Buổi 6.2
Delegate và biểu thức Lambda

I. Đặt vấn đề

Giả sử chúng ta xây dựng 1 class phục vụ việc tính toán các phép tính cộng và trừ giữa 2 số nguyên như sau:

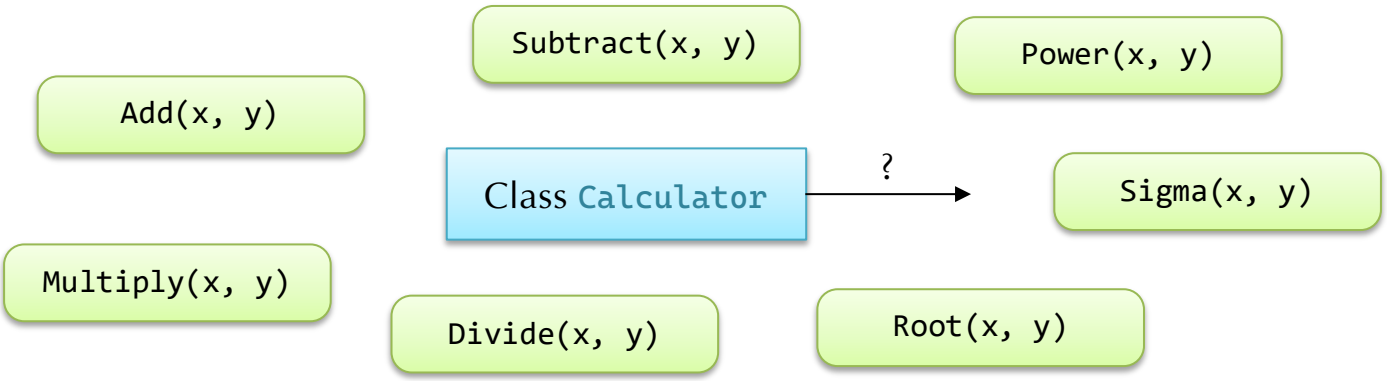
```
class Calculator
{
      public int Add(int x, int y) => x + y;

      public int Subtract(int x, int y) => x - y;
}
```

Nếu muốn mở rộng thêm các phép tính khác, chúng ta sẽ bổ sung thêm phương thức, ví dụ phương thức `Multiply()` cho phép nhân, và `Divide()` cho phép chia…

Vấn đề được đặt ra là, có rất nhiều phép tính giữa 2 số nguyên x và y. Bên cạnh 4 phép tính cơ bản là cộng, trừ, nhân, chia, còn các phép tính khác như lũy thừa x^y, khai căn $\sqrt[y]{x}$… Thậm chí trong quá trình sử dụng class, người dùng có thể tự định nghĩa thêm các phép tính tùy ý như tổng các số nguyên từ x đến y (tức là $\sum_{n=x}^{y} n$). Làm sao có thể cài đặt class `Calculator` một cách linh hoạt khi không biết trước yêu cầu người dùng?

Giải pháp là thay vì cài đặt sẵn các phương thức tính toán trong class, chúng ta sẽ để người dùng tự cài đặt các phép tính này theo ý của họ ở bên ngoài class. Người dùng muốn tính toán như thế nào thì đều có thể cài đặt theo ý muốn.

Tuy nhiên, vì phương thức phải được gọi thông qua tên phương thức, nên nếu để cho người dùng tự cài đặt, thì bản thân đối tượng thuộc class này sẽ không biết gọi các phương thức tính toán như thế nào, vì những phương thức này nằm ngoài class và do người dùng tự đặt tên:



Cách giải quyết vấn đề này là cần phải có 1 cơ chế cho phép gọi nhiều phương thức khác nhau thông qua 1 phương thức duy nhất có tên xác định.

II. Delegate

1. Khái niệm Method Signature

Method signature cho biết kiểu dữ liệu và thứ tự của tham số cũng như kiểu trả về của 1 phương thức, được viết ngắn gọn theo cấu trúc `(danh_sách_tham_số) => kiểu trả về`.

Ví dụ: Signature của một số phương thức:

Khai báo phương thức	Signature
`int Add(int x, int y)`	`(int, int) => int`
`int Subtract(int x, int y)`	
`int Multiply(int x, int y)`	
`void ShowMessage(string msg)`	`(string) => void`
`bool CheckEven(int n)`	`(int) => bool`
`string GetName()`	`() => string`

2. Khái niệm Delegate

Trong C#, **delegate**[1] là 1 kiểu tham chiếu trỏ đến các phương thức có cùng signature. Nhờ vậy, 1 delegate có thể đại diện cho bất kỳ phương thức `F()` nào có signature giống với delegate đó, và do đó, có thể gọi thực hiện phương thức `F()`.

Để dùng được delegate, cần thực hiện 3 bước sau:

- Khai báo (declare) kiểu delegate và khởi tạo (instantiate) biến delegate.
- Gán phương thức vào delegate.
- Thực thi (invoke) delegate.

Kiểu delegate được khai báo bằng từ khóa `delegate` kèm theo method signature.

Ví dụ: Cú pháp khai báo 1 kiểu delegate có signature là `(int, int) => int` như sau:

```
delegate int PerformCalculation(int x, int y);
```

Sau khi khai báo delegate như trên, `PerformCalculation` được tính là 1 kiểu và chúng ta có thể dùng nó để khởi tạo biến delegate tương ứng:

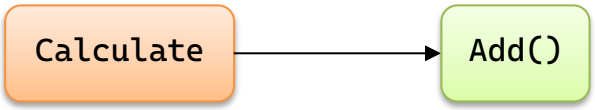
```
PerformCalculation Calculate;
```

Lưu ý: `Calculate` là 1 biến delegate, **không phải** là object.

[1] Tạm dịch là phương thức ủy quyền

Biến **Calculate** có thể trỏ đến các phương thức có cùng signature với nó. Nói 1 cách dễ hiểu hơn, chúng ta có thể gán những phương thức nhận tham số là 2 số nguyên và trả về kiểu số nguyên cho biến **Calculate**:

```
int Add(int x, int y) => x + y;

Calculate = Add;
```



Chúng ta có thể gọi phương thức **Add()** thông qua biến **Calculate** theo 1 trong 2 cách:

```
// Cách 1
Calculate(3, 5);          // Trả về: 8

// Cách 2
Calculate.Invoke(3, 5);   // Trả về: 8
```

Điểm khác biệt giữa 2 cách:

- Với cách 1, nếu biến **Calculate** là **null**, trình biên dịch sẽ báo lỗi.
- Với cách 2, chúng ta có thể kiểm tra biến delegate có đang **null** hay không bằng toán tử **?** như sau: **Calculate?.Invoke(3, 5);**. Nếu biến delegate đang là **null**, trình biên dịch sẽ bỏ qua, không thực thi và không báo lỗi.

Tổng kết lại, vấn đề đặt ra ở phần I sẽ được giải quyết như sau:

- Class **Calculator** sẽ khai báo 1 kiểu delegate có signature là **(int, int) => int**:

```
class Calculator
{
      public delegate int PerformCalculation(int x, int y);
      public PerformCalculation Calculate { get; set; }
}
```

- Tại phương thức **Main()**, người dùng có thể tùy ý định nghĩa các phương thức tính toán, sau đó gán phương thức đó cho biến **Calculate** miễn là phương thức đó có cùng signature với biến **Calculate**.

Ví dụ: Thực hiện phép cộng:

```
static void Main(string[] args)
{
      Calculator c = new Calculator();
      c.Calculate = Add;
      Console.WriteLine(c.Calculate(3, 5));    // Trả về: 8
}
static int Add(int x, int y) => x + y;
```

Ví dụ: Thực hiện phép lũy thừa:

```
static void Main(string[] args)
{
      Calculator c = new Calculator();
```

```
        c.Calculate = Power;
        Console.WriteLine(c.Calculate(3, 5));    // Trả về: 125
}
static int Power(int x, int y) => Convert.ToInt32(Math.Pow(x, y));
```

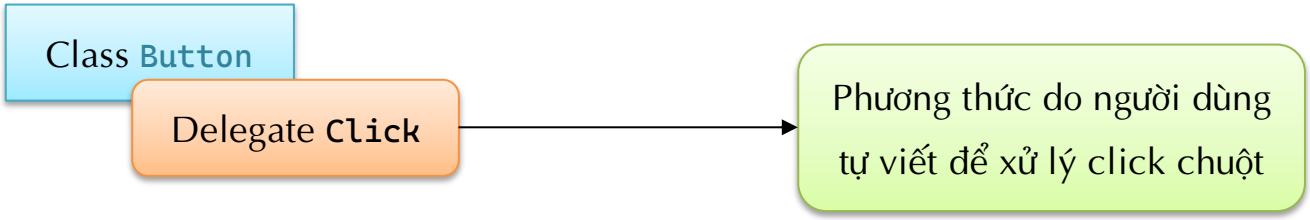
Ví dụ: Thực hiện phép tính tổng từ x đến y:

```
static void Main(string[] args)
{
        Calculator c = new Calculator();
        c.Calculate = Sigma;
        Console.WriteLine(c.Calculate(3, 5));    // Trả về: 12
}
static int Sigma(int x, int y)
{
        int result = 0;
        for (int i = x; i <= y; i++)
        {
                result += i;
        }
        return result;
}
```

3. Mục đích sử dụng của Delegate

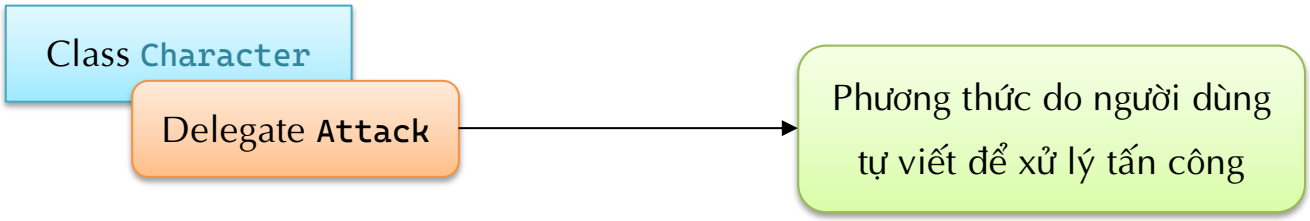
Ví dụ về class `Calculator` ở trên chỉ để minh họa cho khái niệm delegate. Trong thực tế, đôi khi chúng ta cần phải tạo ra 1 class mà không biết cài đặt phương thức xử lý như thế nào (thường thấy đối với các sự kiện – event[2] trong phần mềm). Việc cài đặt các phương thức này sẽ dành cho người dùng:

- Tạo ra 1 class `Button` mô phỏng 1 nút bấm trên giao diện của phần mềm. Nút bấm sẽ có phương thức `Click()` cho phép thực hiện 1 chức năng khi người dùng nhấn chuột vào nút đó. Tuy nhiên, tại thời điểm xây dựng class `Button`, lập trình viên không thể nào biết được người dùng sẽ muốn thực hiện chức năng gì khi click chuột vào nút này. Do đó, thay vì cài đặt phương thức `Click()`, lập trình viên sẽ dùng delegate cho phép người dùng tự viết phương thức xử lý và gán nó cho `Click`:



[2] Một số sự kiện trong phần mềm như click chuột, nhấn phím, di chuyển chuột,...

– Tạo ra 1 class `Character` mô phỏng 1 nhân vật trong game. Nhân vật sẽ có phương thức `Attack()` cho phép thực hiện hành động tấn công. Tuy nhiên, tại thời điểm xây dựng class `Character`, lập trình viên không thể nào biết được người dùng sẽ muốn cài đặt hành động tấn công như thế nào. Do đó, thay vì cài đặt phương thức `Attack()`, lập trình viên sẽ dùng delegate cho phép người dùng tự viết phương thức xử lý và gán nó cho `Attack`:

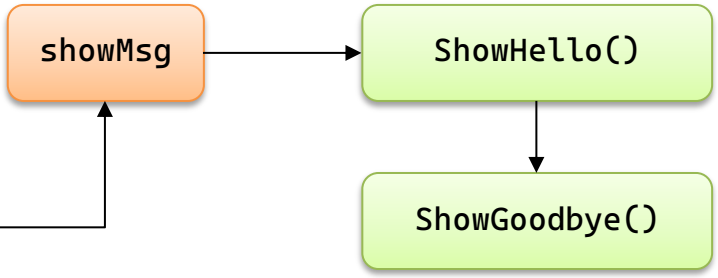


Với cách sử dụng delegate, class được viết ra sẽ rất linh hoạt. Người sử dụng class có thể tùy biến chức năng của class đó tùy ý theo từng trường hợp cụ thể.

III. Multicast

Một delegate có thể cùng lúc trỏ đến nhiều phương thức khác nhau (gọi là **multicast**) bằng cách dùng toán tử `+=` khi gán phương thức. Nếu thực thi delegate, các phương thức sẽ được gọi theo thứ tự. Ví dụ sau đây cho thấy delegate showMsg trỏ đến lần lượt 2 phương thức `ShowHello()` và `ShowGoodbye()`:

```
delegate void ShowMsg();

static void Main(string[] args)
{
      ShowMsg showMsg;
      showMsg = ShowHello;
      showMsg += ShowGoodbye;

      showMsg();
}

static void ShowHello() { Console.WriteLine("Hello"); }

static void ShowGoodbye() { Console.WriteLine("Goodbye"); }
```



Kết quả xuất ra màn hình:

```
Hello
Goodbye
```

Ngược lại, cũng có thể gỡ bỏ 1 phương thức khỏi delegate bằng toán tử `-=` như sau:

```
showMsg -= ShowHello;
```

IV. Generic Delegate

.NET Framework hỗ trợ sẵn một số kiểu delegate để lập trình viên có thể dùng để khởi tạo delegate mà không cần khai báo thủ công 1 kiểu mới. Những kiểu delegate hỗ trợ sẵn này được gọi là **generic delegate**, và nó chấp nhận nhiều kiểu method signature khác nhau với kiểu dữ liệu tùy ý.

Có 3 loại generic delegate: `Func`, `Action` và `Predicate`.

1. Func

`Func` là 1 kiểu delegate hỗ trợ signature có 0 → 16 tham số và trả về 1 kiểu dữ liệu. Cách khai báo 1 delegate với `Func`:

```
Func<kiểu_tham_số, kiểu_trả_về>
```

Trong đó:

- **`kiểu_tham_số`** bao gồm kiểu dữ liệu của 0 → 16 tham số.
- **`kiểu_trả_về`** luôn nằm cuối cùng.

Ví dụ:

- Một số cách khai báo `Func`:

Khai báo	Signature tương ứng
`Func<int>`	`() => int`
`Func<int, int, int>`	`(int, int) => int`
`Func<double, bool>`	`(double) => bool`
`Func<string, double>`	`(string) => double`
`Func<double, double, int, bool>`	`(double, double, int) => bool`

- Dùng `Func` để khởi tạo delegate trỏ tới phương thức tính tổng:

```
static void Main(string[] args)
{
    Func<int, int, int> func = Add;
    func(3, 5);          // Trả về: 8
}
static int Add(int x, int y) => x + y;
```

2. Action

`Action` là 1 kiểu delegate hỗ trợ signature có 0 → 16 tham số và không trả về dữ liệu (tức là kiểu trả về là `void`). Cách khai báo 1 delegate với `Action`:

```
Action<kiểu_tham_số>
```

Trong đó: **`kiểu_tham_số`** bao gồm kiểu dữ liệu của 0 → 16 tham số.

Ví dụ:

- Một số cách khai báo `Action`:

Khai báo	Signature tương ứng
`Action`	`() => void`
`Action<string>`	`(string) => void`
`Action<double, bool>`	`(double, bool) => void`
`Action<string, double, bool>`	`(string, double, bool) => void`

- Dùng `Action` để khởi tạo delegate trỏ tới phương thức hiển thị lời chào và tên:

```
static void Main(string[] args)
{
    Action action1 = ShowHello;
    action1();          // Xuất ra chuỗi: Hello

    Action<string> action2 = ShowName;
    action2("Ricky");   // Xuất ra chuỗi: Ricky
}
static void ShowHello() => Console.WriteLine("Hello");
static void ShowName(string s) => Console.WriteLine(s);
```

3. Predicate

`Predicate` là 1 kiểu delegate hỗ trợ signature nhận vào 1 tham số và trả về kiểu `bool`.

Cách khai báo 1 delegate với `Predicate`:

```
Predicate<kiểu_tham_số>
```

Trong đó: **`kiểu_tham_số`** là kiểu dữ liệu của tham số.

Ví dụ:

- Một số cách khai báo `Predicate`:

Khai báo	Signature tương ứng
`Predicate<int>`	`(int) => bool`
`Predicate<string>`	`(string) => bool`

- Dùng `Predicate` để khởi tạo delegate trỏ tới phương thức kiểm tra số dương:

```
static void Main(string[] args)
{
    Predicate<int> predicate = IsPositive;
    predicate(5);       // Trả về: true
}
static bool IsPositive(int value) => value > 0;
```

Lưu ý: `Predicate<X>` tương đương với `Func<X, bool>` (X là kiểu dữ liệu của tham số).

V. Anonymous Method

Trong các ví dụ về delegate ở trên, chúng ta đều phải cài đặt 1 phương thức đầy đủ, bao gồm cả tên phương thức để có thể gán được cho delegate. Việc này là khá rườm rà và tốn công sức, vì đôi khi những phương thức này chỉ được dùng cho 1 mục đích duy nhất là gán cho delegate.

C# cho phép cài đặt những phương thức vô danh (**anonymous method**), là những phương thức không có tên, được khai báo bằng từ khóa `delegate`, và có thể được gán trực tiếp cho 1 biến delegate.

Ví dụ: Anonymous method cho phương thức:

Anonymous method

- Tính tổng:

```
Func<int, int, int> func = delegate (int x, int y)
                           { return x + y; };
```

- Hiển thị lời chào:

```
Action action = delegate { Console.WriteLine("Hello"); };
```

- Hiển thị tên:

```
Action<string> action = delegate (string s)
                        { Console.WriteLine(s); };
```

- Kiểm tra số dương:

```
Predicate<int> predicate = delegate (int x) { return x > 0; };
```

Lưu ý: Anonymous method **không** viết được dưới dạng expression body.

VI. Biểu thức Lambda

C# cho phép rút gọn anonymous method thành 1 cú pháp ngắn gọn hơn gọi là biểu thức Lambda (**Lambda expression**) thông qua các bước sau:

```
delegate (int x) { return x > 0; }
```

Loại bỏ từ khóa `delegate` và thêm toán tử Lambda

```
(int x) => { return x > 0; };
```

Delegate đã được xác định kiểu dữ liệu tham số nên có thể loại bỏ

Nếu chỉ có 1 tham số, có thể loại bỏ `( )`

```
x => { return x > 0; };
```

Nếu chỉ có 1 câu lệnh, có thể loại bỏ `{ }`

Nếu câu lệnh đó là `return`, loại bỏ `return` và `;`

```
x => x > 0
```

Với cách này, `delegate (int x) { return x > 0; }` có thể rút gọn thành 1 biểu thức Lambda `x => x > 0`.

Ví dụ: Biểu thức Lambda của một số delegate trong các ví dụ ở trên:

Delegate	Biểu thức Lambda
`delegate (int x, int y) { return x + y; }`	`(x, y) => x + y`
`delegate { Console.WriteLine("Hello"); }`	`() => Console.WriteLine("Hello")`
`delegate (string s) { Console.WriteLine(s); }`	`s => Console.WriteLine(s)`
`delegate (int x) { return x > 0; }`	`x => x > 0`

Tương tự như anonymous method, biểu thức Lambda cũng có thể được dùng để gán cho 1 delegate:

```
Func<int, int, int> func = (x, y) => x + y;

Action action1 = () => Console.WriteLine("Hello");

Action<string> action2 = s => Console.WriteLine(s);

Predicate<int> predicate = x => x > 0;
```

VII. Dùng Delegate làm tham số cho phương thức

Do delegate là 1 kiểu, do đó nó hoàn toàn có thể được truyền vào làm tham số cho phương thức.

Giả sử cần cài đặt phương thức `Check()` để kiểm tra số nguyên N thỏa mãn 1 điều kiện nào đó hay không, trong đó điều kiện là do người dùng nhập. Nghĩa là tại thời điểm cài đặt phương thức `Check()`, chúng ta chưa thể biết điều kiện cần kiểm tra là gì. Tuy nhiên, chúng ta biết được điều kiện có thể được viết dưới dạng method có signature là `(int) => bool`.

Do đó, chúng ta có thể tạo ra 1 kiểu delegate có signature là `(int) => bool` và dùng delegate này để làm tham số cho phương thức `Check()`:

```
bool Check(int n, Predicate<int> predicate)
{
    return predicate(n);
}
```

Với cách này, khi gọi phương thức `Check()`, chúng ta có thể tùy ý quy định điều kiện cần kiểm tra dưới dạng anonymous method hoặc biểu thức Lambda. Ví dụ:

- Kiểm tra n có phải số dương hay không:

```
Check(5, x => x > 0);          // Trả về: true
```

– Kiểm tra n có phải số chẵn hay không:

```
Check(11, x => x % 2 == 0);    // Trả về: false
```

Tham số **predicate** của phương thức **Check()** sẽ trỏ tới phương thức điều kiện được quy định bởi anonymous hoặc biểu thức Lambda. Do đó, gọi **predicate()** tức là gọi phương thức điều kiện nói trên.

VIII. Bài tập