Các kiểu đặc biệt trong C#: anonymous, nullable, dynamic

Anonymous, nullable và dynamic là những loại kiểu dữ liệu đặc biệt trong C#. Anonymous type là những kiểu dữ liệu không có tên. Nullable type là những kiểu dữ liệu vốn thuộc nhóm value type nhưng giờ có thể nhận thêm giá trị null. Dynamic giúp đưa đặc điểm "định kiểu động" vào C#. Đây là những kiểu dữ liệu đặc thù được đưa vào C# để giải quyết một số vấn đề riêng.

Bài học này sẽ giúp bạn hiểu rõ hơn về các nhóm kiểu dữ liệu này.

NỘI DUNG CỦA BÀI [Ẩn]

Anonymous type trong C#

Khái niệm anonymous type

Anonymous type (Kiểu dữ liệu vô danh) trong C# là loại kiểu dữ liệu tạm thời mà C# compiler tự suy đoán ra cấu trúc khi object của nó được khởi tạo thông qua cú pháp **Object Initializer**.

> Bạn đã làm quen với cấu trúc cú pháp Object Initializer (khởi tạo object thông qua các thuộc tính) trong bài học về cách khởi tạo object.

Anonymous type trong C# chỉ dùng để chứa dữ liệu và dữ liệu của nó là chỉ đọc. Một khi object của anonymous type được khởi tạo thì giá trị của nó không thể thay đổi được nữa. Vì anonymous type không có tên (!), bạn cũng không thể dùng cách thức khai báo biến thông thường cho nó.

> Cách khai báo và khởi tạo biến thông thường: `<tên_kiểu> <tên_biến> = <giá_trị>;`

Vì lý do này, C# đưa vào *từ khóa **var*** giúp khai báo một biến mà không biết tên kiểu dữ liệu của nó. C# compiler sẽ căn cứ vào giá trị gán cho biến đó để tự suy đoán ra kiểu.

> Anonymous type không thực sự là "vô danh". Trên thực tế, C# compiler sẽ tự tạo ra một cái tên cho kiểu dữ liệu nhưng bạn không biết đến cái tên đó. Cái tên này chỉ tồn tại khi compiler dịch mã nguồn.

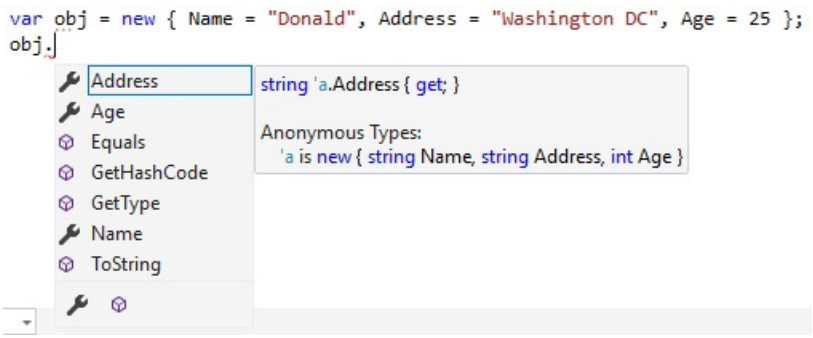
Sử dụng anonymous type trong C#

Hãy cùng thực hiện một ví dụ sử dụng anonymous type (bạn có thể sử dụng C# interactive):

```
 // đây là object của một kiểu vô danh với 3 property: Name, Address, Age
// c# compiler sẽ tự sinh ra class cho object này và tự suy đoán kiểu của các property
var obj = new { Name = "Donald", Address = "Washington DC", Age = 25 };
```

Trong ví dụ này chúng ta sử dụng cú pháp khởi tạo Object Initializer để tạo ra một object mới với 3 thuộc tính Name, Address và Age.

Tuy nhiên, sau từ khóa new lại không có tên class (kiểu dữ liệu) như trong cú pháp khởi tạo object thông thường. Vì không biết tên kiểu nên khai báo biến obj phải dựa vào từ khóa var để C# compiler tự suy đoán ra cấu trúc và tạo một kiểu tạm thời (nhưng người lập trình không biết đến).



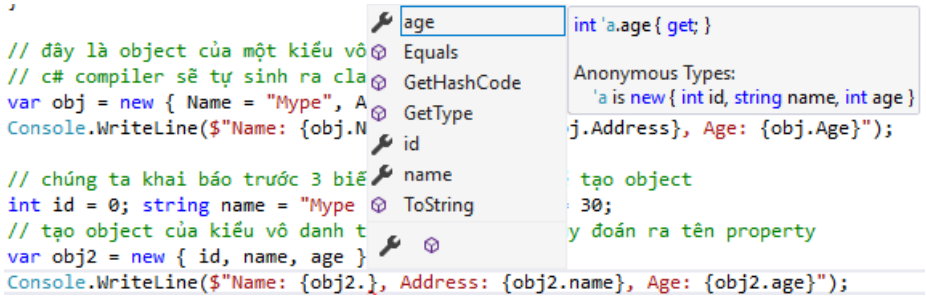
Việc truy xuất phần tử của object vô danh không có gì khác với object bình thường:

```
Console.WriteLine($"Name: {obj.Name}, Address: {obj.Address}, Age: {obj.Age}");
```

Hãy xem một ví dụ khác:

```
 // khai báo trước 3 biến và dùng chúng để tạo object
int id = 0; string name = "Obama"; int age = 30;
// tạo object của kiểu vô danh từ 3 biến, C# tự suy đoán ra tên property
var obj2 = new { id, name, age };
Console.WriteLine($"Name: {obj2.id}, Address: {obj2.name}, Age: {obj2.age}");
```

Trong trường hợp này chúng ta tạo một object thuộc kiểu anonymous type từ các biến khai báo trước. C# thậm chí có thể tự suy đoán ra tên thuộc tính từ tên biến.



C# có thể tự suy đoán ra tên thuộc tính từ tên biến

Anonymous type được sử dụng rất phổ biến trong LINQ với phương thức Select để tạo ra một object mới từ một tổ hợp các thuộc tính của một object khác. Loại kỹ thuật này có tên gọi là object projection.

Nullable type trong C#

Khái niệm nullable type

Trong bài học về các kiểu dữ liệu cơ sở của C#, chúng ta đã nhắc qua rất ngắn gọn về nullable type.

Như bạn đã biết, các kiểu dữ liệu thuộc nhóm value type đều có khoảng giá trị xác định. Ví dụ, `bool` (hay `System.Boolean`) chỉ có thể nhận một trong hai giá trị `true` | `false`. Ngoài ra, C# cũng bắt buộc mọi biến phải được khởi tạo và gán giá trị trước khi sử dụng. Do đó, các biến thuộc kiểu value type luôn luôn có giá trị xác định khi tham gia vào biểu thức.

Giờ hãy tưởng tượng bạn ánh xạ một bản ghi của cơ sở dữ liệu quan hệ sang một object của C#. Mỗi trường của bản ghi đó sẽ tương ứng với một biến thành viên hoặc 1 property của object. Cơ sở dữ liệu quan hệ cho phép một trường nhận giá trị null với ý nghĩa rằng trường đó không có giá trị. Khi đó, giả sử một trường số nguyên của bản ghi nếu có giá trị null, vậy nó sẽ tương ứng với giá trị nào của kiểu số nguyên trong C#? Các kiểu số nguyên của C# không thể biểu diễn trạng thái "*không có giá trị*".

Đối với các kiểu thuộc nhóm reference type, biến của nó có thể nhận giá trị null. Giá trị null biểu diễn trạng thái đặc biệt: biến đó không có giá trị. Hay chính xác hơn, biến đó không trỏ vào một object cụ thể nào. Tuy nhiên các kiểu value lại không thể biểu diễn được trạng thái "không có giá trị".

Để giải quyết những tình huống tương tự, C# đưa vào khái niệm **nullable type**.

Nullable type trong C# là những kiểu dữ liệu value type đặc biệt có thể nhận giá trị null. Hiểu một cách đơn giản, nullable type cũng có thể xem là những value type (về mặt giá trị), đồng thời có thêm giá trị null. Giá trị null của nullable type biểu diễn trạng thái "không có giá trị" gán cho biến.

Ví dụ, nullable bool giờ có thể nhận ba giá trị: `true` | `false` | `null`. Trong đó, giá trị null của bool chỉ cần hiểu đơn thuần là "biến không có giá trị".

Sử dụng nullable type trong C#

C# sử dụng modifier **?** (dấu hỏi chấm) đặc sau value type tương ứng để biến nó thành nullable type. Ví dụ:

```
bool? b = null; // hoàn toàn hợp lệ, vì bool? là kiểu thuộc nhóm nullable và có khoảng giá trị truyền thống của bool
b = true;
int? i = 100;
i = null; // OK, vì int? là kiểu nullable, có khoảng giá trị tương tương int nhưng có thể nhận giá trị null
```

Rất lưu ý rằng, modifier **?** chỉ được phép đặt sau tên kiểu value type. Nếu đặt sau tên kiểu reference type sẽ báo lỗi lúc dịch.

Về bản chất, cách viết ? sau tên kiểu chỉ là một cú pháp tắt để tạo ra object của generic struct `System.Nullable<T>`. Do đó, nullable type có thêm hai property đặc trưng mà value type không có:

```
> int? i = 100;
> i.HasValue // true, vì i khác null
true
> i.Value // Value thuộc kiểu int, trong khi i thuộc kiểu int?
100
> i = null
null
> i.HasValue // false, vì giờ i có giá trị null
false
```

`HasValue` trả về true nếu biến có giá trị khác null. `Value` trả về giá trị value type của biến đó – hiểu đơn giản là chuyển giá trị từ nullable về value type thông thường.

Một số phép toán đặc biệt trên nullable và reference type

Tiếp theo đây chúng ta sẽ xem một số phép toán đặc biệt áp dụng cho các kiểu dữ liệu có khả năng nhận giá trị null, bao gồm các kiểu nullable và reference type.

Phép toán null coalescing

Đối với các kiểu dữ liệu có khả năng nhận giá trị null (reference và nullable type), C# cho phép áp dụng một phép toán đặc biệt: **null coalescing** (tạm dịch là toán tử hợp nhất null – nghe hơi dài dòng!).

Hãy xem một ví dụ đơn giản:

```
> bool? b = null;
> bool b2 = b ?? true;
> b2
true
>
```

Đầu tiên khai báo biến nullable bool? và gán giá trị null.

Lệnh `bool b2 = b ?? true;` giải thích như sau: nếu biến b có giá trị khác null thì b2 sẽ nhận giá trị b; nếu b có giá trị bằng null thì b2 sẽ nhận giá trị true. Phép toán ?? được gọi là null coalescing.

Cú pháp chung của null coalescing như sau:

```
<variable> ?? <value>;
```

Nếu *variable* có giá trị khác null, biểu thức sẽ nhận luôn giá trị của *variable*. Nếu *variable* có giá trị null, biểu thức sẽ nhận giá trị *value*. Thực chất, null coalescing có thể xem là dạng viết tắt của cấu trúc if để kiểm tra giá trị null. Tức là, ví dụ trên có thể viết lại như sau:

```
bool b2 = false;
if(!b.HasValue) b2 = true;
else b2 = b.Value;
// hoặc
if (b == null) b2 = true;
else b2 = b.Value;
```

Phép toán null conditional

Khi làm việc với các kiểu dữ liệu có thể nhận giá trị null như nullable hay reference type, nếu bạn gọi phương thức hoặc truy xuất thành viên trên biến có giá trị null sẽ gây ra lỗi trong quá trình thực thi (runtime exception). Ví dụ:

```
> int? i = null;
> i.Value
Nullable object must have a value.
  + System.ThrowHelper.ThrowInvalidOperationException(System.ExceptionResource)
  + Nullable<T>.get_Value()
> string str = null;
> str.ToLower()
Object reference not set to an instance of an object.
>
```

Trong ví dụ trên, i có giá trị null. Việc đọc giá trị của i qua Value property sẽ gây lỗi. Tương tự, str có giá trị null, gọi phương thức ToLower() trên str sẽ gây lỗi.

Do đó, bạn cần phải kiểm tra giá trị của biến trước khi thực hiện bất kỳ thao tác gì. Nếu biến có giá trị khác null mới thực hiện thao tác đó. Ví dụ:

```
if (str != null) str.ToLower();
```

Việc liên tục phải gọi lệnh kiểm tra điều kiện khác null như vậy khá nhàm chán. C# cung cấp một cú pháp gọn nhẹ đơn giản để làm việc này:

```
str?.ToLower();
```

Dấu ? sau tên biến và trước phép toán truy xuất phần tử có tên gọi là **null conditional operator**. Nó giúp bạn kiểm tra xem biến có bằng null không. Nếu biến khác null thì mới thực hiện phép toán truy xuất phần tử.

Theo đó, biểu thức `str?.ToLower()` sẽ không thực hiện nếu str có giá trị null, do đó sẽ không gây lỗi.

Kiểu dữ liệu dynamic

Kiểu dữ liệu đặc biệt cuối cùng xem xét trong bài này là **dynamic**. Kiểu dữ liệu này khác biệt với tất cả những kiểu dữ liệu khác của C#. Nó thậm chí còn khác biệt hẳn với cách suy nghĩ quen thuộc về kiểu dữ liệu trong C#.

Định kiểu tĩnh và định kiểu động

Nếu bạn đã từng làm việc với JavaScript hay PHP, bạn sẽ thấy một sự khác biệt rất lớn về việc khai báo và sử dụng biến. Khi khai báo biến bạn không cần chỉ định kiểu dữ liệu cụ thể của nó. Giá trị gán cho một biến không được xác định và kiểm tra ở giai đoạn dịch mà là ở giai đoạn thực thi. Đặc thù như vậy của ngôn ngữ gọi là *định kiểu động* (dynamically typed).

Ngược lại, C# bắt buộc kiểu của biến phải được xác định và kiểm tra ở giai đoạn dịch. Đặc thù đó của C# được gọi là *định kiểu tĩnh* (statically typed). Bạn hẳn có thể nhận ra rằng, ngôn ngữ định kiểu tĩnh như C# rất an toàn khi viết code. Nhưng đồng thời, nó lại thiếu đi sự linh hoạt của ngôn ngữ định kiểu động.

> Từ khóa **var** và kiểu **object** không hề giúp C# có đặc điểm của ngôn ngữ định kiểu động, mặc dù bạn không cần tự mình chỉ định kiểu cụ thể của biến khi khai báo với hai từ khóa này.

Bắt đầu từ C# 4.0, một kiểu dữ liệu đặc biệt được đưa vào để hỗ trợ những nhu cầu về định kiểu động trong lập trình: kiểu **dynamic**.

Khai báo và gán giá trị cho biến kiểu dynamic trong C#

Hãy cùng thực hiện một số ví dụ (sử dụng C# interactive):

```
> dynamic t = "Hello worl!";
> t
"Hello worl!"
> t.GetType()
[System.String]
> t = false
false
> t.GetType()
[System.Boolean]
> t = new List<int>()
List<int>(0) { }
> t.GetType()
[System.Collections.Generic.List`1[System.Int32]]
>
```

Bạn có thể thấy, biến `t` khai báo với từ khóa `dynamic` hoạt động có nét tương tự với `object`: bạn có thể gán cho nó bất kỳ giá trị nào. C# không hề phiền lòng khi bạn gán các loại giá trị chả liên quan gì với nhau cho biến `t`. Bạn vẫn có thể gọi phương thức từ biến t như với bất kỳ biến bình thường nào khác.

Tuy nhiên, khi gõ `t.` (gọi phép toán truy xuất thành viên) bạn hẳn sẽ nhận thấy sự khác biệt: Intellisense không hề hoạt động. Nếu bạn dùng từ khóa **`object`**, ít nhất Intellisense còn đưa ra danh sách các thành viên.

Vấn đề là: **`dynamic`** cho phép bạn khai báo biến theo đúng mô hình định kiểu động. C# không hề hiểu kiểu cụ thể của biến ở giai đoạn viết code và dịch. Do đó nó chẳng thể làm gì để giúp đỡ bạn. Giờ đây bạn phải tự mình xác định kiểu của biến là gì, và kiểu đó có những thành viên nào. Bạn phải tự mình viết tên các thành viên đó một cách chính xác. Nếu không, chương trình sẽ bị lỗi lúc chạy (mặc dù vẫn dịch thành công).

Hãy xem một ví dụ khác. Hãy viết đoạn code sau vào một Console App project:

```
1.  static void Main()
2.  {
3.     dynamic t = "Hello world!";
4.     Console.WriteLine(t.ToUpper());
5.     Console.WriteLine(t.toupper()); // compile thành công nhưng chạy sẽ bị lỗi
6.     Console.ReadKey();
7.  }
```

Hẳn bạn có thể thấy ngay, lớp `string` không hề có phương thức `toupper` (viết thường) mà chỉ có `ToUpper`. Tuy nhiên, C# vẫn cho phép bạn viết biểu thức `t.toupper`, khác hoàn toàn với phong cách kiểm soát nghiêm ngặt vốn có của C#. Chương trình compile hoàn toàn bình thường. Tuy nhiên, khi chạy bạn sẽ gặp ngay lỗi dưới đây:

```
Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
'string' does not contain a definition for 'toupper'
```

Đến đây hẳn bạn đã nhìn thấy sự khác biệt của **`dynamic`** với các cách khai báo biến "không chỉ định kiểu" như **`var`** và **`object`**.

Kiểu dynamic cho phép bạn hoàn toàn tự do trong việc gán giá trị và gọi thành viên. C# không hề quản việc này lúc biên dịch nữa. Bạn sẽ phải là người chịu trách nhiệm. "Tự do luôn đi kèm trách nhiệm!" – hãy ghi nhớ khi sử dụng dynamic.

> Bình thường bạn sẽ không mấy khi muốn sử dụng đến dynamic. Tuy nhiên, khi bạn phát triển ứng dụng web, dynamic sẽ thể hiện sức mạnh của nó.

Kết luận

Bài học này đã giúp bạn hiểu rõ một số kiểu dữ liệu đặc thù trong C#: anonymous, nullable, dynamic type.

Có thể bạn sẽ ít khi sử dụng trực tiếp các kiểu dữ liệu này trong phần lập trình cơ bản. Tuy nhiên, nếu bạn đi tiếp vào phần lập trình LINQ, anonymous type sẽ là phần không thể thiếu. Nếu bạn học lập trình với cơ sở dữ liệu, nullable type sẽ thể hiện vai trò của mình. Kiểu dynamic có vai trò quan trọng khi lập trình ứng dụng web.

+ Nếu bạn thấy site hữu ích, trước khi rời đi hãy **giúp đỡ** site bằng một hành động nhỏ để site có thể phát triển và phục vụ bạn tốt hơn.
+ Nếu bạn thấy bài viết hữu ích, hãy giúp **chia sẻ** tới mọi người.
+ Nếu có thắc mắc hoặc cần trao đổi thêm, mời bạn viết trong phần **thảo luận** cuối trang.
Cảm ơn bạn!