    Height = 20;  
    Legs = 2;  
}  
*/  
  
//Vấn đề khởi tạo phải gọi lại với từ khóa base  
1 reference  
public Cat(double weight, double height, int legs): base(weight, height, legs)  
{  
    ...  
}
```

Qua ví dụ trên bạn dễ dàng thấy được cú pháp để gọi **constructor** của lớp cha như sau:

➤ Cú pháp

```
public <tên lớp>(<đanh sách tham số của lớp con>) : base(<đanh sách tham số>)  
{  
    ...  
}
```

➤ Trong đó:

- **<tên lớp>** là tên lớp con (lớp dẫn xuất).
- **<đanh sách tham số của lớp con>** là danh sách tham số của constructor của lớp con.
- **base** là từ khoá để gọi đến constructor của lớp cha.
- **<đanh sách tham số>** là danh sách tham số tương ứng với constructor của lớp cha.

Khi đối tượng của lớp con bị huỷ thì phương thức huỷ bỏ của nó sẽ được gọi trước sau đó mới gọi phương thức huỷ bỏ của lớp cha để huỷ những gì lớp con không huỷ được.

Vấn đề hàm trùng tên và cách gọi phương thức của lớp cha.

Lớp Animal có phương thức TakeABath, lớp Cat kế thừa cũng muốn định nghĩa 1 phương thức như vậy.

- ⇒ Vấn đề sẽ không xảy ra lỗi nhưng trình biên dịch sẽ gọi phương thức của lớp Cat và đồng thời đưa ra cảnh báo biên dịch
- ⇒ Trong C# có hỗ trợ từ khoá **new** nhằm đánh dấu đây là 1 hàm mới và hàm kế thừa từ lớp cha sẽ bị che đi khiến **bên ngoài** không thể gọi được.

Ví dụ:

```
//Vấn đề hàm trùng tên
1 reference
public new void TakeABath()
{
    Console.WriteLine("Take a bath by licking");
}
```

Khi đó hàm **TakeABath()** của lớp cha sẽ bị che giấu đi. Và mọi đối tượng **bên ngoài** chỉ gọi được hàm **TakeABath()** của lớp **Cat**.

Từ khoá này chỉ làm tường minh khai báo của hàm **TakeABath()** chứ về kết quả khi chạy chương trình sẽ không có thay đổi.

Đến đây 1 câu hỏi nữa được đặt ra: **Vậy có cách nào gọi hàm Info() của lớp cha được nữa không?**

Câu trả lời là có nhưng chỉ có thể gọi trong nội bộ của lớp Cat mà thôi.

Bạn có thể sử dụng từ khoá **base** để đại diện cho lớp cha và gọi đến các thành phần của lớp cha.

```
//Vấn đề hàm trùng tên
1 reference
public new void TakeABath()
{
    Console.WriteLine("Take a bath by licking");
    base.TakeABath();
}
```


Kết quả khi gọi hàm **TakeABath()** của lớp **Cat**.

```
G:\My Drive\C Sharp and Revit API Basic\Pham Hoan\Source\Visual Studio Source Code\Lesson09\bin\Debug\Lesson09.exe
Take a bath by licking
Take a bath by water
```

Vấn đề cấp phát vùng nhớ cho đối tượng

“Một đối tượng thuộc lớp cha có thể tham chiếu đến vùng nhớ của đối tượng thuộc lớp con nhưng ngược lại thì không”

Có nghĩa là nếu lớp **Cat** kế thừa từ lớp **Animal** thì câu lệnh **Animal cat = new Cat();** hoàn toàn đúng nhưng ngược lại **Cat cat = new Animal ();** sẽ báo lỗi.

```
//Vấn đề cấp phát vùng nhớ
Animal miumiu = new Cat();
Cat miuYo = new Animal();
```

III. ĐA HÌNH TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

❖ Đa hình là gì?

- **Tính đa hình** là hiện tượng các đối tượng thuộc các lớp khác nhau có thể hiểu cùng 1 thông điệp theo các cách khác nhau.

Để thể hiện được **tính đa hình**:

- Các lớp phải có quan hệ kế thừa với cùng 1 lớp cha nào đó.
- Phương thức đa hình phải được ghi đè (**override**) ở các lớp con (sẽ được trình bày ngay sau đây).

❖ Từ khoá **virtual** và từ khoá **override**:

- **Virtual** là từ khoá dùng để khai báo 1 phương thức ảo (phương thức ảo là phương thức có thể ghi đè được).
- **Override** là từ khoá dùng để đánh dấu phương thức ghi đè lên phương thức của lớp cha

➤ **Lưu ý:**

- Chỉ có thể ghi đè lên phương thức **virtual** hoặc **abstract**.
- Tính đa hình chỉ được thể hiện khi đã ghi đè lên phương thức của lớp cha.

➤ **Ví dụ:**

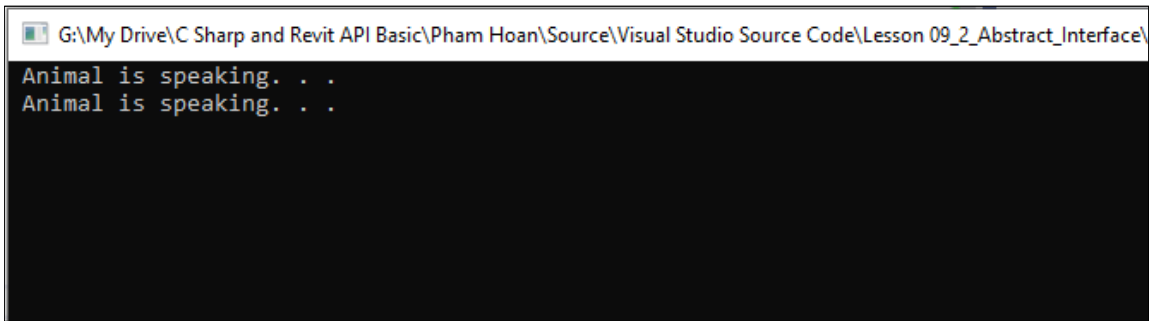
```
4 references
class Animal
{
    2 references
    public void Speak()
    {
        Console.WriteLine(" Animal is speaking. . .");
    }
}
1 reference
class Cat : Animal
{
    0 references
    public void Speak()
    {
        Console.WriteLine(" Cat is speaking. . .");
    }
}
1 reference
class Dog : Animal
{
    0 references
    public void Speak()
    {
        Console.WriteLine(" Dog is speaking. . .");
    }
}
```

```
0 references
static void Main(string[] args)
{
    Animal cat = new Cat();
    Animal dog = new Dog();

    cat.Speak();
    dog.Speak();

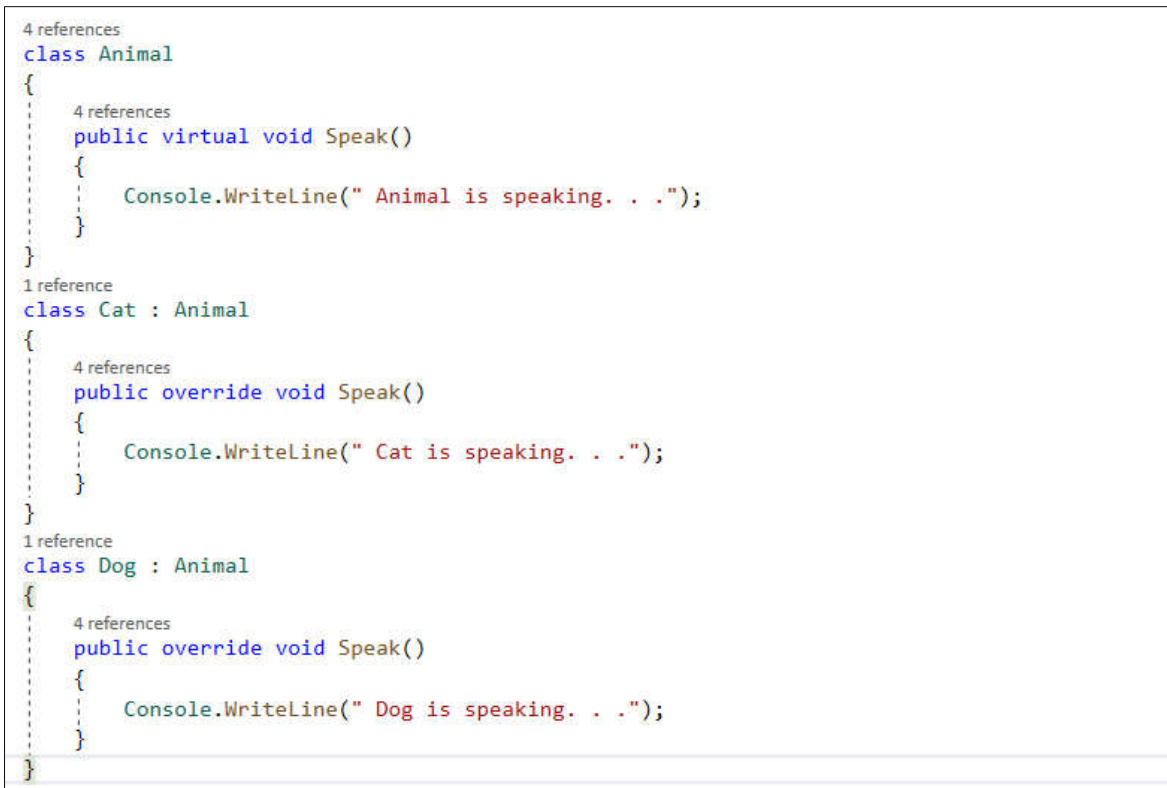
    Console.ReadKey();
}
```

Ta mong muốn chương trình sẽ gọi đúng phương thức **Speak()** của lớp đã được cấp phát vùng nhớ. Nhưng thực tế không phải vậy



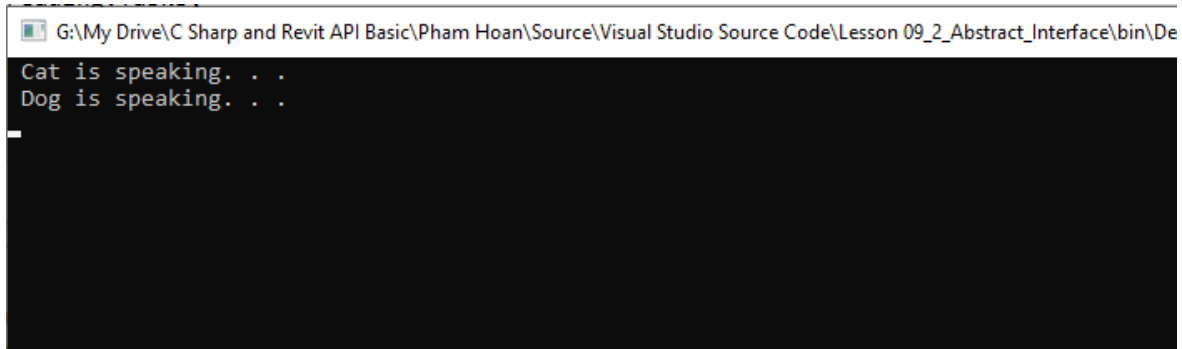
```
G:\My Drive\C Sharp and Revit API Basic\Pham Hoan\Source\Visual Studio Source Code\Lesson 09_2_Abstract_Interface\
Animal is speaking. . .
Animal is speaking. . .
```

Lúc này ta cần phải **override** phương thức **Speak()** của lớp cha (lớp **Animal**) và để **override** được thì ta cần khai báo phương thức **Speak()** của lớp cha là phương thức ảo (**virtual**).



```
4 references
class Animal
{
    4 references
    public virtual void Speak()
    {
        Console.WriteLine(" Animal is speaking. . .");
    }
}
1 reference
class Cat : Animal
{
    4 references
    public override void Speak()
    {
        Console.WriteLine(" Cat is speaking. . .");
    }
}
1 reference
class Dog : Animal
{
    4 references
    public override void Speak()
    {
        Console.WriteLine(" Dog is speaking. . .");
    }
}
```

Chạy lại hàm **main** trên ta được:



```
G:\My Drive\C Sharp and Revit API Basic\Pham Hoan\Source\Visual Studio Source Code\Lesson 09_2_Abstract_Interface\bin\De
Cat is speaking. . .
Dog is speaking. . .
```

Đây cũng chính là ví dụ sử dụng tính đa hình.

Ta thấy 2 đối tượng dog, cat được cấp phát 2 vùng nhớ thuộc 2 lớp 2 khác nhau nhưng khi cùng gọi phương thức **Speak()** thì đối tượng tham chiếu đến vùng nhớ của lớp nào sẽ được gọi đúng phương thức của lớp đó.

❖ Lớp trừu tượng và phương thức thuần ảo

Phương thức thuần ảo là 1 phương thức ảo và không có định nghĩa bên trong.

Lớp trừu tượng là lớp chứa phương thức thuần ảo.

Abstract là từ khoá dùng để khai báo 1 lớp trừu tượng hoặc 1 phương thức thuần ảo.

Xét lại ví dụ trên, Ở đây ta xem lại phương thức **Speak()** của lớp **Animal** ta nhận thấy phần định nghĩa của phương thức này chỉ là hình thức sau đó cũng sẽ bị các lớp kế thừa ghi đè lên.

Việc định nghĩa nội dung phương thức không có tác dụng gì vậy tại sao ta lại phải định nghĩa chúng?

Câu trả lời đã được C# giải đáp qua từ khoá **abstract**. Ở đây ta sử dụng **abstract** để nhấn mạnh 2 điều:

- Phương thức **Speak()** có thể ghi đè (**override**).
- Phương thức **Speak()** không có định nghĩa gì bên trong.

Để khai báo lớp trừu tượng và phương thức thuần ảo ta chỉ cần thêm khoá **abstract** vào trước tên lớp và tên phương thức.

```
// PHƯƠNG THỨC THUẦN ẢO ABSTRACT
4 references
abstract class Animal_2
{
    /*
    Khai báo phương thức thuần ảo nên không cần định nghĩa nội dung cho
    phương thức
    */
    4 references
    public abstract void Speak();
}

1 reference
class Cat_2 : Animal_2
{
    4 references
    public override void Speak()
    {
        Console.WriteLine(" Cat is speaking. . .");
    }
}

1 reference
class Dog_2 : Animal_2
{
    4 references
    public override void Speak()
    {
        Console.WriteLine(" Dog is speaking. . .");
    }
}
```

Khi chạy chương trình mọi thứ vẫn ra đúng như mong muốn.

Lưu ý:

- Khi kế thừa 1 lớp trừu tượng bạn bắt buộc phải **override** tất cả các phương thức thuần ảo nhằm đảm bảo tính hợp lệ cho chương trình.

IV. INTERFACE TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG.

❖ Interface là gì? Tại sao lại sử dụng interface.

Interface (nhiều tài liệu gọi là giao diện hoặc lớp giao tiếp) là 1 tập các thành phần chỉ có khai báo mà không có phần định nghĩa (giống phương thức thuần ảo đã trình bày bên trên).

Một interface được hiểu như là 1 khuôn mẫu mà mọi lớp thực thi nó đều phải tuân theo. Interface sẽ định nghĩa phần “**làm gì**” (khai báo) và những lớp thực thi interface này sẽ định nghĩa phần “**làm như thế nào**” (định nghĩa nội dung) tương ứng.

❖ Đặc điểm của interface.

- Chỉ chứa khai báo không chứa phần định nghĩa (giống phương thức thuần ảo). Mặc dù giống phương thức thuần ảo nhưng bạn không cần phải khai báo từ khoá **abstract**.
- Việc ghi đè 1 thành phần trong **interface** cũng không cần từ khoá **override**.
- Không thể khai báo phạm vi truy cập cho các thành phần bên trong **interface**. Các thành phần này sẽ mặc định là **public**.

- **Interface** không chứa các thuộc tính (các biến) dù là hằng số hay biến tĩnh vẫn không được.
- **Interface** không có **constructor** cũng không có **destructor**.
- Các lớp có thể thực thi nhiều **interface** cùng lúc (ở 1 góc độ nào đó có thể nó là *phương án thay thế đa kế thừa*).
- Một **interface** có thể kế thừa nhiều **interface** khác nhưng không thể kế thừa bất kỳ lớp nào.

❖ Mục đích sử dụng interface

- Vì C# không hỗ trợ đa kế thừa nên **interface** ra đời như là 1 giải pháp cho việc đa kế thừa này.
- Trong 1 hệ thống việc trao đổi thông tin giữa các thành phần cần được đồng bộ và có những thống nhất chung. Vì thế dùng **interface** sẽ giúp đưa ra những quy tắc chung mà bắt buộc các thành phần trong hệ thống này phải làm theo mới có thể trao đổi với nhau được.

❖ Khai báo và sử dụng interface

➤ Khai báo:

```
interface <tên interface>
{
    // Khai báo các thành phần bên trong interface
}
```

- **Interface** là từ khoá dùng để khai báo 1 **interface**
- **<tên interface>** là tên do người dùng đặt và tuân theo các quy tắc đặt tên
- để tránh nhầm lẫn với lớp kế thừa thì khi đặt tên **interface** người ta thường thêm tiền tố “I” để nhận dạng

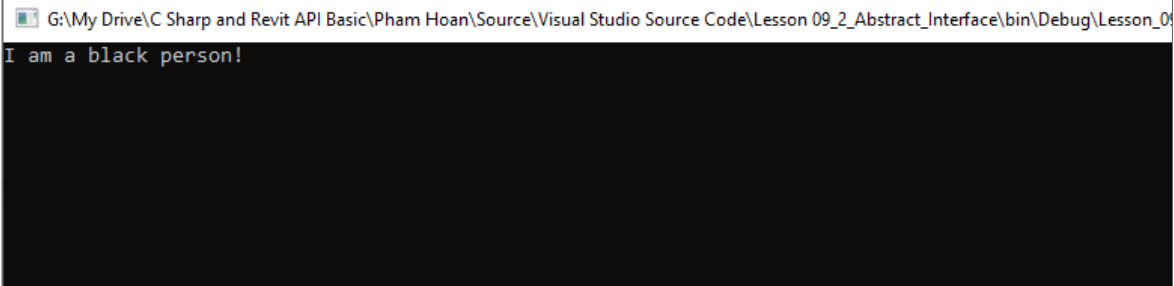
❖ Sử dụng interface:

```
// INTERFACE

1 reference
interface IPerson
{
    //Không chứa pros
    1 reference
    void Speak();// mặc định là public
}

1 reference
class BlackPeople : IPerson
{
    1 reference
    public void Speak() // mặc định là public từ Interface
    {
        Console.WriteLine("I am a black person!");
    }
}
```

```
0 references
static void Main(string[] args)
{
    IPerson black = new BlackPeople();
    black.Speak();
    Console.ReadKey();
}
```



The screenshot shows a Visual Studio console window with the following text:

```
G:\My Drive\C Sharp and Revit API Basic\Pham Hoan\Source\Visual Studio Source Code\Lesson 09_2_Abstract_Interface\bin\Debug\Lesson_09_2_Abstract_Interface.exe
I am a black person!
```

Vì việc thực thi **interface** rất giống với kế thừa nên ta hoàn toàn có thể sử dụng câu lệnh sau:

Việc thiết kế, sử dụng **interface** và **abstract class** chính là cách thể hiện tính trừu tượng trong lập trình hướng đối tượng.

Lưu ý: bạn phải định nghĩa nội dung cho tất cả thành phần trong **interface**.

❖ So sánh giữa interface và lớp trừu tượng

Những điểm giống nhau giữa **interface** và **abstract class**:

- Đều có thể chứa phương thức thuần ảo.
- Đều không thể khởi tạo đối tượng.
- Những điểm khác nhau:

Interface	Abstract class
Chỉ có thể kế thừa nhiều interface khác.	Có thể kế thừa từ 1 lớp và nhiều interface .
Chỉ chứa các khai báo và không có phần nội dung. Không thể chứa biến.	Có thể chứa các thuộc tính (biến) và các phương thức bình thường bên trong.
Không cần dùng từ khoá override để ghi đè.	Sử dụng từ khoá override để ghi đè.
Không có constructor và cũng không có destructor.	Có constructor và destructor.
Không có phạm vi truy cập. Mặc định luôn là public .	Có thể khai báo phạm vi truy cập.
Dùng để định nghĩa 1 khuôn mẫu hoặc quy tắc chung.	Dùng để định nghĩa cốt lõi của lớp, thành phần chung của lớp và sử dụng cho nhiều đối tượng cùng kiểu.
Cần thời gian để tìm phương thức thực tế tương ứng với lớp dẫn đến thời gian chậm hơn 1 chút.	Nhanh hơn so với interface .
Khi ta thêm mới 1 khai báo. Ta phải tìm hết tất cả những lớp có thực thi interface này để định nghĩa nội dung cho phương thức mới.	Đối với abstract class , khi định nghĩa 1 phương thức mới ta hoàn toàn có thể định nghĩa nội dung phương thức là rỗng hoặc những thực thi mặc định nào đó. Vì thế toàn bộ hệ thống vẫn chạy bình thường.

BÀI TẬP

Đề bài: Bài Tập quản lý bán vé máy bay

Xây dựng lớp **Vemaybay** gồm:

- Thuộc tính: **tenchuyen**, **ngaybay**, **giave**
- Phương thức:
 - Nhập
 - Xuất
 - **getgiave()** : hàm trả về giá vé

Xây dựng lớp **Nguoi** gồm:

- Thuộc tính: **hoten**, **gioitinh**, **tuoi**
- Phương thức:
 - Nhập
 - Xuất

Xây dựng lớp **Hanhkhach** (mỗi hành khách được mua nhiều vé) kế thừa lớp **Nguoi** bổ sung thêm:

- Thuộc tính: **Vemaybay *ve**; **int soluong**;
- Phương thức:
 - Nhập
 - Xuất
 - **tongtien()**: trả về Tổng số tiền phải trả của hành khách

Bài toán giả sử có 5 loại vé xuất phát từ HN tới HP (Hải Phòng), DN (Đà Nẵng), NT (Nha Trang), QN (Quy Nhơn), HCM (Hồ Chí Minh). Giá vé lần lượt là 50\$, 80\$, 120\$, 150\$, 250\$.

Gia sử có 10 Hành khách (A, B, C... ~ K), mỗi người mua ngẫu nhiên từ 2- 10 vé, số loại vé cũng ngẫu nhiên.

Hiện thị danh sách hành khách và số tiền phải trả tương ứng của mỗi khách hàng.

Sắp xếp danh sách hành khách theo chiều giảm dần của Tổng tiền.