DELEGATE

Delegate là gì?

Để hiểu kiểu dữ liệu này, hãy cùng xem một số tình huống sau.

Giả sử chúng ta cần xây dựng một class, trong class này sẽ phải gọi một phương thức để thực hiện một hành động nào đó. Tuy nhiên chúng ta lại không biết được phương thức này khi xây dựng class! Phương thức này chỉ xuất hiện khi người khác sử dụng class để khởi tạo object.

Bạn đã từng đụng chạm vào windows forms hay wpf chưa? Nếu có chắc bạn sẽ để ý, nút bấm (Button) là một class đã xây dựng sẵn, còn phương thức xử lý sự kiện bấm nút (`OnClick`) lại do bạn tự viết. Làm sao để người xây dựng class Button biết và chạy phương thức xử lý sự kiện do bạn viết ra?

Vậy phải làm thế nào để hoàn thành yêu cầu **gọi một phương thức khi phương thức chưa tồn tại hoặc chưa xác định** ở giai đoạn xây dựng class?

Những tình huống khi không biết trước được phải gọi một phương thức cụ thể nào dẫn đến việc phải sử dụng một loại công cụ đặc biệt trong C#: delegate.

Delegate là những kiểu dữ liệu trong C# mà biến tạo ra từ nó chứa **tham chiếu tới phương thức**, thay vì chứa giá trị hoặc chứa tham chiếu tới object của các class bình thường.

Thuật ngữ "delegate" dịch sang tiếng Việt có thể là kiểu **đại diện** hoặc kiểu **ủy nhiệm**. Tuy nhiên, hai lối dịch này không được sử dụng phổ biến trong các tài liệu. Vì vậy, trong bài viết này chúng ta sẽ sử dụng thuật ngữ gốc tiếng Anh – **delegate**.

Một biến được tạo ra từ một kiểu delegate được gọi là một *biến delegate*.

Mỗi kiểu delegate khi được định nghĩa chỉ cho phép biến của nó chứa tham chiếu tới những phương thức phù hợp với quy định của delegate này.

Vai trò của delegate

Delegate cho phép một class uyển chuyển và linh động hơn trong việc sử dụng phương thức. Theo đó, nội dung cụ thể của một phương thức không được định nghĩa sẵn trong class mà sẽ do người dùng class đó tự định nghĩa trong quá trình khởi tạo object. Điều này giúp phân chia logic của một class ra các phần khác nhau và do những người khác nhau xây dựng.

Delegate được sử dụng để giúp một class object tương tác ngược trở lại với thực thể tạo ra và sử dụng class đó. Điều này giúp class không bị "cô lập" bên trong thực thể đó. Ví dụ, delegate giúp gọi các phương thức của thực tế tạo ra và chứa class object.

Với khả năng tạo ra tương tác ngược như vậy, delegate trở thành hạt nhân của mô hình lập trình hướng sự kiện, được sử dụng trong công nghệ winforms và WPF.

Ví dụ, khi xây dựng các điều khiển của windows form (nút bấm, nút chọn, menu, v.v.), người lập trình ra các lớp này không thể xác định được người dùng muốn làm gì khi nút được bấm, khi menu được chọn. Do đó, bắt buộc phải sử dụng cơ chế của delegate để chuyển logic này sang cho người sử dụng các lớp đó viết code. Như vậy, Button khi được tạo ra trong một Form có khả năng tương tác với code của Form, chứ không cô lập chính mình.

Delegate cũng được sử dụng phổ biến với mô hình lập trình bất đồng bộ ở dạng các phương thức callback, hoặc trong lập trình đa luồng.

Ví dụ minh họa

Hãy thực hiện và phân tích ví dụ sau để thấy rõ hơn cách khai báo và sử dụng delegate trong C#.

Trong ví dụ này chúng ta "giả lập" một chương trình giúp tính toán và vẽ đồ thị hàm số (vẽ giả thôi, không phải vẽ thật đâu). Trong đó chúng ta sẽ viết một class giúp tính toán và in giá trị của hàm số trong một dải giá trị. Hàm số thật sự sẽ do người dùng class tự tạo và cung cấp sau.

```
using System;
namespace ConsoleApp
{
    internal delegate double MathFunction(double x);

    internal class Graph
    {
        public MathFunction Function { get; set; }

        public void Render(MathFunction function, double[] range)
        {
            Function = function;
            Console.WriteLine($"Drawing the function graph: {function.Method}");
            foreach (var x in range)
            {
                var y = function(x);
                Console.Write($"{y:f3}  ");
            }
            Console.WriteLine("rn-----------------");
        }
    }

    internal class Mathematics
    {
        public double Cos(double x) => Math.Cos(x);
        public static double Tan(double x) => Math.Tan(x);
    }

    internal class Program
    {
        private static double Sin(double x)
        {
            return Math.Sin(x);
        }
```

```
        private static void Main(string[] args)
        {
            Graph graph = new Graph();

            double[] range = new double[] { 1.0, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0 };

            graph.Render(Sin, range);
            graph.Render(Mathematics.Tan, range);
            Mathematics math = new Mathematics();
            graph.Render(math.Cos, range);
            graph.Render(Math.Sqrt, range);
            MathFunction function = delegate (double x) { return x *= 2; };
            graph.Render(function, range);
            graph.Render(delegate (double x) { return x++; }, range);
            graph.Render((double x) => { return x *= 10; }, range);
            graph.Render(x => x / 10, range);
            Console.ReadKey();
        }
    }
}
```

Kỹ thuật lập trình với delegate

Trong phần này chúng ta sẽ phân tích chi tiết kỹ thuật sử dụng delegate đã gặp trong ví dụ minh họa trên.

Khai báo kiểu delegate

Trong ví dụ trên, chúng ta khai báo một kiểu delegate tên là MathFunction trực tiếp trong không gian tên.

```
internal delegate void MathFunction();
```

Vì kiểu delegate có cùng cấp độ với class, nó có thể được khai báo trực tiếp trong không gian tên, cũng như có thể khai báo làm một kiểu nội bộ bên trong class (giống như khai báo class bên trong class).

Kiểu `MathFunction` này có hình thức tương tự với một phương thức nhận một biến kiểu `double` và trả lại một giá trị `double`.

Qua đây chúng ta thấy, về mặt hình thức, khai báo một kiểu delegate giống như khai báo một phương thức không có thân, chỉ cần thêm từ khóa `delegate` trước kiểu trả về và kết thúc khai báo bằng dấu chấm phẩy.

Mô tả phương thức và delegate

Để dễ dàng hơn trong việc mô tả phương thức (và kiểu delegate), người ta đưa ra một quy ước mô tả như sau: (danh sách tham số theo kiểu) -> kiểu đầu ra. Ví dụ, phương thức

```
private float Div(int a, int b) {}
```

có mô tả là `(int, int) -> float`.

Khi đọc mô tả này chúng ta sẽ hiểu: phương thức (không quan tâm đến tên gọi) có hai tham số đầu vào cùng kiểu int và trả về kết quả thuộc kiểu float.

Với quy ước viết này, `MathFunction` ở trên đại diện cho tất cả các phương thước có mô tả là `(double) -> double`.

Tất cả các phương thức có mô tả `(double) -> double` đều có thể gán cho biến thuộc kiểu `MathFunction`. Mô tả này cũng được gọi là mô tả của `MathFunction`.

Khai báo và sử dụng biến delegate

Sau khi khai báo kiểu, có thể khai báo biến thuộc kiểu dữ liệu này tương tự như khai báo các biến bình thường:

```
/* khai báo property thuộc kiểu MathFunction.
 * MathFunction được sử dụng như những kiểu dữ liệu thông thường
 */
public MathFunction Function { get; set; }
```

Tham số thuộc kiểu delegate có thể được khai báo và gán như tham số bình thường:

```
/* phương thức này có 1 tham số đầu vào là kiểu delegate MathFunction.
 * Kiểu delegate làm tham số không khác gì kiểu dữ liệu bình thường
 */
public void Render(MathFunction function, double[] range)
{
    Function = function;
```

Biến thuộc kiểu delegate cũng là một object, tương tự như các object tạo ra từ class, nó cũng có thuộc tính và phương thức như các object bình thường. Thực tế, tất cả kiểu delegate do người dùng định nghĩa đều kế thừa từ lớp `System.Delegate` nên cũng được kế thừa các thuộc tính và phương thức của class này.

Bởi vì biến delegate sẽ chứa tham chiếu tới một phương thức, ta có thể sử dụng tên biến delegate như một phương thức thực thụ, nghĩa là có thể "gọi" biến này như gọi phương thức bình thường. Khi "gọi" một biến delegate, phương thức nó trỏ tới sẽ được thực thi.

```
    Console.WriteLine($"Drawing the function graph: {function.Method}");
    foreach (var x in range)
    {
        var y = function(x);
        Console.Write($"{y:f3}  ");
    }
```

Ngoài cách "gọi" biến delegate như gọi phương thức, các kiểu delegate còn có thêm phương thức `Invoke` giúp gọi phương thức biến này trỏ tới:

```
var y = function.Invoke(x);
```

Cẩn thận hơn nữa chúng ta có thể kiểm tra object trước khi gọi Invoke:

```
var y = function?.Invoke(x);
```

Phép toán ? cho phép kiểm tra xem một object function có giá trị null hay không. Nếu object nhận giá trị khác null mới thực hiện phương thức Invoke. Cách sử dụng này là an toàn nhất. Nếu function nhận giá trị null thì phương thức Invoke sẽ không được gọi. Nếu không kiểm tra null, lời gọi phương thức trên một object null sẽ làm phát sinh lỗi.

Truyền tham số kiểu delegate cho phương thức

Ở giai đoạn khởi tạo object của class, người dùng class mới truyền phương thức cụ thể cho tham số thuộc kiểu delegate.

```
graph.Render(Sin, range);
graph.Render(Mathematics.Tan, range);
Mathematics math = new Mathematics();
graph.Render(math.Cos, range);
graph.Render(Math.Sqrt, range);
```

Nếu một tham số của phương thức là biến delegate, chúng ta có thể trực tiếp truyền tên của một phương thức có chung mô tả với kiểu delegate.

Lưu ý: truyền một phương thức làm tham số khác với truyền lời gọi phương thức làm tham số. Truyền lời gọi phương thức có thể xem như tương đương với truyền dữ liệu bình thường (string, bool, int, v.v.), không liên quan đến delegate.

Generic delegate

Như đã biết khi xem xét về delegate, để làm việc với delegate chúng ta cần trước hết khai báo delegate như khai báo kiểu dữ liệu bình thường, sau đó sử dụng kiểu delegate đó để khai báo biến. Đến giai đoạn sử dụng chúng ta gán biến delegate đó với một phương thức phù hợp với yêu cầu của kiểu delegate.

C# hỗ trợ người lập trình bằng cách định nghĩa ra một loạt kiểu dữ liệu *generic delegate* mà chúng ta có thể trực tiếp sử dụng ngay để khai báo biến. Sử dụng generic delegate giúp bỏ qua giai đoạn khai báo kiểu delegate.

Đọc bài viết này để nắm rõ hơn về generic trong C#.

Về cơ bản, generic delegate là các kiểu delegate đã được định nghĩa sẵn sử dụng cơ chế generic. .NET framework định nghĩa 3 nhóm generic delegate: `Actions`, `Funcs`, `Predicates`.

Actions

Actions là các kiểu delegate tương ứng với các phương thức không trả về dữ liệu (đầu ra là `void`). Các kiểu `Action` được định nghĩa trong không gian tên `System` như sau:

```
namespace System
{
    public delegate void Action();
```

```
    public delegate void Action<in T>(T obj);
    public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);

}
```

.NET framework định nghĩa tổng cộng 16 generic delegate, cái đầu tiên có 1 tham số đầu vào, cái cuối cùng có 16 tham số đầu vào.

Như vậy, kiểu `Action<T1,..>` có thể tương ứng với bất kỳ phương thức nào không trả về giá trị và có từ 1 đến 16 tham số đầu vào (bất kỳ kiểu gì). Riêng `delegate void Action()` tương ứng với các phương thức không nhận tham số và không trả về giá trị.

Như vậy, khi sử dụng `Action` hoặc `Action<T1,..>` sẽ không cần khai báo các kiểu delegate có kiểu ra là void nữa (và có ít hơn 16 tham số đầu vào).

Dưới đây là một số ví dụ về cách sử dụng các kiểu actions:

```
Action action1 = () => Console.WriteLine("Hello world");
Action<string> action2 = (s) => Console.WriteLine(s);
Action<string, int> action3 = (s, i) => { for (int j = 0; j < i; j++) Console.WriteLine(s); };
```

Funcs

Funcs là các kiểu delegate tương ứng với các phương thức có trả về dữ liệu. Các kiểu funcs được định nghĩa trong không gian tên `System` như sau:

```
namespace System
{
    public delegate TResult Func<out TResult>();
    public delegate TResult Func<in T, out TResult>(T arg);
    public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
}
```

Tương tự như với actions, .NET framework cũng định nghĩa 17 kiểu delegate như trên với số lượng tham số đầu vào từ 0 đến 16.

Định nghĩa kiểu funcs khác với actions ở chỗ, funcs luôn phải có kiểu đầu ra đặt ở vị trí cuối cùng trong danh sách kiểu giả generic (out TResult). Sau đây là một số ví dụ sử dụng funcs:

```
Func<int> func1 = () => 0;
Func<int, int> func2 = (i) => i * 10;
Func<int, int, float> func3 = (a, b) => a / b;
Console.WriteLine($"func1: {func1}; func2: {func2(10)}; func3: {func3(1, 2)}");
```

Biến func1 được khai báo và tham chiếu tới hàm lambda không nhận tham số và luôn trả về giá trị 0 (kiểu int); biến func2 tham chiếu tới hàm lambda nhận tham số vào kiểu int, nhân tham số đó với 10 và trả lại kết quả này (kiểu int); biến func3 tham chiếu tới hàm lambda nhận hai số nguyên và trả về kết quả phép chia (kiểu float).

Predicate

Predicate là kiểu delegate được định nghĩa sẵn như sau (trong `System`):

```
namespace System
{
    public delegate bool Predicate<[NullableAttribute(2)] in T>(T obj);
}
```

Như vậy, predicate là kiểu delegate tương ứng với các phương thức có 1 tham số đầu vào và trả về giá trị bool. Predicate được sử dụng trong các biểu thức so sánh. Dưới đây là một số ví dụ:

```
Predicate<int> predicate1 = (i) => i > 100;
Console.WriteLine($"is 10 > 100? it's {predicate1(10)}");
Predicate<string> predicate2 = (s) => s.Length > 10;
```

Biến predicate1 tham chiếu tới một hàm lambda thực hiện so sánh xem một số nguyên có lớn hơn 100 hay không; biến predicate2 tham chiếu tới một hàm lambda nhận một chuỗi ký tự và so sánh xem độ dài chuỗi đó có lớn hơn 10 không.

Như chúng ta thấy trong các ví dụ trên đều sử dụng hàm lambda với delegate. Khi sử dụng theo kiểu này chúng ta có thể hoàn toàn bỏ qua việc khai báo kiểu của tham số trong hàm lambda vì C# có thể tự suy đoán ra kiểu. Ngoài ra, nếu thân phương thức chỉ có 1 lệnh duy nhất thì sử dụng lối viết "expression body" cho gọn.

Sử dụng delegate với phương thức vô danh, hàm lambda, hàm cục bộ

Để truyền tham số thuộc kiểu delegate cho một phương thức, chúng ta phải viết các phương thức có mô tả mà kiểu delegate yêu cầu.

Do đặc thù của C#, phương thức bắt buộc phải được định nghĩa trong một class (dù là instance method hay static method). Đối với instance method, chúng ta còn phải khởi tạo object trước khi truyền phương thức này làm tham số cho phương thức khác qua delegate.

Nếu phương thức này chỉ được sử dụng một lần duy nhất lúc truyền làm tham số, hoặc phương thức quá đơn giản (chỉ bao gồm 1 dòng lệnh), việc xây dựng hàng loạt phương thức nhỏ như vậy có thể làm code bị phân mảnh khiến khó theo dõi và quản lý.

C# 2 bắt đầu đưa vào khái niệm *phương thức vô danh* (anonymous method), C# 3 đưa vào khái niệm *hàm lambda* (lambda statement) giúp xây dựng các phương thức sử dụng một lần như vậy. Hiện nay hàm lambda được sử dụng phổ biến hơn.

Phương thức vô danh

Phương thức vô danh (anonymous method) cũng là một loại phương thức, tuy nhiên khác biệt với phương thức bình thường ở một số điểm:

1. Không có tên: vì phương thức vô danh chủ yếu được sử dụng làm tham số cho phương thức khác, điều quan trọng nhất là hoạt động của phương thức (thân phương thức), còn tên không quan trọng; phương thức loại này cũng không được gọi lại (tái sử dụng) ở nhiều nơi trong code như phương thức bình thường, do đó cũng không cần tên gọi.
2. Có thể khai báo trực tiếp ở chỗ cần dùng: ví dụ có thể khai báo thẳng trong danh sách tham số của hàm khác, và do đó, có thể truy xuất cả các biến cục bộ của phương thức nơi nó được khai báo.

Trong các tài liệu có thể sử dụng hai cách gọi: **phương thức vô danh** hoặc **phương thức nặc danh**. Hai cách gọi này là tương đương. Trong bài giảng này chúng ta thống nhất gọi là **phương thức vô danh**.

Hãy cùng xem ví dụ sau đây về khai báo phương thức vô danh:

Chúng ta vẫn tiếp tục sử dụng ví dụ minh họa đã làm ở bài trước. Ở đây chúng ta không khai báo thêm phương thức bình thường như trước mà trực tiếp khai báo một phương thức vô danh và gán cho biến function thuộc kiểu `MathFunction`:

```
MathFunction function = delegate (double x) { return x *= 2; };
graph.Render(function, range);
graph.Render(delegate (double x) { return x++; }, range);
```

Chúng ta cũng có thể khai báo phương thức vô danh trực tiếp ở vị trí tham số:

```
graph.Render(delegate (double x) { return x++; }, range);
```

Qua ví dụ này có thể thấy, khai báo phương thức vô danh chỉ khác biệt với phương thức thông thường ở chỗ, vị trí tên phương thức được thay thế bằng từ khóa delegate.

Hàm lambda

Hàm lambda cũng có những đặc trưng của phương thức vô danh nhưng có thể bỏ qua luôn khai báo kiểu của các tham số.

Điều này xuất phát từ thực tế là hàm lambda khi được khai báo làm tham số của một phương thức thì các tham số của nó bắt buộc phải tuân thủ theo mô tả của delegate. Do đó C# compiler có thể tự suy ra kiểu của các tham số mà không cần viết rõ ra.

Tương tự, một hàm lambda được khai báo hoàn toàn giống như một phương thức bình thường, khác biệt lớn nhất là nó có thể được khai báo thẳng trong thân phương thức khác (giống phương thức vô danh), và danh sách tham số được nối với thân phương thức bằng dấu =>. Phương thức lambda cũng rất thường được sử dụng với expression body.

```
graph.Render((double x) => { return x *= 10; }, range);
graph.Render(x => x / 10, range);
```

Ở tình huống thứ hai chúng ta thấy danh sách tham số của hàm lambda thậm chí không có cả tên kiểu dữ liệu.

Lý do là vì hàm này dùng làm tham số thứ nhất cho phương thức Render. Tham số này đã được quy định mô tả là `(double) -> double`. Do đó, tham số `x` sẽ được C# tự hiểu là thuộc kiểu double. Khả năng tự suy đoán kiểu này kết hợp với expression body giúp hàm lambda trở nên rất ngắn gọn.

Ở những vị trí chúng ta truyền phương thức vô danh và hàm lambda, C# compiler tự sinh ra tên gọi cho các hàm này. Tuy nhiên, vì lúc code không có tên, chúng ta không có khả năng gọi lại các hàm này ở những vị trí khác trong code.

Hàm cục bộ

Từ C# 7, chúng ta có thêm một khả năng nữa để xây dựng một phương thức trong thân một phương thức khác bên cạnh sử dụng phương thức vô danh và hàm lambda: sử dụng *hàm cục bộ* (local function).

Hàm cục bộ là một phương thức được định nghĩa bên trong thân một phương thức khác.

Hàm cục bộ hoàn toàn không khác biệt với các phương thức thông thường. Tuy nhiên hàm cục bộ không có từ khóa điều khiển truy cập và chỉ có thể được gọi bên trong thân phương thức chứa nó.

Nếu bạn chưa biết: trong C# có hai điều khác biệt với nhiều ngôn ngữ khác thuộc họ C/C++: (1) có thể khai báo class bên trong class, và (2) có thể khai báo phương thức bên trong phương thức.