



Chương 2: LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VỚI C#

Giảng viên: Vũ Minh Sang
Phòng: E9.4
E-mail: sangvm@uit.edu.vn

Nội dung



Lớp và đối tượng



Kế thừa (Inheritance)



Đa hình (Polymorphism)



Lớp trừu tượng (Abstract class)



Giao diện (Interface)



Lớp và đối tượng

Lớp (Class)

- 📖 Là khuôn mẫu (template), là bản mô tả những tính chất và hành vi chung của những sự vật, hiện tượng tồn tại trong thực tế.
- 📖 Là sự trừu tượng hóa tập các đối tượng cùng loại về hai khía cạnh: thông tin mô tả đối tượng (tính chất) và những hành vi (hoạt động) trên các thông tin đó.
- 📖 VD: các lớp: Con người, Sinh viên, Lớp học, Môn học, Phòng học....

Đối tượng (Object)

- 📖 Thể hiện cụ thể của một lớp được gọi là đối tượng (Object).
- 📖 Một lớp có thể có nhiều thể hiện cụ thể (nhiều đối tượng)
- 📖 VD: Sinh viên A, Sinh viên B là đối tượng của lớp Sinh viên.
- 📖 Quan hệ giữa Class và Object là quan hệ giữa mô tả (khuôn mẫu) và sự vật, hiện tượng cụ thể được sinh ra từ mô tả đó.

Class trong C#

- 📖 Là các khối xây dựng cơ sở của ứng dụng, là trung tâm của ngôn ngữ lập trình C#
- 📖 Là một kiểu dữ liệu để chứa định nghĩa những thuộc tính và phương thức (hành vi) dùng để mô tả một nhóm những đối tượng cùng loại.
- 📖 Mỗi class có thể có các thành phần:
 - **Biến thành viên (field):** lưu trữ thông tin mô tả đối tượng hoặc trạng thái của đối tượng.
 - **Đặc tính (property):** dùng lưu trữ tương tự biến thành viên nhưng có khả năng kiểm soát dữ liệu nhập xuất.
 - **Phương thức (method):** dùng cập nhật, tính toán, cung cấp và xử lý thông tin.
 - **Sự kiện (delegate/event):** gửi thông báo về sự thay đổi trạng thái của đối tượng ra bên ngoài.
 - Class có thể chứa định nghĩa kiểu dữ liệu khác (class, struct, interface, delegate, enum) gọi là kiểu thành viên (member /inner/nested type).
- 📖 Class là một kiểu dữ liệu nên object của class chính là biến thuộc kiểu dữ liệu đó.

Class trong C#

Khai báo Class

```
[public|internal] class <tên class>{  
    [Các thành phần trong class]  
}
```

- **public hoặc internal:** là các từ khóa được dùng để quyết định phạm vi truy cập của class (access modifier), cho phép class được sử dụng nội bộ trong phạm vi của project (internal) hoặc có thể được sử dụng bởi các project khác (public). Mặc định là internal.
- **class:** từ khóa bắt buộc, được dùng để khai báo class.

Class trong C#

- **tên class:** định danh của class và được đặt tuân theo quy tắc:
 - Bắt đầu bằng chữ cái, ký tự gạch chân “_”, hoặc ký tự “@”.
 - Chỉ được chứa chữ cái, chữ số và ký tự gạch chân.
 - Không được trùng với các từ khóa của C#, muốn đặt trùng thì đặt ký tự @ phía trước.
 - Độ dài tên class không giới hạn và có thể có ký tự Unicode (có thể đặt tên bằng tiếng Việt có dấu).
 - Phân biệt chữ hoa và chữ thường (case-sensitive).

Một số **quy ước** khi đặt tên:

- Mỗi file mã nguồn chỉ nên chứa 1 class và tên class trùng tên file.
- Tên class bắt đầu bằng chữ viết hoa.
- Nếu tên class gồm nhiều từ ghép lại thì đặt tên theo kiểu PascalCase, tức là viết hoa tất cả các chữ cái đầu. VD: MyClass.

Thành phần trong Class

- 📖 Biến thành viên (member variable)
- 📖 Phương thức (method)
- 📖 Đặc tính (property)
- 📖 Hàm tạo/hàm dựng (constructor)
- 📖 Các loại tĩnh (static)
- 📖 Nạp chồng toán tử (operators)

Biến thành viên

- 📖 Còn được gọi là **Field**
- 📖 Là thành phần chứa dữ liệu.
- 📖 Khai báo trực tiếp trong thân class.
- 📖 Được sử dụng bởi bất kỳ thành phần khác trong class.
- 📖 Có thể được sử dụng bởi thành phần ngoài class.
- 📖 Cú pháp khai báo biến:

[**public|private|protected**] <kiểu dữ liệu> <tên biến> [= <giá trị>];

Public, private, protected: xác định phạm vi truy cập vào biến

- **Public:** biến có thể được truy xuất ngoài class
- **Private (mặc định):** biến chỉ được sử dụng hoàn toàn trong class, không được truy xuất ngoài class kể cả kế thừa.
- **Protected:** truy cập giới hạn trong class, và class thừa kế từ class này

Biến thành viên

Kiểu dữ liệu: là các kiểu dữ liệu do C# định nghĩa (xem lại bài C# căn bản) hoặc do người dùng định nghĩa: mảng (array), bản ghi (struct), liệt kê (enum) hoặc lớp (class).

Tên biến: định danh của biến, được đặt theo quy tắc đặt tên class và theo một số quy ước:

- Biến public và protected thường bắt đầu bằng chữ cái in hoa và theo cách viết PascalCase
- Biến private thường bắt đầu bằng ký tự gạch chân và theo cách viết camelCase, tức là từ đầu tiên viết thường, viết hoa chữ cái đầu mỗi từ tiếp theo. VD: myVar

Có thể gán giá trị ban đầu cho biến hoặc không cần (C# sẽ tự gán giá trị mặc định cho biến).

📖 VD: `string _myVar; public int Age = 0;`

Biến thành viên

📖 Một số lưu ý khi dùng biến thành viên:

- Trong lập trình hướng đối tượng, hạn chế dùng biến kiểu public; C# cung cấp property để thay thế biến public.
- Thường khai báo biến kiểu private hoặc protected, và viết phương thức nhập xuất dữ liệu cho biến (hàm getter/setter), để giúp kiểm soát giá trị nhập xuất của biến.

📖 Biến thành viên chỉ đọc: cho phép gán giá trị vào biến lúc khai báo hoặc sau khi khởi tạo và sẽ không thay đổi được giá trị nữa. Sử dụng từ khóa readonly.

VD: `public readonly double discount = 0.2;`

Phương thức (Method)

☞ Phương thức (method): là thành phần chứa các lệnh để thực thi một nhiệm vụ, hành động hoặc chức năng cụ thể.

☞ Có 2 giai đoạn khi làm việc với phương thức:

- *Khai báo (định nghĩa) phương thức*: mô tả thông tin bắt buộc của phương thức và các câu lệnh (code) trong thân phương thức để thực thi chức năng.
- *Sử dụng (gọi) phương thức*: là giai đoạn cung cấp dữ liệu thực sự để thực thi phương thức.

☞ Cú pháp khai báo phương thức:

```
[public|private|protected] <kiểu dữ liệu trả về> <tên phương  
thức>([danh sách tham số])  
{  
    [Thân phương thức]  
    [return [giá trị];]  
}
```

Phương thức (Method)

Public, private, protected: ý nghĩa tương tự trong biến thành viên.

Mặc định là private

Tên phương thức: do người dùng đặt, giống quy tắc đặt tên class, và thường đặt theo quy ước PascalCase.

Kiểu dữ liệu trả về: là kết quả nhận được sau khi kết thúc phương thức.

- Là bất cứ kiểu dữ liệu do C# hoặc do người dùng định nghĩa.
- Nếu phương thức không trả về kết quả nào thì dùng từ khóa **void**.
- Khi có kiểu trả về khác void thì khi kết thúc phương thức phải dùng lệnh return để trả giá trị về.

Danh sách tham số: là danh sách biến có thể sử dụng trong phương thức. Được định nghĩa như sau:

(<kiểu dữ liệu><tham số 1>,<kiểu dữ liệu> <tham số 2>,...)

VD: `public void MyMethod(int a) {...}`

Phương thức (Method)

- *Kiểu giá trị (value type)*: giá trị của biến tham số truyền vào phương thức vẫn giữ nguyên sau khi phương thức được thực thi xong.
- *Kiểu tham chiếu (reference type)*: giá trị của biến tham số truyền vào phương thức sẽ được thay đổi sau khi kết thúc phương thức. Dùng từ khóa **ref** để khai báo kiểu biến tham chiếu.

```
public void ChangeColor(ref string color){}
```

- Khi gọi phương thức có tham số kiểu tham chiếu thì cũng phải dùng **ref** cho biến đó.

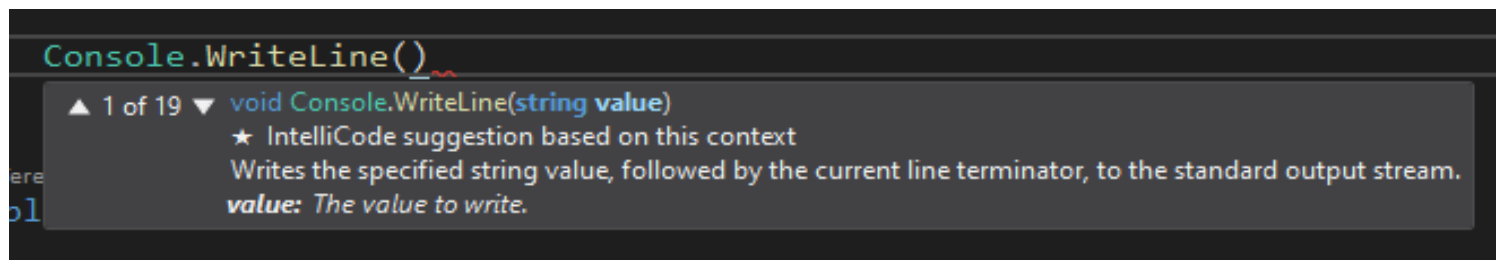
```
test.ChangeColor(ref color);
```

- *Tham số params*: cho phép khai báo phương thức với số lượng không hạn chế tham số truyền vào.

```
VD: public int Tong(params int[] so) {...}  
    int kq = Tong(1, 2, 3);
```

Phương thức (Method)

Nạp chồng phương thức (method overloading): trong một class có thể có nhiều phương thức trùng tên. Phương thức overload có danh sách tham số khác nhau. VD: trong phương thức *WriteLine* của lớp *Console* có 19 overload



Phương thức với Expression body: thân phương thức chỉ có một câu lệnh. *Cách viết:* `<tên phương thức>() => <lệnh>;`

```
public bool IsCheck(int a, int b) => a >= b;
```

Cách thông thường:

```
public bool IsCheck(int a, int b){  
    return a >= b;  
}
```


Phương thức (Method)

Named Arguments: Cho phép gọi phương thức và truyền giá trị tham số bằng cách viết tên tham số và giá trị truyền vào; vì vậy không cần quan tâm đến thứ tự viết tham số.

```
//Khai báo
```

```
public void In(string ma, string hoten, float luong) { }
```

```
//Gọi phương thức
```

```
test.In(ma:"1234", hoten: "ABC", luong: 50.0);
```

Đặc tính (property)

- 📖 **Đặc tính (property):** cho phép lưu trữ và truy xuất trực tiếp dữ liệu.
- 📖 Kiểm soát được việc nhập xuất dữ liệu vào biến, hoặc kiểm soát riêng lẻ từng chiều truy xuất.
- 📖 Thay thế cho biến kiểu public.
- 📖 Thay cho cặp phương thức setter/getter
- 📖 VD:

//Khai báo

```
private string _hoten;  
public string HoTen {  
    get { return _hoten; }  
    set { _hoten = value; }  
}
```

//Cách sử dụng

```
test.HoTen = "ABC";
```

Đặc tính (property)

Trong đó: biến *_hoten* có tên gọi là backed field cho property *HoTen*, là nơi lưu trữ dữ liệu thực sự. Còn property *HoTen* kiểm soát việc nhập xuất dữ liệu cho biến *_hoten*.

VD2:

```
//Khai báo
private string _mssv;
public string MSSV {
    get { return _mssv; }
    protected set { _mssv = value; }
}
```

```
//Lệnh này lỗi vì setter
//của MSSV là protected
//không gán trị ngoài class
test.MSSV = "1234";
```

Đặc tính (property)

Auto-property: nếu không cần kiểm soát việc nhập xuất giá trị vào biến thì có thể sử dụng auto-property. Khi khai báo kết hợp gán giá trị cho biến.

```
[public|protected|private] <kiểu dữ liệu> <tên thuộc tính>
{
    [public|protected|private] get;
    [public|protected|private] set;
} [= <giá trị>];
```

- Tên thuộc tính: theo quy tắc đặt định danh và quy ước sử dụng PascalCase
- Hai phương thức get/set được gọi là accessor hoặc getter, setter.
- Với từ khóa xác định phạm vi truy cập giúp property thành các loại:
 - Chỉ đọc (read-only): public get, protected/private set.
 - Chỉ gán (assign-only): protected/private get, public set.
 - Truy cập tự do (full access): public get, public set (mặc định).

Đặc tính (property)

VD:

```
public string MSSV { get; protected set; } = "0000";  
public string HoTen { get; set; } = "ABC";
```

Full property: khi khai báo đi kèm với backed field (nơi lưu trữ thông tin).

```
[private|protected] <kiểu dữ liệu> <tên biến>;  
[public|protected|private] <kiểu dữ liệu> <tên thuộc tính>  
{  
    [public|protected|private] get { [thân phương thức];  
    return <tên biến>;}  
    [public|protected|private] set { [thân phương thức];  
    <tên biến> = value;  
}
```

Đặc tính (property)

VD:

```
private string _hoten;  
public string HoTen  
{  
    get { return _hoten; }  
    set { if (!string.IsNullOrEmpty(value)) { _hoten = value; } }  
}
```

Expression body: nếu thân getter hoặc setter chỉ có 1 lệnh thì có thể sử dụng theo cách sau:

```
private string _hoten;  
public string HoTen  
{  
    get => _hoten; //Tương đương get {return _hoten;}  
    set => _hoten = value; //Tương đương set { _hoten = value; }  
}
```

Hàm tạo/hàm dựng (constructor)

- 📖 **Hàm tạo/hàm dựng (constructor):** là hàm được dùng để khởi tạo đối tượng (object) của class.
- 📖 Sử dụng từ khóa ***new*** để gọi một trong số các hàm tạo của class.
- 📖 Đặc điểm của hàm tạo:
 - Luôn có cùng tên với class, không có kiểu dữ liệu trả về.
 - Danh sách tham số và thân hàm tương tự phương thức trong class
 - Trong hàm tạo thường được dùng để khởi tạo giá trị cho các biến của class.
 - Không giới hạn số lượng hàm tạo trong class và là dạng **nạp chồng phương thức (method overloading)**.

Hàm tạo/hàm dựng (constructor)

VD:

```
class SinhVien
{
    private string _mssv;
    private string _hoten;
    /// <summary>
    /// Đây là hàm tạo của class Sinh viên có
    /// tham số
    /// </summary>
    /// <param name="mssv"></param>
    /// <param name="hoten"></param>
    public SinhVien(string mssv, string hoten)
    {
        _mssv = mssv;
        _hoten = hoten;
    }
}

class Sinh viên
    /// </summary>
    public SinhVien(){
        _mssv = "0000";
        _hoten = "ABC";
    }

class Program
{
    static void Main(string[] args)
    {
        SinhVien sv = new SinhVien();
        SinhVien sv1 = new SinhVien("0000", "ABC");
    }
}
```


Hàm tạo/hàm dựng (constructor)

📖 Khởi tạo đối tượng với property.

- Thêm các property vào class Sinh viên

```
public string MSSV { get => _mssv; set => _mssv = value; }  
public string HoTen { get => _hoten; set => _hoten = value; }
```

- Khởi tạo lớp Sinh viên trong Main

```
SinhVien sv = new SinhVien  
{  
    MSSV = "1234",  
    HoTen = "ABC"  
};
```

- Quy tắc:

- Lệnh gán giá trị cho property đặt trong dấu ngoặc nhọn.
- Các lệnh gán giá trị tách nhau bởi dấu phẩy.
- Kết thúc bằng dấu chấm phẩy.
- Không bắt buộc phải gán giá trị cho các thuộc tính

Các nguyên tắc của hàm tạo

- 📖 Hàm tạo mặc định (default constructor) sẽ tự động được sinh ra bởi trình biên dịch (compiler) C# khi không có hàm tạo nào được định nghĩa trong class. Hàm tạo mặc định không có tham số.
- 📖 Nếu trong class **có viết** một hàm tạo không tham số thì không còn hàm tạo mặc định nữa mà được gọi là hàm tạo không tham số (parameter-less/zero-parameter constructor).
- 📖 Nếu trong class đã định nghĩa một hàm tạo riêng, trình biên dịch sẽ không tự sinh hàm tạo mặc định nữa, khi đó muốn gọi hàm tạo không tham số thì trong class phải có định nghĩa hàm tạo này.
- 📖 Chuỗi hàm khởi tạo: các hàm tạo gọi lẫn nhau

```
public SinhVien(string mssv){  
    _mssv = mssv;  
}  
public SinhVien(string mssv, string hoten): this(mssv){  
    _hoten = hoten;  
}
```

Cấu trúc **:this(...)** được gọi là **constructor initializer**

Các loại tĩnh (static)

- 📖 Biến tĩnh (static field)
- 📖 Phương thức tĩnh (static method).
- 📖 Hàm tạo tĩnh (static constructor).
- 📖 Đặc tính tĩnh (static property).
- 📖 Lớp tĩnh (static class).

Biến tĩnh (static field)

```
class SinhVien
{
    public int Id { get; set; }
    public string HoTen { get; set; }
    public SinhVien(string hoten)
    {
        HoTen = hoten;
    }
}
```

Đặt vấn đề:

- Mỗi khi khởi tạo một đối tượng class SinhVien, đối tượng đó sẽ được gán 1 giá trị Id duy nhất.
- Nếu tạo nhiều đối tượng SinhVien thì giá trị Id sẽ tăng dần.
- Như vậy, sẽ cần tạo một biến đếm độc lập (counter) nằm ngoài class SinhVien để gán giá trị hiện thời của counter cho Id.
- Do đó khi khởi tạo mới một đối tượng SinhVien thì cần truyền biến counter này vào Id.

Biến tĩnh (static field)

Dùng biến static

```
class SinhVien
{
    public int Id { get; set; }
    public string HoTen { get; set; }
    public static int NextId; //Thêm một biến với static
    public SinhVien(string hoten){
        HoTen = hoten;
        Id = NextId;
        NextId++;
    }
}
```

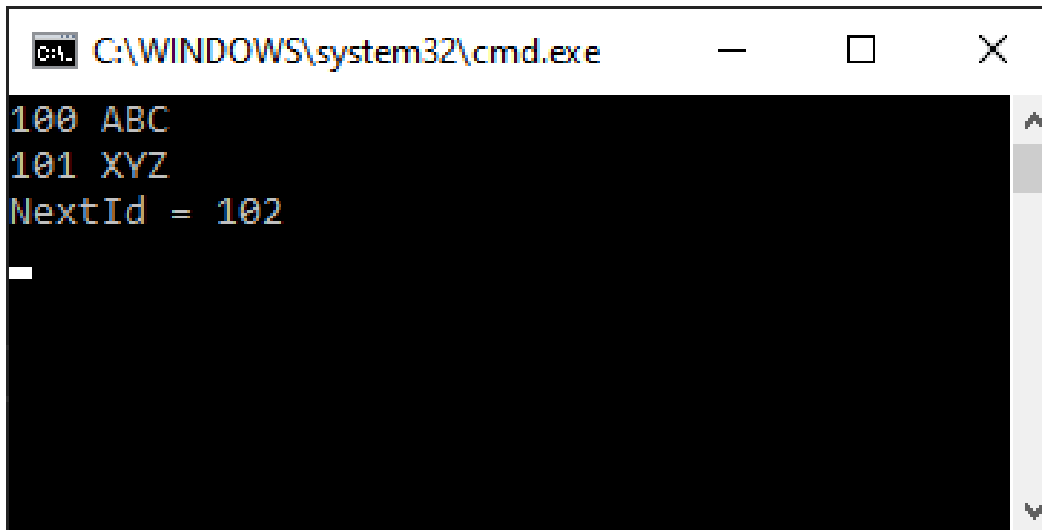
- *NextId* là một biến tĩnh với từ khóa **static**.
- Khi khởi tạo bao nhiêu đối tượng thì đều có chung một biến static này.
- Không cần phải có một biến counter bên ngoài class nữa mà sẽ dùng luôn biến *NextId* này.
- Có thể đếm được có bao nhiêu đối tượng *SinhVien* đã được tạo ra.

Biến tĩnh (static field)

Hàm main

```
static void Main(string[] args)
{
    SinhVien.NextId = 100;
    SinhVien sv1 = new SinhVien("ABC");
    SinhVien sv2 = new SinhVien("XYZ");
    Console.WriteLine("{0} {1}", sv1.Id, sv1.HoTen);
    Console.WriteLine("{0} {1}", sv2.Id, sv2.HoTen);
    Console.WriteLine($"NextId = { SinhVien.NextId}");
    Console.ReadKey();
}
```

Kết quả:



```
C:\WINDOWS\system32\cmd.exe
100 ABC
101 XYZ
NextId = 102
```

Biến tĩnh (static field)

Đặc điểm biến tĩnh

- Được khai báo với từ khóa ***static*** trước tên kiểu dữ liệu.
- Là biến duy nhất trong class (không có biến nào cùng tên với nó)
- Không cần phải khởi tạo đối tượng để truy xuất giá trị.
- Truy xuất trực tiếp bằng tên class: *<tên class>.<tên biến>*
- Khi sử dụng trong nội bộ class thì có thể bỏ qua tên class.
- Biến hằng trong C# sẽ được coi như là biến static. VD:

```
public const int MaxValue = 123;
```

Phương thức tĩnh (static method)

- Được khai báo với từ khóa *static* đặt trước *kiểu dữ liệu trả về*.

```
public static void In() { }
```

- Không cần khởi tạo đối tượng để truy xuất phương thức tĩnh.
- Truy xuất trực tiếp bằng tên class: *<tên class>.<tên phương thức>*
- Không được sử dụng từ khóa *this* bên trong phương thức static.
- Phương thức tĩnh không sử dụng được biến thành viên, phương thức bình thường của class.
- Phương thức bình thường có thể sử dụng được phương thức tĩnh.

Hàm tạo tĩnh (static constructor)

- Được dùng để khởi tạo các giá trị của biến static trong class.

```
static SinhVien(){  
    Random randomGe = new Random();  
    NextId = randomGe.Next(100, 999);  
}
```

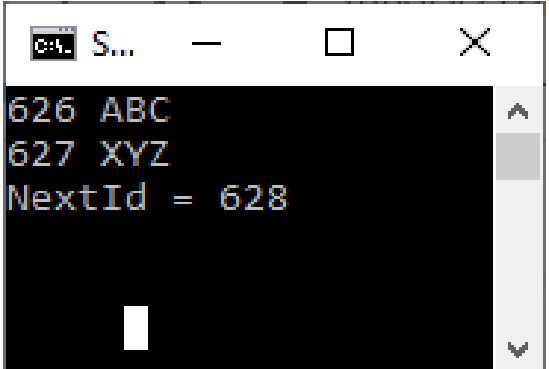
- Hàm tạo tĩnh không cần từ khóa xác định phạm vi truy xuất, vì hàm tạo tĩnh không thể gọi từ class khác.

- Khi class được sử dụng lần đầu hàm tạo tĩnh tự động được gọi

- Không có danh sách tham số trong hàm tạo tĩnh.

- VD:

```
static void Main(string[] args)  
{  
    SinhVien.NextId = 100;  
    SinhVien sv1 = new SinhVien("ABC");  
    SinhVien sv2 = new SinhVien("XYZ");  
    Console.WriteLine("{0} {1}", sv1.Id, sv1.HoTen);  
    Console.WriteLine("{0} {1}", sv2.Id, sv2.HoTen);  
    Console.WriteLine($"NextId = { SinhVien.NextId}");  
    Console.ReadKey();  
}
```



The screenshot shows a console window with the following output:

```
626 ABC  
627 XYZ  
NextId = 628
```

Đặc tính tĩnh (static property)

- Được dùng để kiểm soát việc nhập/xuất giá trị cho biến static.

```
public static int _count;  
public static int Count  
{  
    get { return _count; }  
    set { _count = value; }  
}
```

- Được sử dụng qua tên class giống như biến tĩnh và phương thức tĩnh.
- Dùng thay cho biến static public.

Lớp tĩnh (static class)

- ❏ Là lớp không chứa bất cứ biến hoặc phương thức bình thường nào, chỉ chứa các phương thức tĩnh và biến tĩnh.
- ❏ Lớp tĩnh được khai báo từ khóa static trước class.
- ❏ Không cho phép tạo đối tượng hoặc kế thừa từ lớp tĩnh.
- ❏ Truy xuất biến hoặc phương thức trong lớp tĩnh bằng cách: *<tên class>.<tên biến> và <tên class>.<tên phương thức>*
- ❏ Có thể sử dụng cấu trúc using static để trực tiếp truy xuất các thành phần trong lớp tĩnh mà không cần ghi rõ tên lớp.
- ❏ VD: Lớp Console là một lớp tĩnh.

```
using static System.Console;
class Program{
    static void Main(string[] args){
        WriteLine("Hello world");
        ReadKey();
    }
}
```

Nạp chồng toán tử

- ❏ Các kiểu dữ liệu số trong C# có một số phép toán: phép toán số học, phép so sánh, phép toán tăng giảm hoặc phép toán cộng chuỗi (kiểu string).
- ❏ Nhưng các class do người dùng định nghĩa lại không thể sử dụng ngay được phép toán đó. VD: người dùng định nghĩa kiểu Vector và các phép toán cơ bản không thể thực hiện ngay được trên Vector này: cộng, trừ vector...
- ❏ Nạp chồng toán tử (operator overloading): là việc định nghĩa lại hoạt động của một số phép toán đã có để áp dụng cho các kiểu dữ liệu do người dùng xây dựng.
- ❏ Các toán tử có thể nạp chồng: +, -, *, /, %, !, ~, ++, --, ==, !=, <, >, <=, >=.
- ❏ Cú pháp:
`public static <kiểu trả về> operator <phép toán>(<tham số>){...}`
- ❏ Nạp chồng phương thức và nạp chồng toán tử thuộc về nguyên lý đa hình tĩnh (static polymorphism).

Nạp chồng toán tử

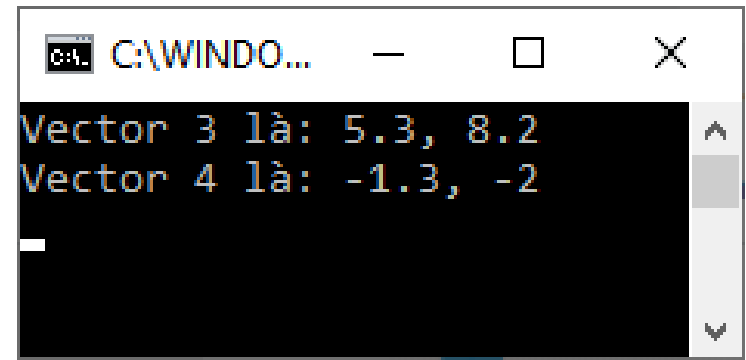
```
class Vector{
    public double X { get; set; }
    public double Y { get; set; }
    public Vector(){}
    public Vector(double x, double y{
        X = x;
        Y = y;
    }
    //Nạp chồng phép cộng 2 vector
    public static Vector operator +(Vector a, Vector b){
        Vector temp = new Vector{
            X = a.X + b.X,
            Y = a.Y + b.Y
        };
        return temp;
    }
    //Nạp chồng phép trừ 2 vector
    public static Vector operator -(Vector a, Vector b){
        Vector temp = new Vector{
            X = a.X - b.X,
            Y = a.Y - b.Y
        };
        return temp;
    }
}
```

Nạp chồng toán tử

```
public override string ToString()
{
    return string.Format("{0}, {1}", X, Y);
}
// Hết class Vector
```

```
//Main
static void Main(string[] args)
{
    Vector vec1 = new Vector(2.0, 3.1);
    Vector vec2 = new Vector(3.3, 5.1);
    Vector vec3 = new Vector();
    Vector vec4 = new Vector();

    vec3 = vec1 + vec2;
    WriteLine("Vector 3 là: {0}", vec3.ToString());
    vec4 = vec1 - vec2;
    WriteLine("Vector 4 là: {0}", vec4.ToString());
    ReadKey();
}
```



A screenshot of a Windows command prompt window. The title bar shows 'C:\WINDO...'. The window contains two lines of text: 'Vector 3 là: 5.3, 8.2' and 'Vector 4 là: -1.3, -2'. The text is displayed in a monospaced font with some color coding (green for 'Vector', red for 'là:', and blue for the values).

Nạp chồng toán tử

Một số lưu ý:

📖 Các phép toán chia làm 3 loại:

- *Unary*: chỉ cần 1 toán hạng: phép ++, --.
- *Binary*: cần 2 toán hạng: +, -, *, /
- *Ternary*: cần 3 toán hạng: phép toán điều kiện ?
- Nạp chồng phép toán nào thì phải cung cấp đủ số lượng tham số phù hợp.

📖 Đối với các phép toán so sánh, phải thực hiện nạp chồng cả cặp; tức là: nếu nạp chồng phép == thì phải định nghĩa cả nạp chồng phép !=; nếu nạp chồng phép > thì phải có nạp chồng phép <.

📖 Các phép gán +=, -= không cho phép nạp chồng trực tiếp. Tuy nhiên, nếu đã có nạp chồng phép +, - thì các phép gán sẽ được nạp chồng.

Nạp chồng toán tử

VD:

```
//Class Vector
public static Vector operator +(Vector a, double i)
{
    return new Vector(a.X += i, a.Y += i);
}
```

```
//Main
Vector vec1 = new Vector(2.0, 3.1);
vec3 = vec1 += 5;
```



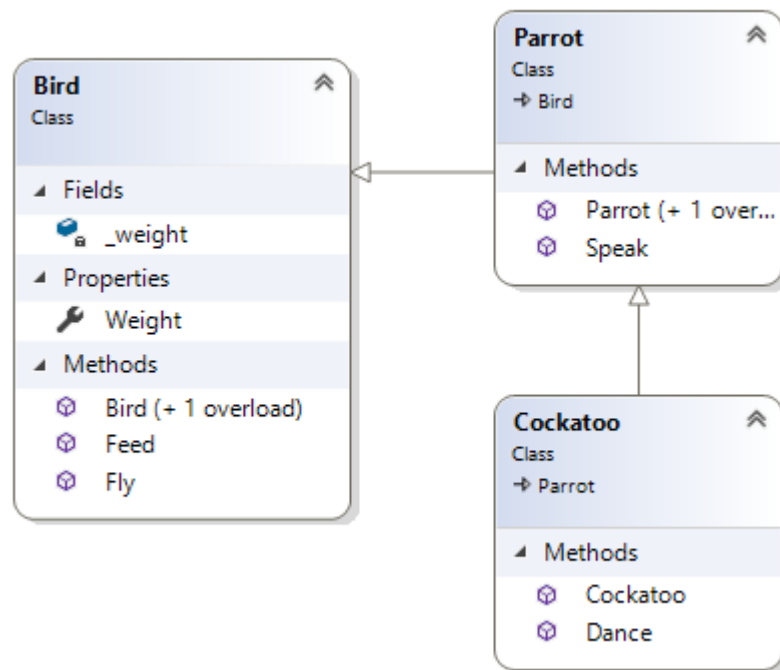

Kế thừa (Inheritance)

Kế thừa (inheritance)

- 📖 Kế thừa (inheritance): cho phép tạo ra class mới từ một class đã có.
- 📖 Cho phép tái sử dụng code của class đã có, giảm thiểu việc lặp code khi xây dựng class mới.
- 📖 Dễ bảo trì và giảm thời gian xây dựng, phát triển.
- 📖 Quan hệ kế thừa được thể hiện bằng dấu ":" phân chia tên class mới với tên class được kế thừa.

📖 VD:

```
class Bird
class Parrot : Bird
class Cockatoo : Parrot
```



Đặc điểm kế thừa của C#

- ❏ Lớp có sẵn để từ đó tạo ra lớp khác gọi là lớp cha/lớp cơ sở.
- ❏ Lớp mới được định nghĩa dựa trên lớp cơ sở gọi là lớp con/lớp dẫn xuất.
- ❏ Chỉ cho phép một lớp con có một lớp cha trực tiếp; gọi là kế thừa đơn (single inheritance).
- ❏ Lớp con có thể trở thành lớp cơ sở cho lớp khác.
- ❏ Từ khóa **base** dùng để truy xuất các thành viên của lớp cha.
- ❏ Class với từ khóa **sealed** không cho phép kế thừa.
- ❏ Lớp kế thừa từ lớp khác sẽ thừa hưởng tất cả các thành viên của lớp cha và cả những thành viên của *lớp ông*; trừ các thành viên kiểu private, các hàm tạo, hàm hủy.
- ❏ Mọi class đều kế thừa của lớp **System.Object**.
- ❏ Khi lớp cha khai báo **internal** thì lớp con phải là **internal**, không được public. Ngược lại, khi lớp cha là **public** thì lớp con có thể là **public hoặc internal**

Kế thừa của hàm khởi tạo

- 📖 Khi tạo đối tượng của lớp con, thì hàm tạo của lớp cha luôn được gọi trước.
- 📖 Hàm tạo không được kế thừa; hàm tạo của lớp cha được gọi tự động (hàm tạo không tham số) hoặc được gọi từ hàm tạo của lớp con (hàm tạo có tham số).
- 📖 Để gọi trực tiếp hàm tạo của lớp cha ở lớp con: dùng lệnh **base**
- 📖 Xét một số trường hợp :

Kế thừa của hàm khởi tạo

📖 **Lớp cha không có hàm tạo:** lớp con có hàm tạo (có tham số hoặc không tham số) thì khi khởi tạo đối tượng của lớp con bằng hàm tạo nào thì sẽ thực thi hàm tạo đó.

```
internal class Bird {}
```

```
internal class Parrot : Bird
```

```
{
```

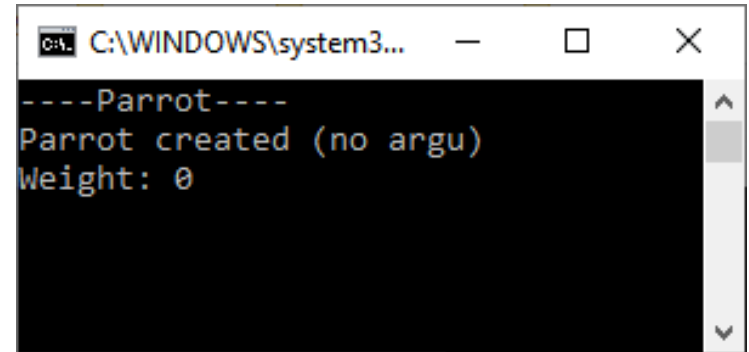
```
    public Parrot() => Console.WriteLine("Parrot created (no  
argu)");
```

```
    public Parrot(int weight) => Console.WriteLine("Parrot created  
(argu)");
```

```
}
```

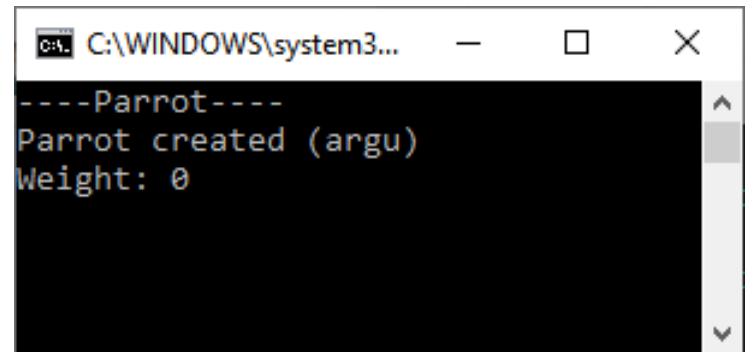
Kế thừa của hàm khởi tạo

```
//main  
Console.WriteLine("----Parrot----");  
Parrot parrot = new Parrot();
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text: '----Parrot----', 'Parrot created (no argu)', and 'Weight: 0'.

```
//main  
Console.WriteLine("----Parrot----");  
Parrot parrot = new Parrot(200);
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text: '----Parrot----', 'Parrot created (argu)', and 'Weight: 0'.

Kế thừa của hàm khởi tạo

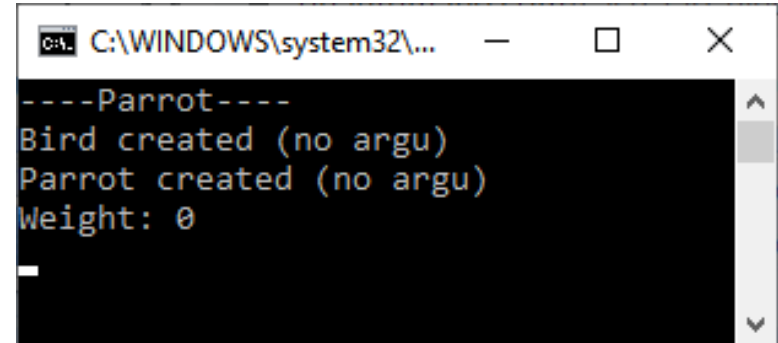
📖 **Lớp cha có hàm tạo không tham số:** lớp con có hàm tạo (có tham số hoặc không tham số) thì khi khởi tạo đối tượng của lớp con bằng hàm tạo nào thì sẽ thực thi hàm tạo đó, và trước đó sẽ gọi hàm tạo không tham số của lớp cha.

```
internal class Bird {  
    public Bird() => Console.WriteLine($"Bird created (no argu)");  
}
```

```
internal class Parrot : Bird  
{  
    public Parrot() => Console.WriteLine("Parrot created (no  
argu)");  
    public Parrot(int weight) => Console.WriteLine("Parrot created  
(argu)");  
}
```

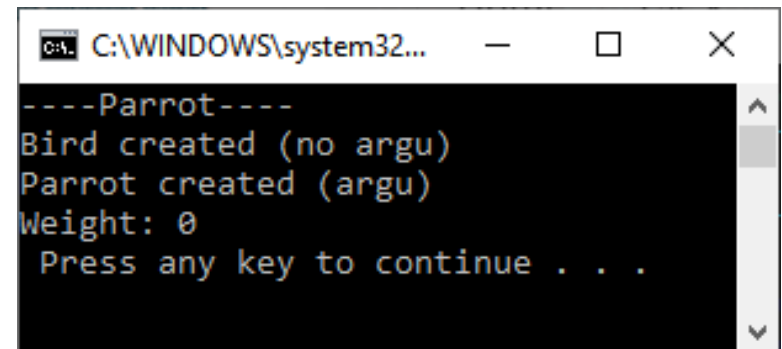
Kế thừa của hàm khởi tạo

```
//main  
Console.WriteLine("----Parrot----");  
Parrot parrot = new Parrot();
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\...'. The command prompt displays the output of the first code block: '----Parrot----', 'Bird created (no argu)', 'Parrot created (no argu)', and 'Weight: 0'. There is a cursor on the line following 'Weight: 0'.

```
//main  
Console.WriteLine("----Parrot----");  
Parrot parrot = new Parrot(200);
```

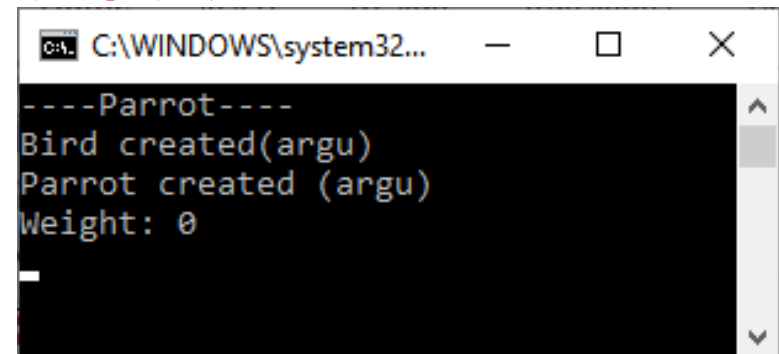


A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\...'. The command prompt displays the output of the second code block: '----Parrot----', 'Bird created (no argu)', 'Parrot created (argu)', and 'Weight: 0'. Below this, it says 'Press any key to continue . . .'. There is a cursor on the line following 'Press any key to continue . . .'.

Kế thừa của hàm khởi tạo

📖 **Lớp cha có hàm tạo có tham số, không có hàm tạo không tham số:** lớp con không thể có hàm tạo không tham số được; chỉ được viết hàm tạo có tham số theo cấu trúc sau.

```
internal class Bird {  
    public Bird(int weight) => Console.WriteLine("Bird created  
(argu)");  
}  
  
internal class Parrot : Bird{  
    public Parrot() => Console.WriteLine("Parrot created (no  
argu)");  
    public Parrot(int weight) => Console.WriteLine("Parrot created  
(argu)");  
    public Parrot(int weight) : base(weight) =>  
        Console.WriteLine("Parrot created (argu)");  
}  
  
//main  
Parrot parrot = new Parrot();  
Parrot parrot = new Parrot(200);
```



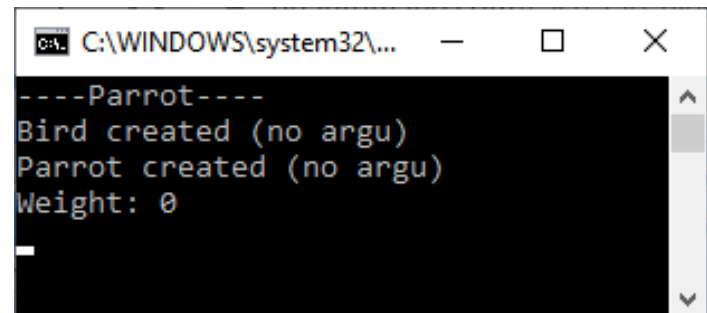
```
C:\WINDOWS\system32...  
----Parrot----  
Bird created(argu)  
Parrot created (argu)  
Weight: 0
```

Kế thừa của hàm khởi tạo

📖 **Lớp cha có hàm tạo không tham số và hàm tạo có tham số:**
lớp con có hàm tạo không tham số được và hàm tạo có tham số.

```
internal class Bird {  
    public Bird() => Console.WriteLine("Bird created (no argu)");  
    public Bird(int weight) => Console.WriteLine("Bird created  
(argu)");  
}  
internal class Parrot : Bird{  
    public Parrot() => Console.WriteLine("Parrot created (no  
argu)");  
    public Parrot(int weight) => Console.WriteLine("Parrot created  
(argu)"); hoặc  
    public Parrot(int weight) : base(weight) =>  
        Console.WriteLine("Parrot created (argu)");  
}
```

```
//main  
Console.WriteLine("----Parrot----");  
Parrot parrot = new Parrot();
```

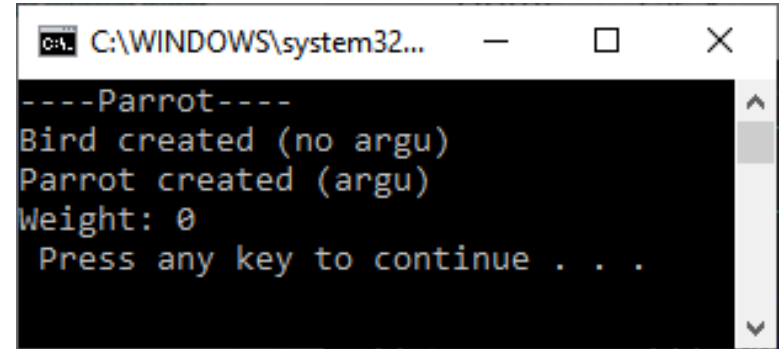


```
C:\WINDOWS\system32\...  
----Parrot----  
Bird created (no argu)  
Parrot created (no argu)  
Weight: 0
```

Kế thừa của hàm khởi tạo

Với: `public Parrot(int weight) => Console.WriteLine("Parrot created (argu)");`

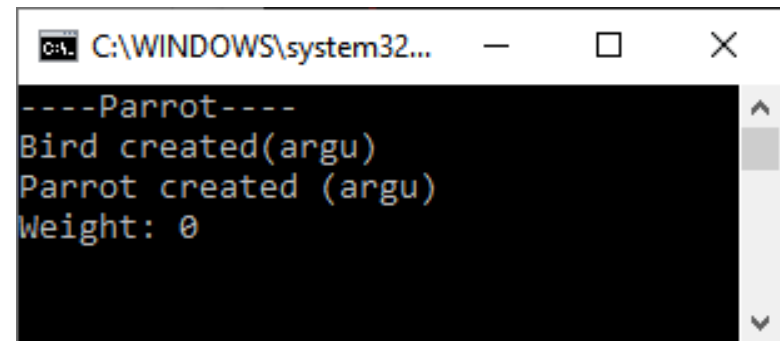
```
//main  
Console.WriteLine("----Parrot----");  
Parrot parrot = new Parrot(200);
```



```
C:\WINDOWS\system32...  
----Parrot----  
Bird created (no argu)  
Parrot created (argu)  
Weight: 0  
Press any key to continue . . .
```

Với: `public Parrot(int weight) : base(weight) => Console.WriteLine("Parrot created (argu)");`

```
//main  
Console.WriteLine("----Parrot----");  
Parrot parrot = new Parrot(200);
```



```
C:\WINDOWS\system32...  
----Parrot----  
Bird created(argu)  
Parrot created (argu)  
Weight: 0
```

Kế thừa (inheritance)

```
internal class Bird{
    private int _weight;
    public int Weight{
        get => _weight;
        set{
            if (value > 0)
                _weight = value;
        }
    }
    public Bird() => Console.WriteLine($"Bird created");
    public Bird(int weight){
        _weight = weight;
        Console.WriteLine($"Bird created, {_weight} gr.");
    }
    public void Feed() => _weight += 10;
    public void Fly() => Console.WriteLine("Bird is flying");
}
```

Kế thừa (inheritance)

```
/// <summary>
/// Class Parrot kế thừa class Bird
/// </summary>
internal class Parrot : Bird{
    public Parrot() => Console.WriteLine("Parrot created");
    public Parrot(int weight) : base(weight) { }
    public void Speak() => Console.WriteLine("Parrot is speaking");
}

/// <summary>
/// Class Cockatoo kế thừa class Parrot
/// </summary>
internal class Cockatoo : Parrot{
    public Cockatoo() => Console.WriteLine("Cockatoo created");
    public void Dance() => Console.WriteLine("Cockatoo is dancing");
}
```

Kế thừa (inheritance)

```
class Program{
    static void Main(string[] args){
        Console.WriteLine("----Bird:----");
        Bird bird = new Bird(50) { Weight = 100 };
        bird.Feed();
        Console.WriteLine($"Weight {bird.Weight}");
        bird.Fly();

        Console.WriteLine("----Parrot----");
        Parrot parrot = new Parrot(200);
        parrot.Feed();
        Console.WriteLine($"Weight: {parrot.Weight}");
        parrot.Fly();
        parrot.Speak();
        Console.WriteLine("----Cockatoo:----");
        Cockatoo cockatoo = new Cockatoo() { Weight = 300 };
        cockatoo.Feed();
        Console.WriteLine($"Weight: {cockatoo.Weight}");
        cockatoo.Fly();
        cockatoo.Speak();
        cockatoo.Dance();
        Console.ReadKey();
    }
}
```

Kế thừa (inheritance)

```
C:\WINDOWS\system...
----Bird:----
Bird created, 50 gr.
Weight 110
Bird is flying

----Parrot----
Bird created, 200 gr.
Weight: 210
Bird is flying
Parrot is speaking

----Cockatoo:----
Bird created
Parrot created
Cockatoo created
Weight: 310
Bird is flying
Parrot is speaking
Cockatoo is dancing
```



Đa hình (Polymorphism)

Đa hình (polymorphism)

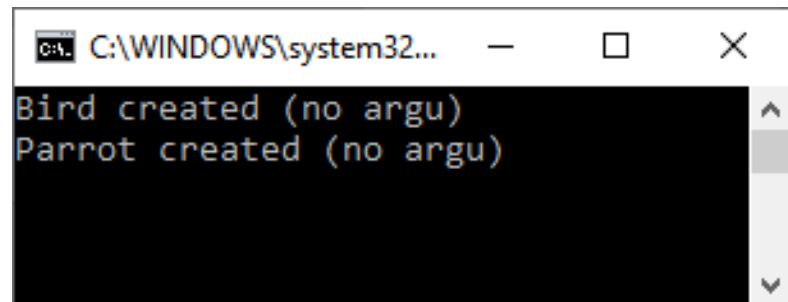
📖 **Đa hình (polymorphism):** nghĩa là nhiều hình thức; trong mô hình lập trình hướng đối tượng được diễn tả “một giao diện, nhiều chức năng”.

📖 Có 2 loại đa hình trong C#:

- Đa hình thời gian biên dịch (*compile time*) còn gọi là đa hình tĩnh (*static polymorphism*): thực hiện bằng nạp chồng phương thức (*method overloading*) và nạp chồng toán tử (*operator overloading*).
- Đa hình thời gian chạy (*runtime*) còn gọi là đa hình động (*dynamic polymorphism*): thực hiện bằng ghi đè phương thức (*method overriding*)

📖 VD:

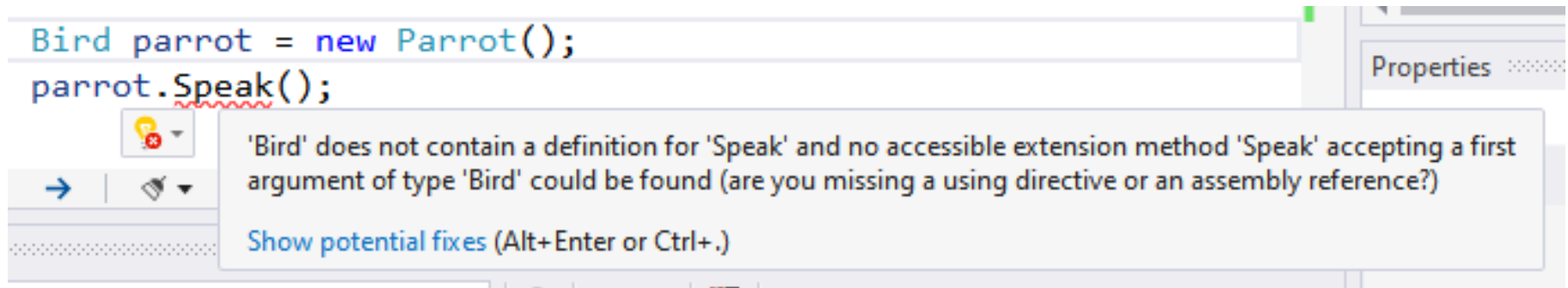
```
Bird parrot = new Parrot();
```



```
C:\WINDOWS\system32...  
Bird created (no argu)  
Parrot created (no argu)
```

Đa hình (polymorphism)

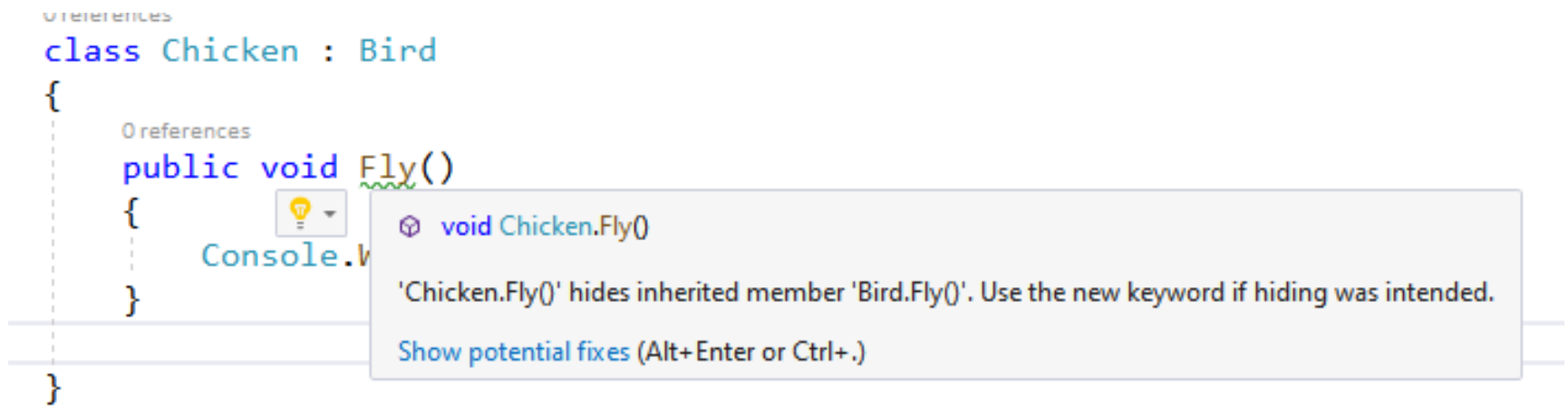
- ❏ Khi đối tượng của lớp con được khởi tạo, các hàm tạo được gọi theo trật tự của hàm tạo.
- ❏ Tuy nhiên, đối tượng này được tham chiếu đến biến thuộc kiểu cha. Vì vậy, đối tượng này chỉ có thể sử dụng được biến và phương thức của lớp cha.



- ❏ Để giải quyết, C# cung cấp hai cách: *che dấu (hiding)* và *ghi đè (overriding)*.

Che dấu phương thức (method hiding)

- Class con được thừa hưởng tất cả các biến và phương thức mà class cha có (trừ private).
- Khi định nghĩa một phương thức ở class con trùng với phương thức được kế thừa từ class cha:



- Ý nghĩa: phương thức `Fly` của lớp `Chicken` sẽ che đi phương thức `Fly` của `Bird`.
- Lúc này C# sẽ áp dụng cơ chế *che dấu phương thức (method hiding)*

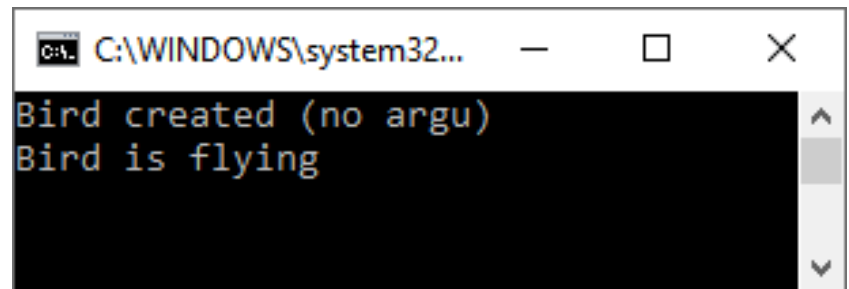
Che dấu phương thức (method hiding)

☐ Để đảm bảo đây là phương thức của lớp con, dùng từ khóa ***new*** trước khai báo phương thức.

```
public new void Fly() => Console.WriteLine("Chicken cannot fly");
```

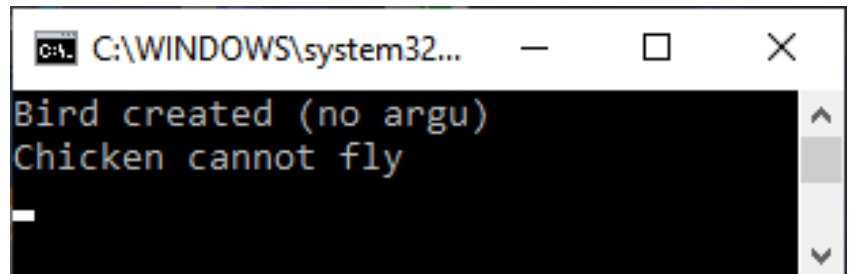
☐ Khi đó, cả hai phương thức Fly đều tồn tại trong đối tượng của lớp con. Tùy vào việc khi khởi tạo đối tượng lớp con thuộc kiểu nào thì sẽ thực thi phương thức của lớp kiểu đó.

```
Bird chicken = new Chicken();  
chicken.Fly();
```



```
C:\WINDOWS\system32...  
Bird created (no argu)  
Bird is flying
```

```
Chicken chicken = new Chicken();  
chicken.Fly();
```



```
C:\WINDOWS\system32...  
Bird created (no argu)  
Chicken cannot fly
```

Ghi đè phương thức (method overriding)

- Trong cơ chế ghi đè phương thức (method overriding) thì phương thức của lớp cha được kế thừa ở lớp con sẽ bị xóa bỏ và thay bằng phương thức mới định nghĩa ở lớp con.
- Do đó bất kể đối tượng của lớp con được khởi tạo theo kiểu biến nào (kiểu lớp cha hoặc lớp con) thì cũng chỉ truy xuất vào phương thức ở lớp con.
- Để thực hiện ghi đè phương thức cần:
 - Phương thức ở lớp cha được cho phép ghi đè với từ khóa ***virtual*** trước khai báo phương thức.
 - Phương thức ở lớp con phải có từ khóa ***override*** trước định nghĩa phương thức để xác định đây là phương thức ghi đè.

Ghi đè phương thức (method overriding)

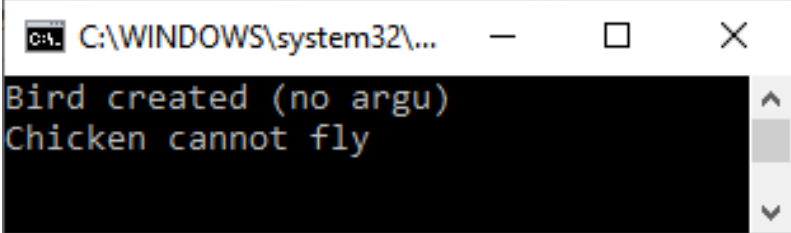
```
//Lớp Bird
```

```
public virtual void Fly() => Console.WriteLine("Bird is flying");
```

```
//Lớp Chicken
```

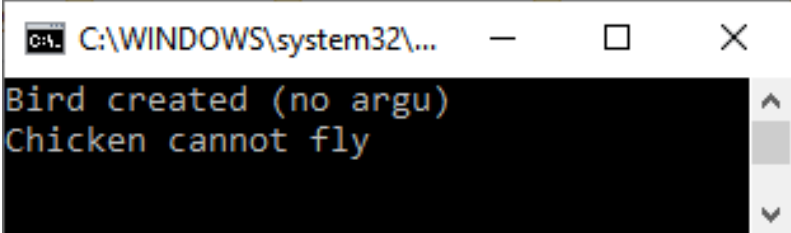
```
public override void Fly() => Console.WriteLine("Chicken cannot fly");
```

```
Chicken chicken = new Chicken();  
chicken.Fly();
```



```
C:\WINDOWS\system32\...  
Bird created (no argu)  
Chicken cannot fly
```

```
Bird chicken = new Chicken();  
chicken.Fly();
```



```
C:\WINDOWS\system32\...  
Bird created (no argu)  
Chicken cannot fly
```



Lớp trừu tượng (Abstract class)

Lớp trừu tượng (abstract class)

📖 Là lớp được dùng làm khuôn mẫu để tạo ra các lớp khác mà không cài đặt chi tiết.

📖 Khai báo lớp trừu tượng:

```
[public|internal] abstract class <tên class>{...}
```

📖 VD: `internal abstract class Animal{...}`

📖 Đặc điểm:

- Thêm từ khóa ***abstract*** vào trước từ khóa class để khai báo một lớp trừu tượng.
- Lớp trừu tượng không được sử dụng để tạo ra đối tượng như lớp bình thường.
- Lớp trừu tượng có thể có hàm tạo.
- Bên trong lớp trừu tượng chứa các phương thức khuôn mẫu để cho lớp con kế thừa và cài đặt gọi là phương thức trừu tượng (**abstract method**)

Lớp trừu tượng (abstract class)

- Các lớp con dẫn xuất từ lớp trừu tượng phải cài đặt (override) tất cả các phương thức trừu tượng trong lớp trừu tượng. Nếu không cài đặt đủ thì lớp con cũng phải khai báo là abstract
- Lớp trừu tượng không được khai báo là **sealed** và khai báo với từ khóa **static**.
- Trong lớp trừu tượng có thể có 3 loại phương thức: trừu tượng, virtual và thường.
- Nếu một class có một phương thức trừu tượng thì class đó bắt buộc phải khai báo là trừu tượng.

Phương thức trừu tượng (abstract method)

📖 Khai báo phương thức trừu tượng:

```
[public|protected] abstract <kiểu dữ liệu trả về> <tên phương thức>([danh sách tham số]);
```

📖 Đặc điểm:

- Chỉ là dạng khuôn mẫu (prototype) nên không có nội dung (không có phần trong “{}”, kết thúc bằng dấu “;”)
- Mặc định là **virtual** nên không được dùng từ khóa virtual trước phương thức abstract.
- Phạm vi truy cập là **public** hoặc **protected** không là **private**
- Phạm vi truy cập phải giống nhau ở phần khai báo của cả lớp cha và lớp con. Nếu ở lớp cha khai báo **protected** thì lớp con cũng phải khai báo **protected**.
- Không thể là phương thức **static**

Lớp trừu tượng

```
public abstract class Animal{  
    /// <summary>  
    /// Abstract method  
    /// </summary>  
    public abstract void AnimalSound();  
  
    /// <summary>  
    /// Virtual method  
    /// </summary>  
    public virtual void Eat() => Console.WriteLine("Ăn cỏ");  
  
    /// <summary>  
    /// Phương thức thường  
    /// </summary>  
    public void Run() => Console.WriteLine("Chạy bằng 4 chân");  
}
```

Lớp trừu tượng

```
/// <summary>
/// Class Dog kế thừa class Animal
/// </summary>
public class Dog: Animal
{
    /// <summary>
    /// Override abstract method
    /// </summary>
    public override void AnimalSound() => Console.WriteLine("Gâu  
gâu...");

    /// <summary>
    /// Override virtual method
    /// </summary>
    public override void Eat() => Console.WriteLine("Ăn cơm");
}
```

Lớp trừu tượng

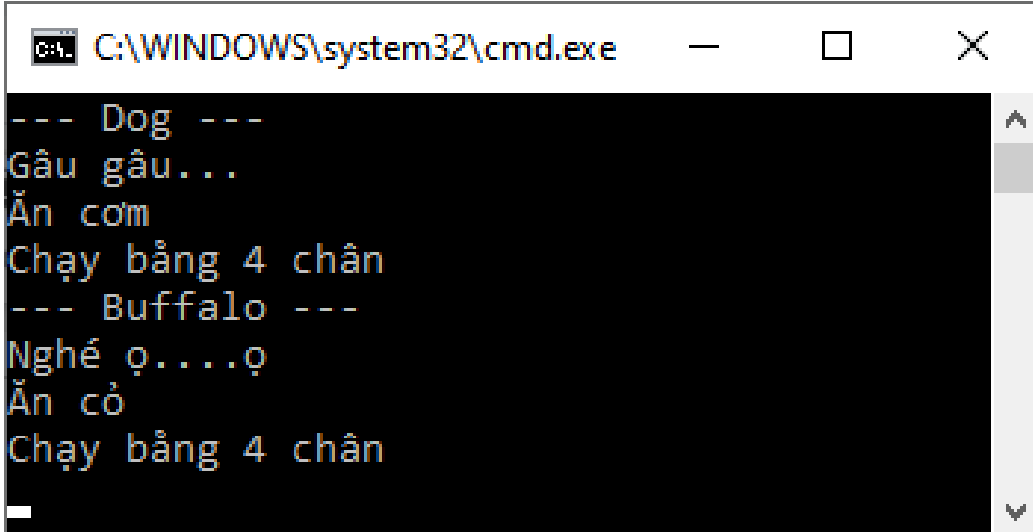
```
/// <summary>
/// Lớp Buffalo kế thừa lớp Animal
/// </summary>
public class Buffalo: Animal
{
    /// <summary>
    /// Override abstract method
    /// </summary>
    public override void AnimalSound() => Console.WriteLine("Nghé
ợ....ợ");

    /// <summary>
    /// Override virtual method
    /// </summary>
    public override void Eat() => Console.WriteLine("Ăn cỏ");
}
```

Lớp trừu tượng

```
static void Main(string[] args)
{
    Console.WriteLine("--- Dog ---");
    Dog dog = new Dog();
    dog.AnimalSound();
    dog.Eat();
    dog.Run();

    Console.WriteLine("--- Buffalo ---");
    Buffalo buffalo = new Buffalo();
    buffalo.AnimalSound();
    buffalo.Eat();
    buffalo.Run();
    Console.ReadKey();
}
```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a black background and displays the output of the C# program. The output is as follows:

```
--- Dog ---
Gâu gâu...
Ăn cơm
Chạy bằng 4 chân
--- Buffalo ---
Nghé ọ....ọ
Ăn cỏ
Chạy bằng 4 chân
```

The text is displayed in a monospaced font, with the program's output lines matching the code's formatting (e.g., "Dog" and "Buffalo" are in red in the original image). The window includes standard Windows window controls (minimize, maximize, close) in the top right corner.

Khác nhau giữa abstract và virtual method

Abstract method	Virtual method
Không có thân hàm	Có thân hàm
Bắt buộc phải override ở lớp con kế thừa.	Không bắt buộc phải override ở lớp con kế thừa
Chỉ sử dụng trong abstract class	Sử dụng được ở class thường và abstract class



Giao diện (Interface)

Giao diện (interface)

- 📖 **Giao diện (interface):** là một kiểu class nhưng chỉ có phần mô tả (specification/declaration) của các chi tiết phương thức trong class mà không có phần thực thi.
- 📖 Cài đặt của các phương thức sẽ được thực hiện trong class con dẫn xuất từ giao diện này.

- 📖 Khai báo lớp giao diện:

```
[public|internal] interface <tên class>{...}
```

- 📖 VD:

```
interface IPet {  
    void AnimalSound();  
    void Eat();  
}
```

- 📖 Đặc điểm:

- Giao diện được khai báo với từ khóa **interface**.
- Không thể khởi tạo đối tượng từ giao diện.
- Phạm vi truy cập là *public* hoặc *internal* (mặc định *internal*)

Giao diện (interface)

- Không thể định nghĩa các biến thành viên (field).
- Một lớp con có thể thực thi nhiều class interface.
- Tên của giao diện đặt giống quy ước đặt tên class nhưng thường có thêm chữ “I” đứng trước.
- Không có hàm tạo.
- Mặc định các phương thức đều là phương thức trừu tượng và public; nên không cho phép ghi **public** và **abstract**.
- Chỉ có mô tả, không có thân phương thức; khi mô tả phương thức không có các từ khóa điều khiển truy cập (không có public, private, protected).

📖 Class thực thi interface khai báo như kế thừa.

📖 VD: `class Dog : Ipet`
`class Cat : Ipet`

📖 Một class cũng có thể thực thi nhiều interface

`class Parrot : Ipet, IBird`

Giao diện (interface)

📖 Cài đặt các phương thức trong interface, 2 cách:

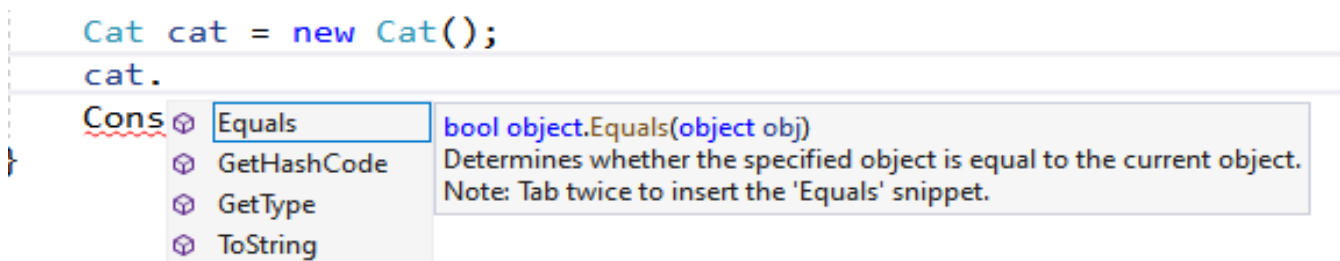
- Implicit

```
class Dog : IPet{  
    public Dog() => Console.WriteLine("Tôi là chó");  
    public void AnimalSound() => Console.WriteLine("Gâu gâu");  
    public void Eat() => Console.WriteLine("Ăn cơm");  
}
```

- Explicit: mỗi phương thức khi cài đặt phải chỉ rõ thuộc interface nào, và không cho phép sử dụng từ khóa điều khiển truy cập.

```
class Cat : IPet{  
    public Cat() => Console.WriteLine("Tôi là mèo");  
    void IPet.AnimalSound() => Console.WriteLine("Méo méo...");  
    void IPet.Eat() => Console.WriteLine("Ăn cá");  
}
```

- Khi dùng explicit thì không thể gọi được trên đối tượng.



Giao diện (interface)

Để sử dụng interface:

- Có thể sử dụng như một kiểu dữ liệu để khai báo biến và tham chiếu tới đối tượng của class đã cài đặt interface đó. Dùng cho các class cài đặt theo kiểu explicite.

```
IPet cat = new Cat();  
cat.AnimalSound();
```

- Khởi tạo đối tượng với class cài đặt interface (kiểu implicit) và sử dụng như class bình thường.

```
Dog dog = new Dog();  
dog.AnimalSound();  
dog.Eat();
```

So sánh abstract và interface

Giống:

- ❏ Đều không thể dùng để khởi tạo đối tượng.
- ❏ Đều định ra khuôn mẫu cho phương thức và không cài đặt
- ❏ Các lớp con dẫn xuất phải cài đặt các phương thức khuôn mẫu này.
- ❏ Có thể kế thừa từ nhiều interface khác.

Khác:

Abstract class	Interface class
Cho phép khai báo biến	Không cho phép
Có thể chứa hàm tạo.	Không thể có hàm tạo.
Phương thức có từ khóa điều khiển truy cập	Mặc định là public và không được ghi
Có các phương thức đã được cài đặt	Chỉ khai báo phương thức và không cài đặt.
Chỉ được kế thừa từ 1 abstract class khác và nhiều interface	Một class có thể cài đặt nhiều interface