

A teal-colored arrow pointing to the right, with a white outline and a slight 3D effect, serving as a background for the chapter title.

Chương 2: LẬP TRÌNH HĐT TRÊN .NET

Mục tiêu

- - Cài đặt được ứng dụng Visual Studio .NET
- - Hiểu được các tạo một ứng dụng .NET
- - Áp dụng cấu trúc điều khiển trong C#
- - Kỹ năng debug code
- - Áp dụng kiến thức OOP vào C#.
- - Xây dựng được class, tạo đối tượng trong C#
- - Áp dụng kỹ thuật kế thừa trong C#
- - Áp dụng kỹ thuật kết tập trong C#

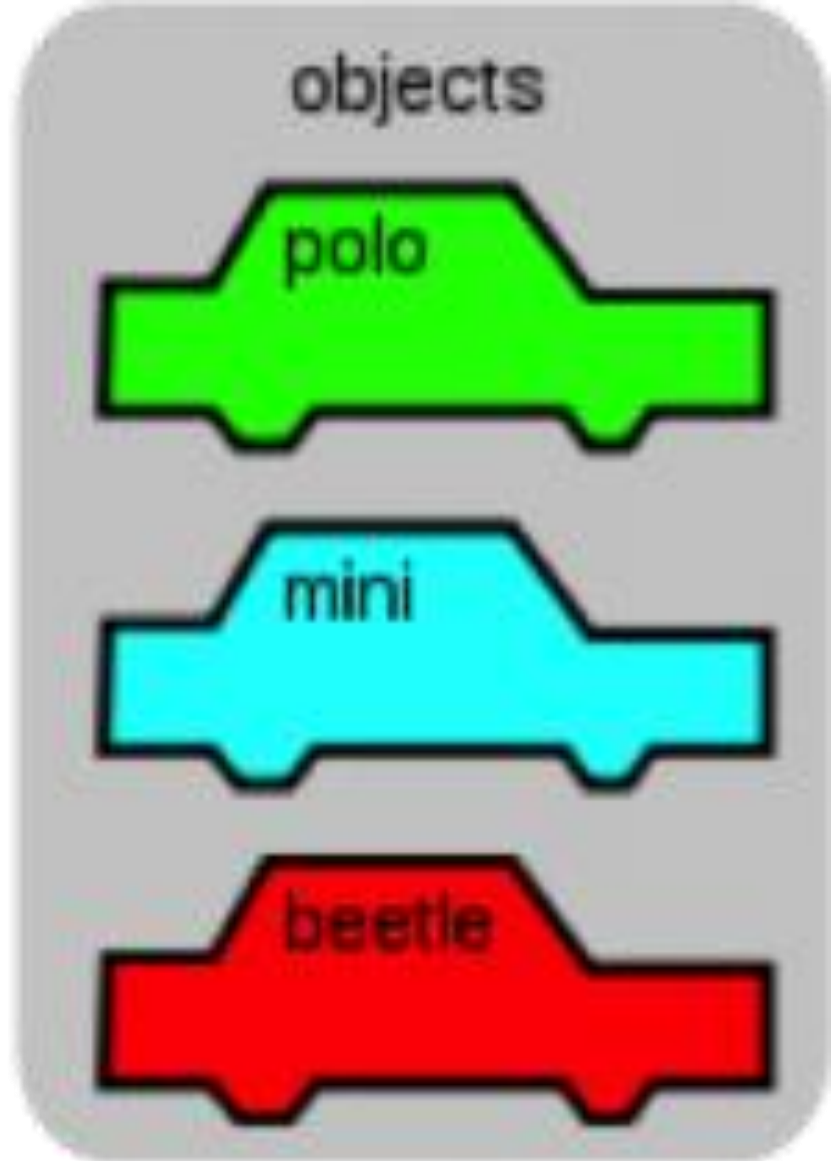
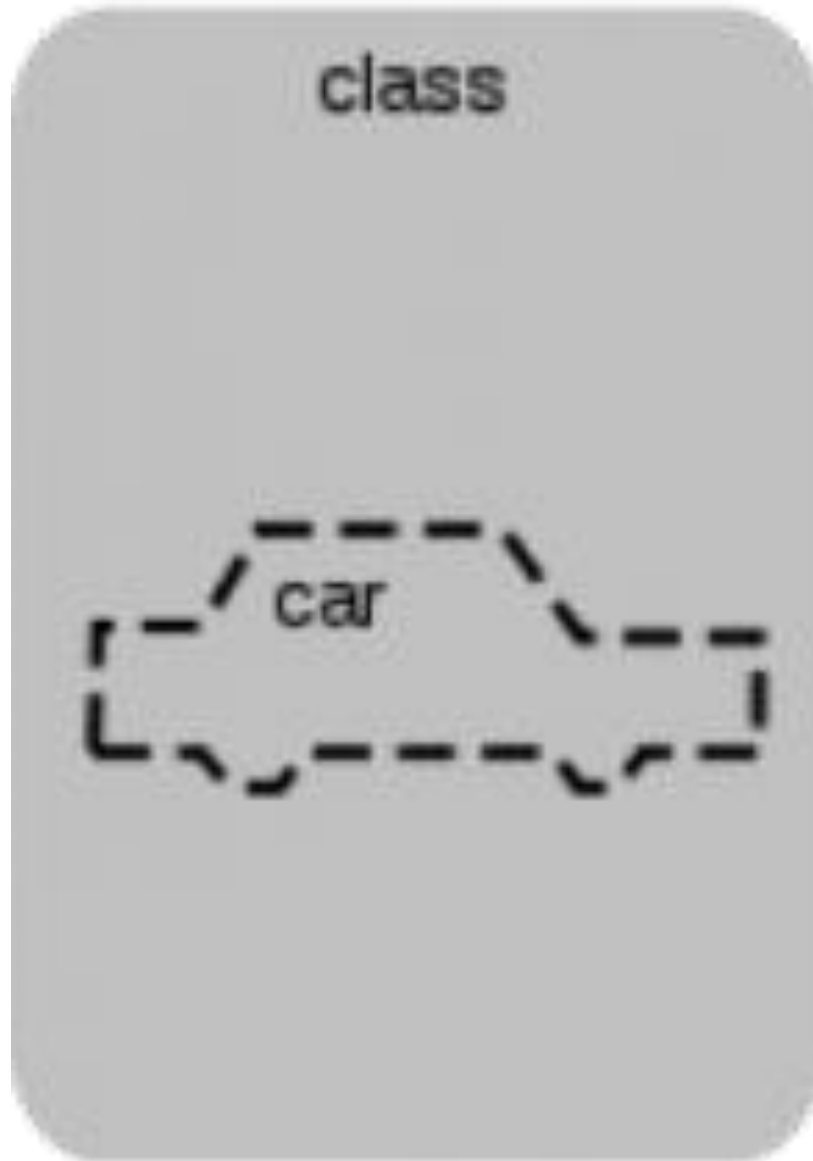
Nội dung

- Xây dựng lớp, đối tượng
- Kế thừa, đa hình
- Nạp chồng toán tử
- Triển khai Nạp chồng toán tử trong C#
- Bài tập tổng hợp (Class Room).

Xây dựng lớp, đối tượng

- **Class** trong **OOP** định nghĩa **trình tượng** các đặc tính của đối tượng, chúng ta hiểu nôm na giống như **bản thiết kế** hay **khuôn mẫu** của đối tượng nào đó, ví dụ như **bản thiết kế** của ô tô, bản vẽ của toà nhà, các control như TextBox, Button, Label.
- **Object** là một sản phẩm được tạo ra từ bản thiết kế của **Class**, ví dụ bản thiết kế ô tô đó được mang đi sản xuất, bản vẽ toà nhà được dùng để xây dựng, TextBox, Button ... được kéo vào Form thì lúc đó sản phẩm tạo từ các bản thiết kế đó gọi là Object.

Xây dựng lớp, đối tượng



Định nghĩa lớp

➤ Định nghĩa lớp

↪ Cú pháp:

```
[attributes] [modifiers] class <ClassName>[:BaseClassName]  
{  
    [class-body]  
}[;]
```

1

2

Định nghĩa lớp

➤ Ví dụ

```
class NhanVien  
{  
    private long maNV;  
}
```

Access
modifier

- Không cần thiết phải “;” để kết thúc lớp.

```
class MyClass  
{  
    // members  
}
```

```
class MyClass  
{  
    // members  
};
```

Thân của lớp

- Một lớp trong C# có thể chứa các loại thành viên
 - ↳ Constants
 - ↳ Fields
 - ↳ Methods
 - ↳ Operators
 - ↳ Properties
 - ↳ Events
 - ↳ Indexers
 - ↳ Instance constructors, static constructors
 - ↳ Destructors
 - ↳ Các kiểu khai báo lồng nhau (class, struct, interface, enum, delegate)

Thân của lớp

```
public class First  
{
```

```
    const int MAX = 5;  
    string name;
```

```
    public int Say(string message)  
    {  
        return message + " " + name;  
    }
```

```
    public class Second  
    {  
        ...  
    }
```

```
}
```

Kiểu ở mức
cao nhất
(top-level)

Kiểu bị lồng
(nested)

Bổ từ truy cập Cho kiểu ở mức cao nhất

- Có 2 bổ từ truy cập cho kiểu ở mức cao nhất

- ↪ public

- ↪ internal

```
public class AccessTypeInCSharp  
{  
  
}
```

- DEFAULT: internal

```
class AccessInCSharp  
{  
  
}
```

```
internal class AccessInCSharp  
{  
  
}
```

Bổ từ truy cập Cho thành viên

- Có 5 bổ từ truy cập cho thành viên

↪ public

↪ protected

↪ private

↪ internal

↪ internal protected

- DEFAULT: private

```
class AccessMembersInCSharp
{
    public int a;
    public int b;
    public int c;
    protected int d;
    protected int e;
}
```

Bổ từ truy cập Cho thành viên

C++

```
class AccessMembersInCSharp
{
public:
    int a;
    int b;
    int c;

protected:
    int d;
    int e;
};
```

C#

```
class AccessMembersInCSharp
{
    public int a;
    public int b;
    public int c;
    protected int d;
    protected int e;
}
```

```
class AccessMembersInCSharp
{
    protected int a;
    int b;
}
```

Bổ từ truy cập Cho thành viên

- Nếu kiểu có bổ từ truy cập là **internal** (hay không có bổ từ truy cập) thì các thành viên của kiểu này có thể khai báo 3 cấp độ truy cập (giống C++)
 - ↪ public = internal = internal protected
 - ↪ protected
 - ↪ private
- Nếu kiểu có bổ từ truy cập là **public** thì các thành viên của kiểu này có thể khai báo 5 cấp độ truy cập
 - ↪ public
 - ↪ protected
 - ↪ private
 - ↪ internal
 - ↪ internal protected

Thân của lớp Field

➤ Field

➤ Một field là một biến thành viên dùng để lưu giữ giá trị của một đối tượng

➤ Cú pháp:

```
<type> <name>;
```

- Trong đó <type> có thể là
 - Kiểu cơ sở: char, int, float, double, ...
 - Enum
 - Struct
 - Class
 - Delegate

Thân của lớp Field

➡ Quy tắc về khai báo Field

- Field có thể là một đối tượng của lớp đang định nghĩa
- Có thể vừa khai báo, vừa khởi tạo dữ liệu cho field

Thân của lớp Phương thức

- Phương thức và tham số
 - ↳ Trong C#, khái niệm phương thức (method) và hàm (function) là đồng nghĩa với nhau. Phương thức là đoạn mã thao tác trên các field.
 - ↳ Trong C#, định nghĩa phương thức theo quy tắc
 - Phương thức phải nằm trong class hay struct
 - Thân của phương thức nằm trong định nghĩa lớp (nghĩa là không có sự phân biệt giữa khai báo và định nghĩa phương thức)

Thân của lớp Phương thức

```
class ClassName
```

```
{
```

```
    Method(int data)
```

```
    {
```

```
        ...
```

```
    }
```

```
}
```

```
class Program
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        int x = 7;
```

```
        ClassName obj = new ClassName();
```

```
        obj.Method(x);
```

```
    }
```

```
}
```

Tham số
hình thức

Tham số
thực

Thân của lớp Phương thức

- Các tham số của phương thức: có 2 loại tham số
 - Tham số giá trị (Value parameter – tham trị):
 - Khi gọi phương thức có tham số giá trị thì chúng ta đang gửi **một bản sao** của tham số thực cho phương thức (bất kỳ thay đổi dữ liệu của tham số trong phương thức cũng **không** ảnh hưởng đến dữ liệu ban đầu được lưu trong tham số thực)
 - Đây là cách truyền mặc định

Thân của lớp Phương thức

➤ Tham tri với kiểu giá trị

```
class SomeClass
{
    public void Change(int data)
    {
        data = data*2;
    }
}
class Program
{
    static void Main()
    {
        SomeClass obj = new SomeClass();
        int x=7;
        obj.Change(x);
        Console.WriteLine("x= {0}" + x);
    }
}
```



Thân của lớp Phương thức

➤ Tham tri với kiểu tham chiếu

```
class AnotherClass
{
    public int data;
}
class SomeClass
{
    public void Change(AnotherClass obj)
    {
        obj = new AnotherClass();
        obj.data = 100;
    }
}
class Program
{
    static void Main()
    {
        AnotherClass at = new AnotherClass()
        SomeClass sc = new SomeClass();
        at.data = 7;
        sc.Change(at);
        Console.WriteLine("data=" + at.data);
    }
}
```

Thân của lớp Phương thức

➤ Tham tri với kiểu tham chiếu

```
class AnotherClass
{
    public int data;
}
class SomeClass
{
    public void Change(AnotherClass obj)
    {
        obj.data = obj.data*2;
    }
}
class Program
{
    static void Main()
    {
        AnotherClass at = new AnotherClass()
        SomeClass sc = new SomeClass();
        at.data = 7;
        sc.Change(at);
        Console.WriteLine("data=" + at.data);
    }
}
```

Thân của lớp Phương thức

➤ Tham trị với kiểu giá trị

```
class SomeClass
{
    public void Change(string data)
    {
        data = "New String";
    }
}
class Program
{
    static void Main()
    {
        SomeClass obj = new SomeClass();
        string s="Old String";
        obj.Change(s);
        Console.WriteLine("s=" + s);
    }
}
```

Thân của lớp Phương thức

↪ Tham số tham chiếu (tham chiếu):

- Khi gọi phương thức có tham số tham chiếu, thì tham số thực và tham số hình thức đều chỉ đến cùng ô nhớ (bất kỳ thay đổi dữ liệu của tham số trong phương thức cũng ảnh hưởng đến dữ liệu ban đầu được lưu trong tham số thực)
- Thêm modifier: **ref** hay **out**

Thân của lớp Phương thức

```
class ClassName
{
    void Square1 (int x)
    {...}

    void Square2 (ref int x)
    {...}

    void Square3 (out int x)

}
```


Thân của lớp Phương thức

➤ Phương thức với tham số ref

↪ Cú pháp:

```
class ClassName
{
    Method(ref type variableName)
    {
        ...
    }
}
```

Thân của lớp Phương thức

↪ Cú pháp: sử dụng phương thức có tham số ref

```
variableName = value;  
Method(ref variableName);
```

- Chú ý
 - Tham số thực phải được khởi tạo trước khi gọi phương thức.
 - Mọi thay đổi giá trị trong tham số hình thức đều thay đổi giá trị trong tham số thực.

Thân của lớp Phương thức

➤ Tham chiếu với kiểu giá trị

```
class SomeClass
{
    public void HoanVi(ref int x, ref int y)
    {
        int tam = x;
        x = y;
        y = tam;
    }
}
class Program
{
    static void Main()
    {
        SomeClass obj = new SomeClass();
        int x=7, y=9;
        obj.HoanVi(ref x, ref y);
        Console.WriteLine("x={0} y={1}", x, y);
    }
}
```

Thân của lớp Phương thức

➤ Tham chiếu với kiểu tham chiếu

```
class SomeClass
{
    public void HoanViChuoi(ref string s1, ref string s2)
    {
        string s = s1;
        s1 = s2;
        s2 = s;
    }
}

class Program
{
    static void Main()
    {
        SomeClass obj = new SomeClass();
        string x="ABC", y="XYZ";
        obj.HoanViChuoi(ref x, ref y);
        Console.WriteLine("x={0} y={1}", x, y );
    }
}
```

Thân của lớp Phương thức

➤ Phương thức với tham số out

↪ Cú pháp

```
class ClassName ()  
{  
    Method(out type variableName)  
    {  
        ...  
        variableName = ...;  
    }  
}
```

Thân của lớp Phương thức

↪ Cú pháp: sử dụng phương thức có tham số out

```
//variableName = value;  
Method(out variableName);
```

- Chú ý
 - Tham số thực không nhất thiết phải được khởi tạo trước khi gọi phương thức.
 - Giá trị của tham số thực không được chuyển đến tham số hình thức, vì vậy không được sử dụng tham số hình thức trong phương thức nếu chưa khởi tạo.
 - Tham số hình thức trong phương thức phải được gán giá trị trước khi phương thức kết thúc

Thân của lớp Phương thức

```
class AnotherClass
{
    public int ID;
}

class SomeClass
{
    public void Change1(out AnotherClass ref1)
    {
        ref1.ID = ref1.ID * 2;
    }
    public void Change2(out AnotherClass ref1)
    {
        ref1.ID = 4;
    }
}
```

Thân của lớp Phương thức

```
public void Change3(out AnotherClass ref1)
{
    int x = ref1.ID;
    ref1 = new AnotherClass();
    ref1.ID = x * 2;
}
public void Change4(out AnotherClass ref1)
{
    ref1 = new AnotherClass();
    ref1.ID = 99;
}
}
```


Thân của lớp Phương thức

- Nhận xét: Nếu chúng ta
 - ↳ Chỉ muốn truyền giá trị vào cho phương thức
 - Tham trị
 - ↳ Chỉ muốn nhận 1 giá trị trả về từ phương thức
 - Dùng lệnh return
 - ↳ Chỉ muốn nhận nhiều giá trị trả về từ phương thức
 - Tham chiếu out
 - ↳ Vừa muốn truyền giá trị vào phương thức vừa cho phép thay đổi giá trị đó
 - Tham chiếu ref

Thân của lớp

Phương thức

- Phương thức với tham số params
 - Tham số params cho phép chúng ta tạo ra phương thức có số lượng không xác định các tham số
 - Cú pháp

```
class ClassName
{
    Method(..., params type[] variableName)
    {
        ...
    }
}
```

Thân của lớp Phương thức

```
class SomeClass
{
    public int TinhTong(params int[] arr)
    {
        int sum = 0;
        for (int i=0; i<arr.Length; i++)
            sum += arr[i];
        return sum;
    }
}

class Program
{
    static void Main()
    {
        SomeClass obj = new SomeClass();
        int sum = obj.TinhTong(5,2,3);
    }
}
```

Thân của lớp Phương thức

- Chú ý:
 - Khi dùng từ khóa params thì mảng theo sau params là tham số cuối cùng của phương thức.
 - Cũng có thể truyền 1 mảng cho phương thức

Thân của lớp

Method overloading

- Một phương thức được xác định duy nhất bởi **Chữ ký của phương thức**. Chữ ký phương thức thông thường gồm: Tên phương thức, số lượng và kiểu của tham số, kiểu trả về
- Trong C#, chữ ký của phương thức được xác định bởi: Tên phương thức, số lượng và kiểu của tham số
- Dựa trên chữ ký của phương thức, C# thực hiện chức năng **method overloading**: Cho chúng ta tạo ra các phương thức trùng tên nhưng khác nhau về danh sách tham số.

Thân của lớp

Method overloading

➤ Khác kiểu tham số

```
class SomeClass
{
    public void WriteData(string data)
    {
        Console.WriteLine(data);
    }

    public void WriteData(int resource)
    {
        Console.WriteLine(resource);
    }
}
```

Thân của lớp

Method overloading

- Khác về thứ tự các kiểu tham số

```
class SomeClass
{
    public void WriteData(string data, int resource)
    {
        Console.WriteLine(data + " " + resource);
    }
    public void WriteData(int resource, string data)
    {
        Console.WriteLine(resource + " " + data);
    }
}
```

Thân của lớp

Method overloading

➤ Khác nhau về Kiểu trả về và access modifier

```
class SomeClass
{
    public void WriteData(string data)
    {
        Console.WriteLine(data);
    }
    public int WriteData(string data)
    {
        Console.WriteLine(data);
        return data.Length;
    }
    protected void WriteData(string data)
    {
        Console.WriteLine(data);
    }
}
```


Thân của lớp

Method overloading

- Khác nhau giữa **ref** hay **out**

```
class SomeClass
{
    public void WriteData(string data)
    {
        Console.WriteLine(data);
    }
    public void WriteData(ref string data)
    {
        Console.WriteLine(data);
    }
}
```

Thân của lớp

Method overloading

- Khác nhau giữa ref và out

```
class SomeClass
{
    public void WriteData(ref string data)
    {
        Console.WriteLine(data);
    }
    public void WriteData(out string data)
    {
        data = "Empty";
        Console.WriteLine(data);
    }
}
```

Thân của lớp

Phương thức Toán tử

➤ Operator Overloading

↪ Toán tử 1 ngôi

+	-	!	~
++	--	true	false

↪ Toán tử 2 ngôi

+	-	*	/	%	&
	^	<<	>>	==	!=
>	<	>=	<=		

Thân của lớp

Phương thức Toán tử

➤ Cú pháp

```
public static retval operator op (object1 [,object2])
```

- Quy tắc tạo phương thức toán tử
 - Tất cả các phương thức toán tử được định nghĩa: public và static
 - Kiểu trả về là bất kỳ kiểu nào (thông thường là kiểu mà phương thức đang định nghĩa. Ngoại lệ, toán tử true, false luôn trả về kiểu bool)
 - Số lượng tham số phụ thuộc vào số ngôi của toán tử
 - Nếu toán tử 1 ngôi, tham số của phương thức có kiểu là lớp đang định nghĩa toán tử

Thân của lớp

Phương thức Toán tử

➤ Cú pháp

```
public static retval operator op (object1 [,object2])
```

- Nếu toán tử 2 ngôi, tham số đầu tiên của phương thức có kiểu là lớp đang định nghĩa
- Khi overload toán tử 2 ngôi, phép gán kết hợp của toán tử đó được tự động overload
- Toán tử so sánh (>, <, >=, <=, ==, !=) phải overload từng cặp
- Nếu toán tử == và != được overload → phải overload phương thức Equals() và GetHashCode()



Thân của lớp Phương thức Toán tử

```
using System;

public class Distance
{
    int longitude, latitude;
    public Distance()
    {
        longitude = 0;
        latitude = 0;
    }

    public Distance(int longitude, int latitude)
    {
        this.longitude = longitude;
        this.latitude = latitude;
    }

    public static Distance operator - (Distance first, Distance second)
    {
        return new Distance(first.longitude - second.longitude,
                             first.latitude - second.latitude);
    }
}
```

Thân của lớp

Phương thức Constructor

➤ Constructor

↪ Cú pháp

```
class ClassName
{
    public ClassName (...)
    { ... }
}
```

- Chú ý

- Constructor được gọi tự động khi một instance của lớp được tạo. Không thể gọi phương thức constructor rõ ràng.
- Các constructor có thể được đa năng hóa để cung cấp sự đa dạng cho việc khởi tạo đối tượng.
- Phương thức constructor có thể gọi các phương thức constructor khác

Thân của lớp

Phương thức Destructor

➤ Destructor

↪ Cú pháp

```
class ClassName
{
    public ~ClassName ()
    { ... }
}
```

- Chú ý
 - Không có bất kỳ tham số nào
 - Được gọi bởi Garbage Collector - GC

Đối tượng

➤ Tạo đối tượng với từ khóa new

↪ Khai báo một biến class

- Cú pháp

```
<ClassName> <ObjectName>;
```

- Tạo một đối tượng

– Cú pháp

```
<ObjectName> = new <ClassName> (...);  
<ClassName> <ObjectName> = new <ClassName> (...);
```

Đối tượng

C++

```
CMyClass myClass;  
CMyClass myClass = new CMyClass();
```

C#

```
CMyClass myClass;  
CMyClass myClass = new CMyClass();
```

Đối tượng

➤ Truy cập thành viên public

↪ Cú pháp

```
ObjectName.Method(params) ;  
ObjectName.Field;
```

- Trong C# cũng có con trỏ this giống C++
 - Cú pháp

```
this.member
```

Thân của lớp

Thành viên tĩnh

- Thành viên tĩnh (method, property, field, event)
 - Các thành viên tĩnh có thể gọi trên một lớp ngay cả khi không có instance nào được tạo ra
 - Không thể dùng các instance để truy cập các thành viên tĩnh
 - Chỉ có một bản sao của field tĩnh và event tĩnh tồn tại
 - Các method tĩnh và property tĩnh chỉ có thể truy cập các field tĩnh và event tĩnh

Thân của lớp

Thành viên tĩnh

↪ Cú pháp

tĩnh

```
class ClassName
{
    public static type variableName;
    public static type MethodName (...)
    {
        ...
    }
    public static type PropertyName
    {
        get {...}
        set {...}
    }
    public static event EventType EventName;
}
```

Thân của lớp

Thành viên tĩnh

➡ Cú pháp: Gọi field và phương thức tĩnh

```
ClassName.variableName;  
ClassName.MethodName (...);  
ClassName.PropertyName;  
ClassName.EventName;
```

- Chú ý
 - Nếu không khai báo public thì thành viên tĩnh chỉ được dùng cho các phương thức của lớp.
 - Nếu phương thức gọi thành viên tĩnh trong cùng lớp thì dùng tên thành viên tĩnh không cần thông qua tên lớp

Thân của lớp Field hằng

➤ Field hằng

↪ Field hằng là một field có giá trị không thay đổi trong chu kỳ sống của đối tượng

↪ Ví dụ: `const int x = 5;`

↪ Quy tắc:

- Field hằng là field có giá trị được thiết lập lúc biên dịch
- Field hằng mặc nhiên là tĩnh

Thân của lớp Field Read-Only

➤ Field Read-Only

- ↪ Giống như field hằng nhưng giá trị có thể được trì hoãn cho đến khi chương trình **bắt đầu chạy** (khi đối tượng của lớp chứa field được tạo)
- ↪ Giá trị của field được khởi tạo trong constructor hay khi khai báo

```
class CaptureApp
{
    public readonly uint data = (uint)DateTime.Now.Ticks;
    public readonly int screenColor;
    public CaptureApp()
    {
        screenColor=65536;
    }
}
```


Thân của lớp Constructor tĩnh

➤ Constructor tĩnh

↪ *Constructor tĩnh là constructor được dùng để khởi tạo tất kỳ dữ liệu static nào hay thực hiện một hành động chỉ thực hiện một lần*

```
class CaptureApp
{
    public readonly int screenColor;

    static CaptureApp()
    {
        screenColor=65536;
    }
}
```

Thân của lớp Constructor tĩnh

- ➡ Quy tắc của constructor tĩnh
- Chỉ có duy nhất một constructor tĩnh
 - Không có tham số hay access modifier
 - Không thể truy cập thành viên không tĩnh (kể cả con trỏ this)
 - Được gọi trước khi instance đầu tiên được tạo ra hay các thành viên static được dùng

Định nghĩa thừa kế

➤ Thừa kế

⇒ Thừa kế cho chúng ta tạo ra lớp mới bằng cách tái sử dụng, mở rộng và chỉnh sửa hành vi đã được định nghĩa trong lớp khác.

⇒ Cú pháp

```
class DerivedClass : BaseClass  
{  
    ...  
}
```

■ Đơn thừa kế

- C# không hỗ trợ đa thừa kế thông qua dẫn xuất
- Tập hợp các đặc tính hành vi của các thực thể được hiện thực bằng đa thừa kế giao diện

Gọi Constructor

- Mặc nhiên, Constructor không tham số của lớp cơ sở được gọi trước constructor của lớp dẫn xuất
- Bộ khởi tạo giúp chúng ta quyết định lớp nào và constructor nào muốn gọi.

↳ `base(...)`

↳ `this(...)`

```
class ClassName:BaseClass
{
    public ClassName(type obj, ...) :base (...)
    {...}

    public ClassName(type obj, ...) :this (...)
    {...}
}
```

Phương thức “*new*”

- Phương thức trong lớp cơ sở và lớp dẫn xuất có thể trùng tên

```
class Token
{
    :
    public string Name() { ... }
}
class IdentifierToken : Token
{
    :
    public string Name() { ... }
}
```

- Trình biên dịch sinh ra warning message cảnh báo *IdentifierToken.Name* ẩn *Token.Name*

Phương thức “*new*”

- *new* được dùng để ẩn đi thành viên được thừa kế từ lớp cơ sở
- Một phương thức không thể đồng thời có *new* và *override*

```
class Token
{
    :
    public string Name() { ... }
}
class IdentifierToken : Token
{
    :
    new public string Name() { ... }
}
```

Phương thức “*new*”

```
static void Method(Token t)
{
    Console.WriteLine(t.Name());
}
static void Main()
{
    IdentifierToken variable = new IdentifierToken("variable");
    Method(variable);
}
```

Phương thức “*virtual*”

- Phương thức *virtual* là phương thức cho phép các lớp con hiện thực lại tùy theo chức năng của lớp con
- *Virtual* không thể được dùng với *static* và *override*

```
class Token
{
    :
    public virtual string Name ()
    { ... }
}
```

Đây là phiên bản hiện thực đầu tiên của phương thức *Name()*

Phương thức “*virtual*”

- *virtual* được dùng để định nghĩa phương thức hỗ trợ đa hình
- Các lớp con tự lo hiện thực phiên bản mới của phương thức *virtual* đã định nghĩa trong lớp cha bằng cách dùng từ khóa *override*

Phương thức “*override*”

- Một phương thức override cung cấp một thực thi mới cho phương thức của lớp cơ sở. Phương thức lớp cơ sở nên khai báo là **virtual**
- Access modifier của phương thức cơ sở không thể bị thay đổi bởi phương thức override nó
- Từ khóa *new*, *static*, *virtual* không thể được dùng cùng *override*

Phương thức “*override*”

```
class IdentifierToken : Token
{
    :
    public override string Name ()
    { ... }
}
```

Hiện thực khác của
phương thức Name()

Phương thức “*override*”

- Phương thức private không thể là virtual hay override
- Hai phương thức phải: cùng tên, cùng kiểu và số lượng tham số, cùng kiểu trả về
- Hai phương thức phải cùng kiểu truy cập
- Chỉ có thể override phương thức virtual
- override ngầm hiểu là virtual → có thể override ở những phương thức con tiếp theo

Phương thức “sealed”

- Từ khóa sealed chỉ ra phương thức không được override trong lớp con của nó

```
class X
{
    public void method1 ()
    { }
    virtual public void method2 ()
    { }
}
class Y : X
{
    sealed override public void
method2 ()
    { }
}
class Z : Y
{
    override public void method2 ()
    { }
}
```

Đa hình

- Để thực hiện tính đa hình chúng ta thực hiện các bước sau:
 - ↪ Phương thức trong lớp cha là phương thức virtual, override hay abstract
 - ↪ Phương thức trong lớp con là phương thức override

Đa hình

➤ Cú pháp

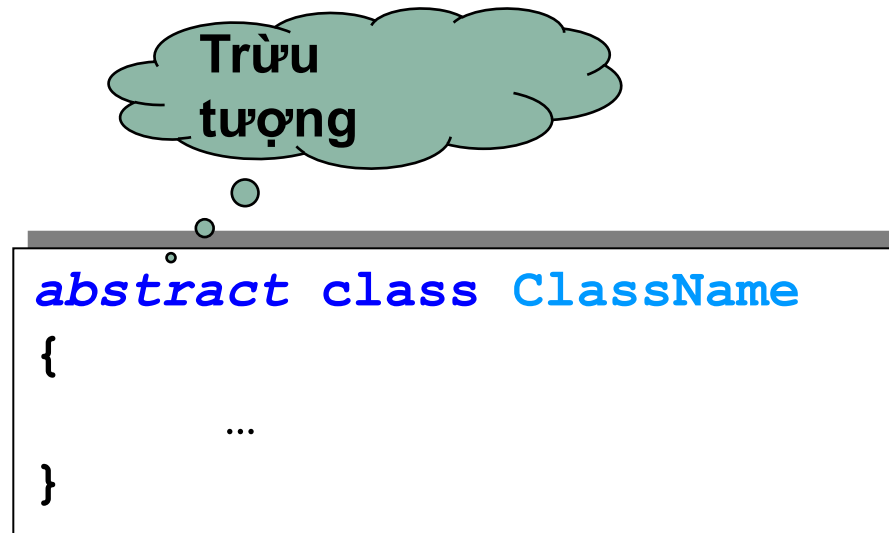
```
class ClassName1
{
    1 → virtual public type Method(...)
    {
    }
}

class ClassName2: ClassName1
{
    2 → override public type Method(...)
    {
    }
}
```

Lớp sealed và abstract

- Lớp trừu tượng
 - ↪ Lớp trừu tượng là lớp được khai báo để làm lớp cơ sở cho lớp khác.

- Cú pháp



```
abstract class ClassName  
{  
    ...  
}
```


Lớp sealed và abstract

- Chú ý:
 - ↪ Lớp trừu tượng không cho tạo đối tượng
 - ↪ Khai báo phương thức trừu tượng chỉ trong lớp trừu tượng
 - ↪ Thành viên trừu tượng không thể là static
 - ↪ Phương thức của Lớp trừu tượng không thể **không thể private**
 - ↪ Phương thức trừu tượng không thể có modifier virtual

Lớp sealed và abstract

- Cú pháp: phương thức trừu tượng

```
modifier abstract <type>AbstractMethodName () ;
```

- Hiện thực lớp trừu tượng

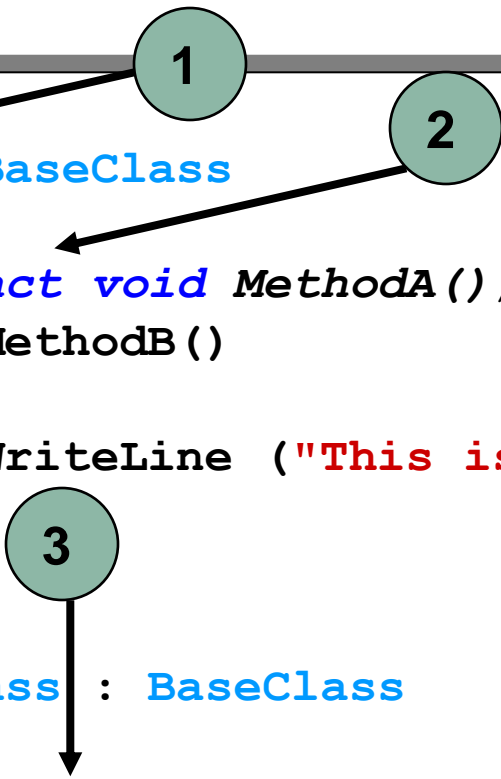


Trừu
tượng

```
class ClassName:AbstractClassName
{
    ...

    modifier <type> AbstractMethodName ()
    { ... }
}
```

Lớp sealed và abstract



```
using System:
abstract class BaseClass
{
    public abstract void MethodA();
    public void MethodB()
    {
        Console.WriteLine ("This is the non abstract method");
    }
}

class DerivedClass : BaseClass
{
    public override void MethodA()
    {
        Console.WriteLine ("This is the abstract method
        overridden in derived class");
    }
}
```

Annotation 1 points to `using System;`
Annotation 2 points to `abstract class BaseClass`
Annotation 3 points to `class DerivedClass : BaseClass`

Lớp sealed và abstract

```
class AbstractDemo
{
    public static void Main()
    {
        DerivedClass objDerived = new DerivedClass();
        BaseClass objBase = objDerived;
        objBase.MethodA();
        objDerived.MethodB();
    }
}
```

Lớp sealed và abstract

➤ Lớp sealed

- Là một lớp không bao giờ dùng làm lớp cơ sở
- Lớp abstract không thể được dùng như là một lớp sealed

```
sealed class Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int x;
    public int y;
}
```

Lớp static

➤ Lớp static

↪ Lớp static là lớp chỉ chứa các thành viên static, không thể tạo instance của lớp static và lớp static được nạp tự động bởi CLR

↪ Đặc điểm của lớp static

- Chỉ chứa thành viên static
- Không thể tạo đối tượng
- Là lớp sealed
- Không chứa các instance constructor

Lớp static

```
static class CompanyInfo
{
    public static string GetCompanyName ()
    {
        return "CompanyName" ;
    }
    public static string GetCompanyAddress ()
    {
        return "CompanyAddress" ;
    }
    // ...
}
```

Bài tập

- **Bài 1.** Tạo lớp Điểm, có 2 thuộc tính tọa độ x, tọa độ y và phương thức tính khoảng cách đến 1 điểm khác.
- **Bài 2.** Kế thừa lớp Điểm, tạo ra lớp Tam giác, có 2 thuộc tính là điểm 2, điểm 3. Phương thức tính chu vi, diện tích tam giác, phương thức xét tam giác đó là tam gì.
- **Bài 3.** Tạo lớp Sinh viên có các thuộc tính MaSV, TenSV, QueQuan, phương thức nhập n sinh viên, phương thức hiển thị n sinh viên theo cột, bảng.

Bài tập

- **Bài 1.** Tạo 1 lớp HH có các thuộc tính Mã hàng, tên hàng, đơn giá, tạo phương thức hiển thị mặt hàng trên.
- **Bài 2** Tạo lớp HoaDon kế thừa lớp HH, có thêm thuộc tính tên khách hàng, tạo phương thức nhập số lượng mặt hàng có trong hóa đơn, phương thức hiển thị các mặt hàng trong hóa đơn.
- **Bài 3.** Tạo lớp HCN có các thuộc tính chiều dài, chiều rộng, phương thức tính chu vi, tính diện tích.

Bài tập

- Bạn hãy xây dựng lớp có tên là `clsSinhVien` có các thuộc tính: Mã sinh viên, tên sinh viên, số điện thoại, quê quán, ghi chú, phương thức khởi dựng không tham số (constructor) và phương thức khởi dựng có tham số, phương thức in 1 thông tin sinh viên ra màn hình, phương thức in n sinh viên ra màn hình.
- Xây dựng lớp `HangHoa`, `HoaDon`.