



# LẬP TRÌNH C# 1

## BÀI 6: INTERFACE VÀ ĐA HÌNH

- ⊙ Đa kế thừa
- ⊙ Đa hình



## Phần I: Đa kế thừa

 Interface

 Kế thừa interface

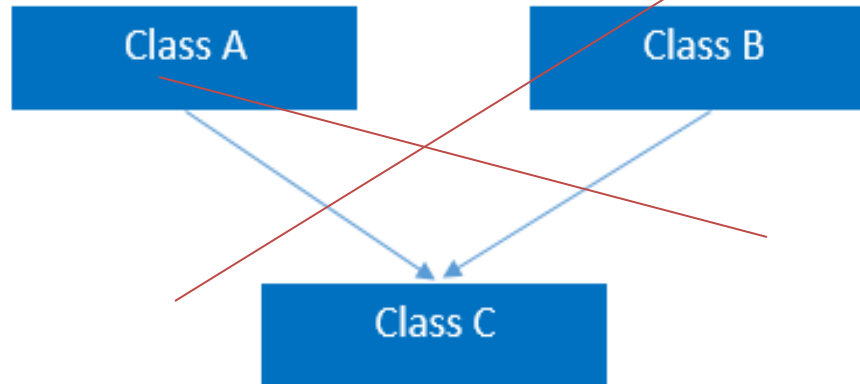
## Phần II: Đa hình (Polymorphism)

 Đa hình khi chạy

 Đa hình khi biên dịch



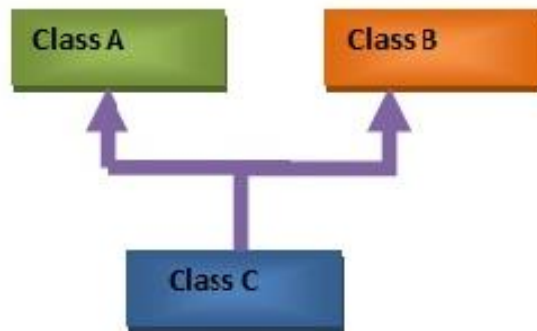
## ❑ C# không hỗ trợ đa thừa kế



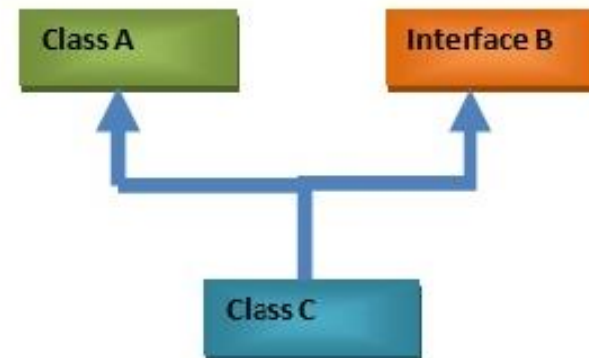
- ❑ Làm thế nào để một lớp có thể sử dụng chung được các thuộc tính và phương thức của 2 hay nhiều lớp khác?
- ❑ Làm thế nào để áp dụng các quy tắc khác nhau, tùy vào hoàn cảnh khác nhau của từng đối tượng?

- ❑ C# hỗ trợ sử dụng interface giải quyết nhiều vấn đề.
- ❑ Interface được xem như là một lớp, lớp đó có thể được một class hoặc struct khác implement nó

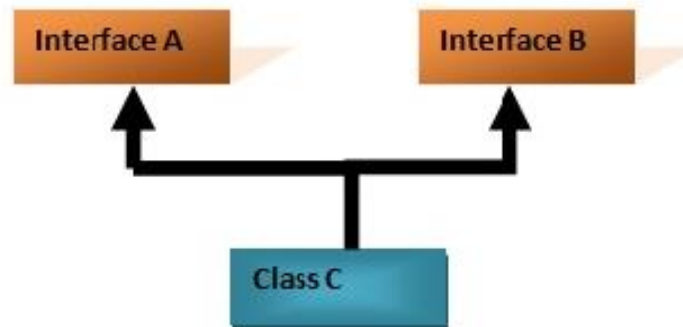
Not Allowed



Allowed



Allowed



## □ Cú pháp khai báo interface

```
<access_modifier> interface <tên_interface> <: tên_base_interface>
{

    // interface member

}
```

- **<access\_modifier>** : là public hoặc internal, nếu không ghi rõ mặc định là internal.
- **interface** : từ khóa khai báo một interface.
- **<tên\_interface>** : thường bắt đầu bằng chữ I, ví dụ: IShape, IAnimal, IStudent,...
- **<: tên\_base\_interface>** : trường hợp có implement từ interface khác thì dấu 2 chấm : biểu thị sự implement, tiếp theo sau là tên base interface.

```
// Khai báo interface IPeople
interface IPeople
{
    // interface member
}

// Khai báo interface IStudent, implement từ interface IPeople
public interface IStudent : IPeople
{
    // interface member
}
```

## ❑ Các lưu ý khi khai báo interface

- ❖ Khai báo interface chỉ chứa các khai báo của các non-static function member (phương thức (methods), thuộc tính (properties), sự kiện (events), chỉ mục (indexers))
- ❖ Chỉ khai báo các thành phần, không khai báo code thực thi, không định nghĩa nội dung code
- ❖ Các thành phần không được sử dụng trong interface là: constructor, destructor, field, hằng, thành phần static.
- ❖ Không thể khai báo hay chỉ định phạm vi truy cập (access modifiers) cho các thành phần bên trong interface. Các thành phần này sẽ mặc định là **public**

- ❑ Khai báo một interface IDung, bao gồm các thành phần ĐÚNG:

```
interface IDung
{
    // method
    // references
    void xyz();

    // property
    // references
    string name { get; set; }

    // indexer
    // references
    double this[int index] { get; set; }

    // event
    event EventHandler OnChanged;
}
```



## ❑ Khai báo một interface ISai, bao gồm các thành phần SAI:

```
interface ISai
{
    // Không được sử dụng hàm [constructor]
    // error CS0526: Interfaces cannot contain constructors
    // References
    ISai() { }

    // Không được sử dụng hàm [destructor]
    // error CS0575: Only class types can contain destructors
    // References
    ~ISai() { }

    // Không được chỉ định phạm vi truy cập (access modifiers)
    // error CS0106: The modifier 'public' is not valid for this item
    // References
    public void xyz();
    // error CS0106: The modifier 'protected' is not valid for this item
    // References
    protected void abc();

    // Không được khai báo [field]
    // error CS0525: Interfaces cannot contain fields
    int number;

    // Không được khai báo hằng [const]
    // error CS0525: Interfaces cannot contain fields
    public const double PI = 3.14;

    // Không được định nghĩa hàm
    // error CS0531: 'ISai.xyz()': interface members cannot have a definition
    // References
    void xyz()
    {
        Console.WriteLine("Print xyz");
    }

    // Không được sử dụng thành phần [static]
    // error CS0106: The modifier 'static' is not valid for this item
    // References
    static void xyz();
}
```

## ❑ Thực thi interface

- ❖ Chỉ có các class/ struct có thể thực thi interface
- ❖ Khi thực thi interface, các class/ struct phải:
  - Thêm khai báo interface vào khai báo class/ struct
  - Thực thi tất cả các thành phần của interface trong class/ struct

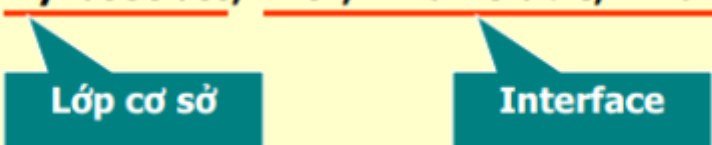
```
interface MyInterface1{  
    int DoStuff(int nVar1 int nVar2);  
    double DoOtherStuff(string s,long x);  
}
```

```
class MyClass : MyInterface1  
{  
    int DoStuff(int nVar1, long nVar2){  
        // implement code  
    }  
    double DoOtherStuff(string s, long x){  
        // implement code  
    }  
}
```


## ❑ Thực thi interface

- ❖ Nếu một lớp thực thi interface, nó phải thực thi hết tất cả các thành phần của interface
- ❖ Nếu lớp được dẫn xuất từ lớp cơ sở và có thực thi interface : tên lớp cơ sở phải đặt trước các tên interface trong khai báo lớp

```
class Derived: MyBaseClass, IIfc1, IEnumerable, IEnumerator
{
    ...
}
```



- ❖ Mỗi **interface** không được phép kế thừa từ một class nào cả
- ❖ Không thể tạo ra một đối tượng từ interface



Xây dựng 1 interface Iperson có 1 phương  
thức: void show(),

- 1 class person inherit từ Iperson có:

- + 2 property: name, address

- + phương thức: show( show ra name và  
address)

- 1 lớp student kế thừa từ class person

- + có thuộc tính course

- + phương thức show ra course

DEMO



❑ Interface là một kiểu tham chiếu (reference type)

- ❖ Không thể truy xuất interface trực tiếp thông qua các thành phần của đối tượng thuộc lớp
- ❖ Có thể tham chiếu đến interface thông qua ép kiểu của đối tượng thuộc lớp sang kiểu interface

```

interface IIfc1{
    void PrintOut(string s);
}
class MyClass: IIfc1{
    public void PrintOut(string s){
        Console.WriteLine("Calling through: {0}",s);
    }
}
...
MyClass mc=new MyClass();
mc.PrintOut("Object");
IIfc1 ifc=(IIfc1)mc;
ifc.PrintOut("Interface");
...
  
```

Calling through: Object  
Calling through: Interface

Ép kiểu

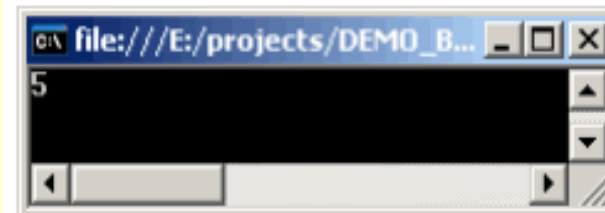
Toán tử .

## ❑ Kế thừa Interface

- ❖ Một interface có thể được kế thừa từ 1/nhiều interface
- ❖ Chỉ định 1 interface kế thừa từ các interface khác:
  - Thêm danh sách các interface kế thừa vào khai báo interface
  - Các interface cách nhau bởi dấu phẩy (,)

```
interface IDataRetrieve{
    int GetData();
}
interface IDataStore{
    void SetData(int s);
}
interface IDataIO: IDataRetrieve, IDataStore{
}
class MyData: IDataIO{
    int nPrivateData;
    public int GetData(){
        return nPrivateData;
    }
    public void SetData(int x){
        nPrivateData=x;
    }
}
```

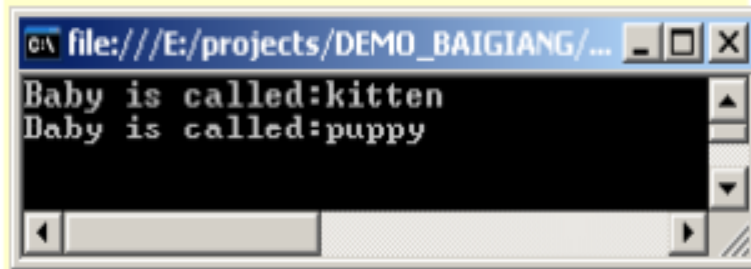
```
class Program{
    static void Main(){
        MyData data=new MyData();
        data.SetData(5);
        Console.WriteLine("{0}",
            data.GetData());
    }
}
```



## ❑ Ví dụ nhiều lớp thực thi interface

```
interface ILiveBirth{
    string BabyCalled();
}
class Animal {
}
class Cat : Animal, ILiveBirth{
    string ILiveBirth.BabyCalled(){
        return "kitten";
    }
}
class Dog : Animal, ILiveBirth {
    string ILiveBirth.BabyCalled(){
        return "puppy";
    }
}
class Bird : Animal{
}
```

```
Animal[] animalArray = new Animal[3];
animalArray[0] = new Cat();
animalArray[1] = new Bird();
animalArray[2] = new Dog();
foreach( Animal a in animalArray ){
    ILiveBirth b = a as ILiveBirth;
    if (b != null)
        Console.WriteLine("Baby is called:
        {0}", b.BabyCalled());
}
```





# DEMO

Hiện thực hóa các ví dụ





# SỰ GIỐNG NHAU CỦA ABSTRACT CLASS VÀ INTERFACE

- ❑ Abstract class và interface đều không thể khởi tạo đối tượng từ chính nó được
- ❑ Abstract class và interface đều có chứa *abstract method*.
- ❑ Abstract class và interface đều được kế thừa hoặc thực thi phương thức hoặc properties từ các class dẫn xuất nó.
- ❑ Abstract class và interface đều có thể implement từ một hoặc nhiều interface
- ❑ Abstract class và interface đều giúp cho code trở nên sáng sủa và gọn gàng, dễ bảo trì và nâng cấp

# SỰ KHÁC NHAU GIỮA ABSTRACT CLASS VÀ INTERFACE

Abstract class	Interface
<ul style="list-style-type: none"> <li>Có thể tạo được đối tượng thông qua lớp dẫn xuất. (Xem lại bài: <a href="#">Lớp trừu tượng Abstract class trong C#</a>)</li> </ul>	<ul style="list-style-type: none"> <li>Không thể tạo được đối tượng từ nó hoặc từ lớp dẫn xuất.</li> </ul>
<ul style="list-style-type: none"> <li>Vừa có thể kế thừa class, vừa có thể hiện thực interface. (Trong trường hợp này, tên base class phải để đầu tiên trong danh sách, tiếp theo là tên của các interface)</li> </ul>	<ul style="list-style-type: none"> <li>Không thể kế thừa class, chỉ có thể hiện thực interface.</li> </ul>
<ul style="list-style-type: none"> <li>Có thể khai báo field, const, constructor, destructor.</li> </ul>	<ul style="list-style-type: none"> <li>Không được khai báo field, const, constructor, destructor.</li> </ul>
<ul style="list-style-type: none"> <li>Có thể chứa phương thức đã định nghĩa hoàn chỉnh.</li> </ul>	<ul style="list-style-type: none"> <li>Không được chứa phương thức định nghĩa, chỉ khai báo prototype(nguyên mẫu hàm) của phương thức.</li> </ul>
<ul style="list-style-type: none"> <li>Ở lớp dẫn xuất khi kế thừa phải dùng từ khóa <i>override</i> lúc định nghĩa.</li> </ul>	<ul style="list-style-type: none"> <li>Không dùng từ khóa <i>override</i> khi implement (vi định nghĩa mới hoàn toàn).</li> </ul>
<ul style="list-style-type: none"> <li>Được phép chỉ định access modifiers cho phương thức, thuộc tính... như public, protected, private. Đối với các methods hay properties trừu tượng bắt buộc khai báo với từ khóa <b>abstract</b> và access modifiers bắt buộc là <b>public</b> hoặc <b>protected</b>.</li> </ul>	<ul style="list-style-type: none"> <li>Không được phép chỉ định access modifiers cho phương thức, thuộc tính. Tất cả giả định là <b>public</b> và không thể thay đổi thành access modifiers khác.</li> </ul>
<ul style="list-style-type: none"> <li>Chỉ dùng để định nghĩa cốt lõi của một lớp và chỉ sử dụng cho những đối tượng có cùng bản chất. Lập trình một lớp.</li> </ul>	<ul style="list-style-type: none"> <li>Dùng để định nghĩa những khả năng bên ngoài của một lớp. Tức là những đối tượng không cùng bản chất với interface, cũng implement được.</li> </ul>



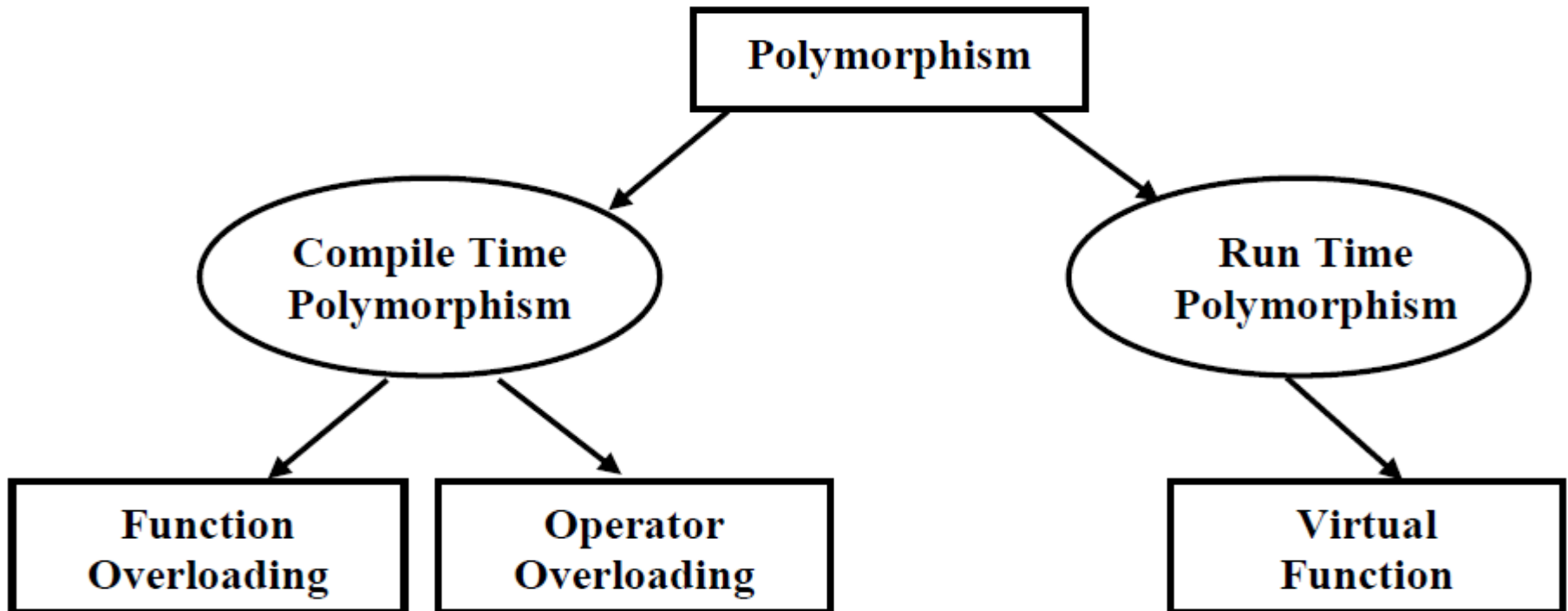
# LẬP TRÌNH C# 1

## BÀI 6: INTERFACE VÀ ĐA HÌNH(P2)

- ❑ 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
- ❑ Cho phép **một thao tác có các cách xử lý khác nhau** trên các đối tượng khác nhau



□ Tính đa hình được phân loại:



## ❑ Đa hình khi chạy (Run-time Polymorphism) – Override.

- ❖ 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

## ❑ Đa hình khi chạy (Run-time Polymorphism) – Override.

```

class Tau
{
    2 references
    public virtual void LayThongTin()
    {
        Console.WriteLine("Đây là chiếc Tàu.");
    }
}
0 references
class TauChien : Tau
{
    2 references
    public override void LayThongTin()
    {
        Console.WriteLine("Đây là Tàu chiến.");
    }
}
0 references
class TauDuLich : Tau
{
    2 references
    public override void LayThongTin()
    {
        Console.WriteLine("Đây là Tàu du lịch.");
    }
}
  
```

```

static void Main(string[] args)
{
    // Khai báo 3 đối tượng tàu
    Tau tau1 = new Tau();
    Tau tau2 = new TauChien();
    Tau tau3 = new TauDuLich();

    // Gọi phương thức lấy thông tin
    tau1.LayThongTin();
    tau2.LayThongTin();
    tau3.LayThongTin();
}
  
```

## ❑ Đa hình khi biên dịch (Compile-time Polymorphism)

❖ Tính đa hình khi biên dịch thể hiện ở sự đa dạng nhờ sự khác biệt :

- Số lượng tham số
- Kiểu dữ liệu của tham số

```
public class Calculate
{
    public void AddNumbers(int a, int b)
    {
        Console.WriteLine("a + b = {0}", a + b);
    }
    public void AddNumbers(int a, int b, int c)
    {
        Console.WriteLine("a + b + c = {0}", a + b + c);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Calculate c = new Calculate();
        c.AddNumbers(1, 2);
        c.AddNumbers(1, 2, 3);
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
}
```





# DEMO

Hiện thực hóa các ví dụ đa hình



# Tổng kết bài học

## Phần I: Đa kế thừa

 Interface

 Kế thừa interface

## Phần II: Đa hình (Polymorphism)

 Đa hình khi chạy

 Đa hình khi biên dịch





# KẾT THÚC

Lập trình C#1