KẾ THỪA

Kế thừa là gì?

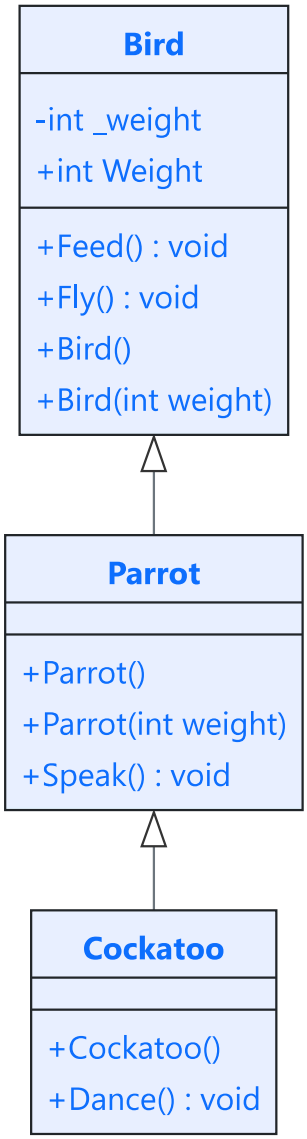
Kế thừa (inheritance) là một công cụ rất mạnh trong lập trình hướng đối tượng cho phép tạo ra các class mới từ một class đã có, và qua đó cho phép tái sử dụng code của class đã có, giúp giảm thiểu việc lặp code giữa các class, dễ dàng bảo trì và giảm thời gian phát triển.

Hãy cùng xem xét ví dụ sau:

```
using System;
namespace P01_Inheritance
{
    internal class Bird
    {
        private int _weight;
        public int Weight
        {
            get => _weight;
            set {
                if (value > 0)
                    _weight = value;
            }
        }
        public void Feed() => _weight += 10;
        public Bird() => Console.WriteLine($"Bird created");
        public Bird(int weight)
        {
            _weight = weight;
            Console.WriteLine($"Bird created, {_weight} gr.");
        }
        public void Fly() => Console.WriteLine("Bird is flying");
    }
    internal class Parrot : Bird
    {
        public Parrot() => Console.WriteLine("Parrot created");
        public Parrot(int weight) : base(weight) { }
        public void Speak() => Console.WriteLine("Parrot is speaking");
    }
    internal class Cockatoo : Parrot
    {
        public Cockatoo() => Console.WriteLine("Cockatoo created");
        public void Dance() => Console.WriteLine("Cockatoo is dancing");
    }
```

```
    internal class MainClass
    {
        private static void Main(string[] args)
        {
            Console.WriteLine("Bird:");
            Bird bird = new Bird(50) { Weight = 100 };
            bird.Feed();
            Console.WriteLine($"Weight: {bird.Weight}");
            bird.Fly();
            Console.WriteLine("rnParrot:");
            Parrot parrot = new Parrot(200);
            parrot.Feed();
            Console.WriteLine($"Weight: {parrot.Weight}");
            parrot.Fly();
            parrot.Speak();
            Console.WriteLine("rnCockatoo:");
            Cockatoo cockatoo = new Cockatoo() { Weight = 300 };
            cockatoo.Feed();
            Console.WriteLine($"Weight: {cockatoo.Weight}");
            cockatoo.Fly();
            cockatoo.Speak();
            cockatoo.Dance();
            Console.ReadKey();
        }
    }
}
```

Mối quan hệ giữa các class được thể hiện qua sơ đồ:



Trong ví dụ trên chúng ta tạo ra ba class: `Bird` (chim), `Parrot` (vẹt) và `Cockatoo` (vẹt châu Úc). `Parrot` kế thừa `Bird`; `Cockatoo` kế thừa `Parrot`. Quan hệ kế thừa này trong C# được thể hiện bằng dấu hai chấm phân chia tên của class mới với tên của một class có sẵn:

1. `class Parrot : Bird`
2. `class Cockatoo : Parrot`

Kết quả chạy chương trình

Khi chạy chương trình trên có thể nhận xét như sau:

Mỗi khi khởi tạo object của class con thì hàm tạo của class cha luôn được gọi trước. Điều này có nghĩa là trước khi khởi tạo object của class con thì object của class cha được khởi tạo, và do đó bản thân object con có chứa trong nó object cha. Object cha này được truy cập từ object con thông qua từ khóa `base`. Thông qua từ khóa `base` cũng có thể truy xuất các thành viên của lớp cha.

Hàm tạo không được kế thừa mà hàm tạo của lớp cha được gọi tự động (nếu là hàm tạo không tham số) hoặc được gọi từ hàm tạo của lớp con (nếu là hàm tạo có tham số). Hàm tạo của lớp cha được gọi bằng lệnh base(), tương tự như gọi phương thức (chỉ thay tên phương thức bằng từ khóa base).

Mặc dù lớp con không thể kế thừa thành viên `private` của lớp cha (tức là không thấy và không thể trực tiếp sử dụng, như lớp Parrot không thể nhìn thấy và trực tiếp sử dụng biến thành viên `_weight` của `Bird`) nhưng qua phương thức/thuộc tính kế thừa của lớp cha vẫn có thể gián tiếp sử dụng thành viên `private` này. Phương thức `Feed` và thuộc tính `Weight` ở trên là ví dụ. Lý do là vì trong object con có cả object cha tồn tại. Lời gọi tới phương thức kế thừa từ lớp cha thực chất là hoạt động với object cha này.

Đặc điểm của kế thừa

Lớp có sẵn mà từ đó tạo ra các lớp khác được gọi là *lớp cha/lớp cơ sở*; lớp mới xây dựng trên cơ sở lớp cũ được gọi là *lớp con/lớp dẫn xuất*.

Để tiện lợi, trong một số trường hợp chúng ta sẽ sử dụng thuật ngữ tiếng Anh thay thế: base class, parent class, super class (lớp cha/lớp cơ sở), derived class, child class, subclass (lớp con, lớp dẫn xuất).

C# chỉ cho phép mỗi lớp con có một lớp cha trực tiếp (khác với C++ cho phép lớp có nhiều lớp cha trực tiếp). Cách kế thừa này được gọi là *kế thừa đơn* (single inheritance).

Lớp con, đến lượt mình, lại có thể trở thành lớp cơ sở cho các lớp khác (tạm gọi vui là lớp cháu J). Quá trình này có thể tiếp diễn với nhiều thế hệ lớp khác nhau, tạo ra một *cấu trúc phân cấp* (class hierarchy) của các class có quan hệ kế thừa nhau.

Tất cả các lớp con, cháu, chắt, v.v. của một class gọi chung là các *lớp hậu duệ* (descendant) của class đó; các lớp cha, ông, cụ, v.v. của một class được gọi chung là các *lớp tiền bối* (ancestor) của nó.

Trong ví dụ trên, `Bird` là lớp cha của `Parrot` (`Parrot` là lớp con trực tiếp của `Bird`), còn `Parrot` lại trở thành lớp cha của `Cockatoo` (`Cockatoo` là lớp con trực tiếp của `Parrot`, lớp con gián tiếp của `Bird`). `Bird` ➡️ `Parrot` ➡️ `Cockatoo` tạo ra một cấu trúc phân cấp của các class. `Parrot`, `Cockatoo` đều có thể gọi chung là các lớp hậu duệ của `Bird`.

Class được đánh dấu với từ khóa `sealed` không cho phép kế thừa. Hiểu đơn giản dòng dõi class này đến đây là "tuyệt tự" 🙄

Khi một class kế thừa từ một class khác, nó thừa hưởng tất cả các thành viên của class cha (kể cả những thành viên mà cha nó kế thừa từ ông), trừ những thành viên được đánh dấu là private, hàm tạo, hàm hủy.

Trong ví dụ trên, `Parrot` thừa hưởng thuộc tính `Weight`, phương thức `Feed` và `Fly` của `Bird`. `Cockatoo` sẽ thừa hưởng `Weight`, `Feed` và `Fly` (từ `Bird`), đồng thời thừa hưởng phương thức `Speak` từ `Parrot`.

Tuy nhiên, `Parrot` (và `Cockatoo`) lại không kế thừa được biến thành viên `_weight` từ `Bird` vì biến này để mức truy cập là `private`, cũng như không thể kế thừa các hàm tạo `Bird()` và `Bird(int)`. Tương tự, `Cockatoo` cũng không kế thừa được hàm tạo `Parrot()` và `Parrot(int)`.

Lớp Object và kế thừa

Toàn bộ .NET framework được xây dựng dựa trên khái niệm "**tất cả đều là object**", vốn hoạt động trên cơ sở kế thừa.

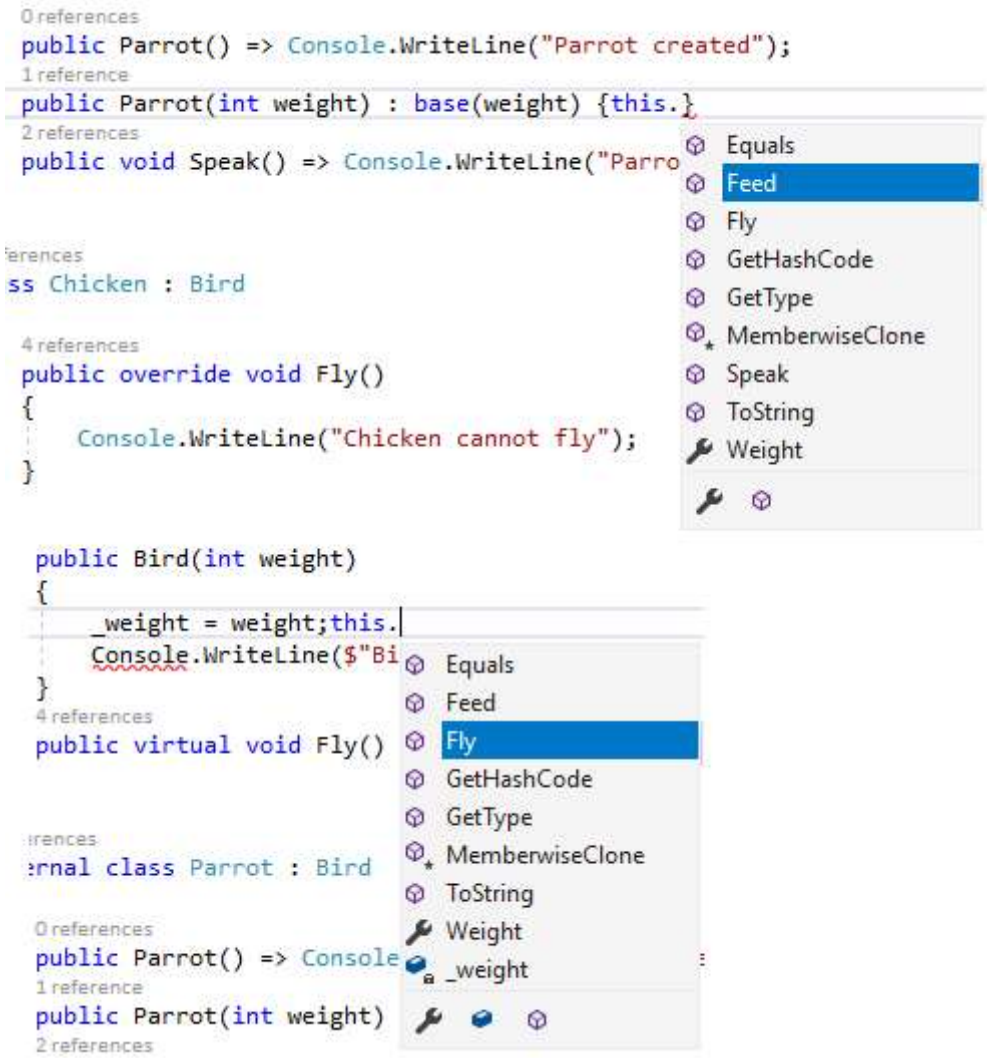
Trong C#, mọi class đều là hậu duệ của lớp `System.Object`, kể cả khi không ghi quan hệ kế thừa với lớp này.

Một số nguyên, số thực, biến logic, v.v. trong C# đều là object của các class tương ứng (`Int32`, `Double`, `Boolean`) kế thừa từ lớp `System.Object`. Tuy nhiên, C# hỗ trợ để có thể, ví dụ, gán giá trị số trực tiếp, thay thì phải khởi tạo object của lớp số nguyên tương ứng.

Bởi vì mọi class trong C# đều kế thừa (trực tiếp hoặc gián tiếp) từ lớp System.Object, bất kỳ class nào xây dựng xong đều có sẵn 4 phương thức: ToString, GetHashCode, GetType, Equals. Đây là bốn phương thức của lớp Object. Như vậy, kế thừa là một cơ chế tái sử dụng code để mở rộng ứng dụng.

Trong ví dụ trên, `Bird`, `Parrot`, `Cockatoo` đều là các hậu duệ của lớp `System.Object`, vì vậy các class này đều thừa hưởng 4 phương thức kể trên.

Để xem một class có những thành viên nào có thể dùng từ khóa `this` ở bên trong thân bất kỳ phương thức nào như sau:



Từ khóa `this` sử dụng trong phương thức của class

Như vậy, chúng ta thấy rằng, cơ chế kế thừa cho phép tái sử dụng code từ những class sẵn có để tạo ra class mới một cách nhanh chóng. Mỗi class con là một bản mở rộng của class cha bằng cách thêm vào những thành viên của riêng mình.

Quan hệ giữa kế thừa và đa hình

Trong lập trình hướng đối tượng, kế thừa và đa hình là hai nguyên lý khác nhau.

- *Đa hình* thiết lập mối quan hệ "là" (is-a relationship) giữa kiểu cơ sở và kiểu dẫn xuất. Ví dụ, nếu chúng ta có lớp cơ sở `Bird` và lớp dẫn xuất `Parrot` thì một object của `Parot` cũng là object của `Bird`, kiểu `Parrot` cũng là kiểu `Bird` (đương nhiên rồi, vẹt là chim mà!). Mối quan hệ này nhìn rất giống như quan hệ kế thừa ở trên.

- Trong khi đó, *kế thừa* liên quan chủ yếu đến tái sử dụng code: code của lớp con thừa hưởng code của lớp cha. Một cách nói khác, đa hình liên quan tới quan hệ về ngữ nghĩa, còn kế thừa liên quan tới cú pháp.

Trong các ngôn ngữ như C++, C#, Java, hai khái niệm này hầu như được đồng nhất, thể hiện ở chỗ:

1. class con thừa hưởng các thành viên của class cha (kế thừa, tái sử dụng code);
2. một object thuộc kiểu con có thể gán cho biến thuộc kiểu cha, tức là kiểu cơ sở có thể dùng để thay thế cho kiểu dẫn xuất (đa hình).

Vì những lý do trên mà các lệnh khai báo và khởi tạo sau là hoàn toàn đúng:

1. `Bird parrotTheBird = new Parrot();`
2. `Parrot cockatooTheParrot = new Cockatoo();`
3. `Bird cockatooTheBird = new Cockatoo();`

Lệnh thứ nhất khai báo một object thuộc kiểu `Bird` nhưng được gán một object thuộc kiểu `Parrot`. **Kết quả thực hiện sẽ là:**

Lệnh thứ hai khai báo biến thuộc kiểu `Parrot` nhưng được gán object thuộc kiểu `Cockatoo`. **Kết quả thực hiện sẽ là:**

Lệnh thứ ba khai báo biến thuộc kiểu `Bird` nhưng được gán object thuộc kiểu `Cockatoo`. **Kết quả thực hiện sẽ là:**

Chúng ta cũng có thể để ý thấy rằng, các object của lớp con thực sự được khởi tạo (các hàm tạo được gọi theo trật tự giống như đã gặp ở bài trước). Nhưng các object này lại được tham chiếu tới từ các biến thuộc kiểu cha.

Trong những trường hợp này, object chỉ có thể sử dụng được những thành viên của lớp cha. Ví dụ, object `parrotTheBird` ở trên chỉ có thể sử dụng các thành viên của lớp `Bird` mà không biết về các thành viên mới của `Parrot` (như phương thức `Speak`).

Cơ chế quan hệ này kết hợp với *ghi đè* (overriding) và *che giấu* (hiding) cung cấp cho người lập trình công cụ đặc biệt mạnh.

Che giấu phương thức (method hiding)

Như trên đã phân tích, class con thừa hưởng tất cả các thành viên mà lớp cha cho phép. Vậy điều gì xảy ra nếu trong lớp con chúng ta định nghĩa một phương thức trùng với phương thức nó kế thừa từ lớp cha?

Hãy tưởng tượng bây giờ chúng ta xây dựng lớp `Chicken` kế thừa từ `Bird` như sau:

```
class Chicken : Bird
{
    public void Fly()
    {
        Console.WriteLine("Chicken cannot fly");
    }
}
```

Vì `Chicken` kế thừa `Bird`, nó cũng thừa hưởng phương thức `Fly` của `Bird`. Nhưng trong lớp `Chicken` lại định nghĩa một phương thức `Fly` với mô tả giống hệt `Fly` của `Bird`.

Nếu để ý trong trình soạn thảo code, Intellisense của Visual Studio hiển thị "Cảnh báo của Visual Studio về che giấu phương thức"

Thông báo này có ý nghĩa là phương thức `Fly` của lớp `Chicken` sẽ che đi phương thức `Fly` của lớp `Bird`.

Trong những tình huống tương tự, C# tự động áp dụng cơ chế *che giấu phương thức* (method hiding).

Để đảm bảo đây đúng là hành động mà người lập trình mong muốn, C# yêu cầu phải ghi rõ từ khóa `new` trước khai báo phương thức như sau:

```
public new void Fly()
{
    Console.WriteLine("Chicken cannot fly");
}
```

Trong tình huống này cả hai phương thức `Fly` đều cùng tồn tại trong object (của class con) nhưng phụ thuộc vào loại biến tham chiếu tới (biến chứa địa chỉ) object (biến khai báo thuộc kiểu con hay biến khai báo thuộc kiểu cha) sẽ quyết định sử dụng phương thức nào:

1. Nếu biến chứa địa chỉ object được khai báo là kiểu cha, phương thức `Fly` của object cha sẽ được gọi;
2. Nếu biến chứa địa chỉ object được khai báo là kiểu con, phương thức `Fly` của object con sẽ được gọi.

Trong ví dụ trên, nếu khai báo và khởi tạo như sau:

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

Kết quả thực hiện sẽ là:

Nếu khai báo và khởi tạo như sau:

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

Kết quả thực hiện sẽ là:

Như vậy trong trường hợp này, phương thức Fly định nghĩa ở lớp con sẽ đơn giản là "che" phương thức Fly mà nó kế thừa từ lớp cha. Cả hai cùng tồn tại trong cùng một object. Kiểu của biến sẽ quyết định phương thức nào được gọi.

Ghi đè phương thức (method overriding)

Bây giờ hãy điều chỉnh lớp `Bird`, phương thức `Fly` như sau (thêm từ khóa `virtual` vào trước khai báo phương thức):

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

Thay đổi phương thức `Fly` của lớp `Chicken` như sau (đổi từ khóa `new` thành `override`);

```
public override void Fly()
{
    Console.WriteLine("Chicken cannot fly");
}
```

Đoạn code:

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

Kết quả thực hiện sẽ là:

Đoạn code:

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

Kết quả thực hiện sẽ là:

Hai kết quả này giống nhau. Vậy điều gì đã xảy ra?

Đây là kết quả hoạt động của cơ chế *ghi đè* (overring) phương thức.

Trong cơ chế ghi đè, phương thức `Fly` của lớp cha (mà `Chicken` kế thừa) sẽ bị xóa bỏ và thay thế bằng phương thức `Fly` mới định nghĩa trong lớp `Chicken`. Nói cách khác, trong object tạo ra từ `Chicken` giờ đây chỉ còn một phương thức `Fly` duy nhất. Do đó, bất kể biến tham chiếu tới nó được khai báo là kiểu gì thì cũng chỉ truy xuất được phương thức `Fly` này.

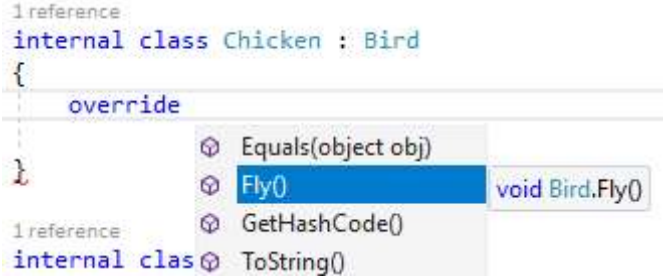
Để áp dụng được cơ chế ghi đè, cả lớp cha và lớp con cần phải phối hợp:

1. lớp cha phải cho phép phương thức được phép ghi đè bằng cách thêm từ khóa `virtual` trước khai báo phương thức;
2. lớp con phải thông báo rõ việc ghi đè bằng cách thêm từ khóa `override` trước định nghĩa phương thức.

Mặc định các phương thức của class không cho ghi đè mà chỉ cho phép che giấu.

Tuy nhiên, các phương thức `Equals`, `GetHashCode`, `ToString` của lớp tổ tiên `System.Object` đều cho phép ghi đè ở lớp hậu duệ.

Để xác định những phương thức nào cho phép ghi đè, chỉ cần viết từ khóa `override` trong thân class (bên ngoài phương thức).



Che dấu được sử dụng chủ yếu để đảm bảo tương thích ngược giữa các class. Cơ chế này không được sử dụng nhiều trong thực tế.

Ở phía khác, ghi đè được sử dụng rất phổ biến cùng với đa hình giúp tạo ra một class đại diện cho các biến thể khác nhau.

Lớp trừu tượng và kế thừa

Lớp và trừu tượng hóa

Theo cách suy nghĩ hướng đối tượng, chúng ta phải trừu tượng hóa các đối tượng để tạo ra class. Ví dụ, từ việc phân tích nhiều chiếc bàn cụ thể chúng ta rút ra:

- Những chiếc bàn cụ thể phải có chân, dù là 3 chân, 4 chân hoặc nhiều chân hơn. Như vậy, số chân là một đặc điểm chung của bàn.
- Mỗi chiếc bàn có thể sơn màu trắng, đỏ, vàng, v.v.. Vậy màu sắc cũng là một đặc điểm chung.
- Mỗi chiếc bàn có thể to nhỏ khác nhau nhưng đều có một diện tích mặt để sử dụng. Vậy diện tích bề mặt sử dụng cũng là một đặc điểm của bàn.

Qua phân tích này chúng ta thấy có 3 loại thông tin có thể dùng để mô tả cho một chiếc bàn bất kỳ: số chân, màu sắc, diện tích mặt. Tuy nhiên, mỗi chiếc bàn cụ thể lại không giống nhau, thể hiện ở giá trị cụ thể của số chân, màu sắc và diện tích mặt.

Như vậy, khi chúng ta mô tả "Bàn" bằng ba loại thông tin đại diện như trên, chúng ta đã trừu tượng hóa từ những chiếc bàn cụ thể về một loại thông tin chung mô tả cho bàn. Loại thông tin chung này chính là class, và từng chiếc bàn cụ thể là object. Class, do đó, là dạng trừu tượng hóa, là mô tả chung của các đối tượng cụ thể.

Bây giờ chúng ta lại phân tích tiếp những chiếc ghế và tủ theo cách tương tự và lần lượt thu được các lớp Ghế và Tủ.

Nếu chúng ta tiếp tục phân tích những điểm chung của Bàn, Ghế, và Tủ, chúng ta lại có thể trừu tượng hóa một lần nữa để tạo ra lớp Nội thất.

Tuy nhiên, khi nói đến nội thất, chúng ta lại không thể đưa ra hình dung chính xác của nó. Khác với khi nói đến Bàn chúng ta hình dung đại khái được một chiếc bàn.

Như vậy, Nội thất là một loại trừu tượng hóa cấp độ cao hơn nữa. Nó không cho ra hình dung cụ thể nào mà chỉ có thể được hình dung thông qua các class con cụ thể của nó là Bàn, Ghế, hoặc Tủ. Loại class để mô tả nội thất như vậy trong lập trình hướng đối tượng có tên gọi riêng: lớp trừu tượng (abstract class).

Lớp trừu tượng

Lớp trừu tượng (abstract class) là loại class có mức độ trừu tượng cao dùng làm khuôn mẫu để tạo ra các class khác.

Như vậy, class bình thường là khuôn mẫu để tạo ra object (là những thực thể), còn class trừu tượng lại dùng làm khuôn mẫu để tạo ra class khác. Sự khác biệt này dẫn đến tình huống là lớp trừu tượng không được sử dụng để tạo ra object như class bình thường.

Trong C#, lớp trừu tượng được xây dựng bằng cách thêm từ khóa **abstract** vào trước từ khóa class khi khai báo. Lớp trừu tượng không thể dùng để khởi tạo object mà chỉ đóng vai trò lớp cơ sở để tạo ra các lớp dẫn xuất, là những trường hợp cụ thể hơn.

Ví dụ khai báo lớp trừu tượng `Animal`:

```
abstract class Animal // đây là một lớp trừu tượng
{
}
```

Lớp `Animal` không cho phép tạo object. Do đó, lệnh khởi tạo sau sẽ bị báo lỗi:

```
var animal = new Animal(); // lỗi! Lớp Animal không cho phép khởi tạo object
```

Phương thức trừu tượng

Một điểm rất mạnh của lớp trừu tượng là nó chứa bên trong các *phương thức trừu tượng* (abstract method).

Phương thức trừu tượng (abstract method) là loại phương thức được khai báo trong thân lớp trừu tượng với từ khóa **abstract** và không có thân phương thức. Ví dụ:

```
abstract class Animal // đây là một lớp trừu tượng
{
    public abstract void Eat(); // đây là khai báo phương thức trừu tượng Eat không có thân.
    public abstract void Move(); // khai baos phương thức trừu tượng Move.
}
```

Nếu trong thân một class khai báo một phương thức trừu tượng thì class chứa nó bắt buộc phải khai báo là abstract.

Một class kế thừa từ lớp trừu tượng này bắt buộc phải ghi đè **tất cả** phương thức trừu tượng của class mà nó thừa kế.

```
class Dog : Animal
{
    public override void Eat() => Console.WriteLine("I love bone!");
    public override void Move() => Console.WriteLine("I walk on 4 feet");
}
```

Phương thức trừu tượng mặc định được đánh dấu virtual (cho phép ghi đè) nên bạn không cần (không được) dùng từ khóa virtual trước phương thức abstract nữa.

Nếu không ghi đè đủ các phương thức abstract của lớp cha, lớp con bắt buộc cũng phải đánh dấu là abstract:

```
abstract class Dog : Animal // Dog không ghi đè hết phương thức abstract của Animal nên nó phải đ
{
    public override void Eat() => Console.WriteLine("I love bone!");
}
```

Yêu cầu này làm cho lớp và phương thức trừu tượng trở thành một công cụ rất mạnh: nó tạo ra một "bản hợp đồng" chứa danh sách các phương thức mà tất cả các lớp dẫn xuất bắt buộc phải thực thi.

Nói theo cách khác, lớp trừu tượng được sử dụng làm khuôn mẫu đề tạo ra class khác. Để bắt các class dẫn xuất tuân thủ theo các quy tắc chung, trong lớp trừu tượng sử dụng các phương thức trừu tượng với vai trò hợp đồng. Các class dẫn xuất từ lớp trừu tượng bắt buộc phải tuân thủ hợp đồng khi kế thừa từ lớp trừu tượng bằng cách xây dựng các phương án cụ thể của phương thức trừu tượng.