Делегати. Анонімні функції

Делегати

Делегат – об’єкт, який може посилатися на метод.

Даний метод можна викликати за допомогою цього посилання.

Один делегат можна використовувати для виклику різних методів, змінюючи посилання на них під час виконання програми.

Загальна форма оголошення делегату

delegate тип_повернення ім'я(список_параметрів);

Приклади:

```
delegate void Del();
delegate int Deleg1(int i, double x, bool b);
delegate bool Deleg1(int i, double x, string s);
delegate bool Deleg2(int i2, double x, string s3);
delegate void Deleg3(int i, double x, string s);
```

Виклик методу

Делегат може викликати метод екземпляру класу або статичний метод

```
//Приклад
using System;
delegate void del();
class Program
{
    static void Main()
    {
        SayHelloWorld(); // викликаємо метод
        // створюємо об'єкт делегата dl і задаємо його посилання на метод SayHelloWorld()
        del dl = new del(SayHelloWorld);

        dl();  // викликаємо метод через делегат
    }

    static void SayHelloWorld()
    {
        Console.WriteLine("Hello World!");
    }
}
```

Приклад 2

```
using System;
// Оголошуємо делегат
delegate string strMod(string str);
class DelegateTest {
    static string replaceSpaces(string a)
    {
        Console.WriteLine("Заміна пробілів дефісами.");
        return a.Replace(' ', '-');
    }
    // Метод видаляє пробіли.
    static string removeSpaces(string a)
    {
        string temp = "";
        int i;
        Console.WriteLine("Видалення пробілів.");
        for (i = 0; i < a.Length; i++)
            if (a[i] != ' ') temp += a[i];
        return temp;
    }
    // Метод реверсує рядок.
    static string reverse(string a)
    {
        string temp = "";
```

Виклик нестатичного методу

Делегат може викликати метод екземпляру класу або статичний метод

```
//Приклад
using System;
delegate void del();
class Program
{
    static void Main()
    {
        Program pr = new Program(); // створюємо об'єкт
        pr.SayHelloWorld();         // викликаємо метод
        // створюємо об'єкт делегата dl і задаємо його посилання на метод SayHelloWorld()
        del dl = new del(pr.SayHelloWorld);
        dl();  // викликаємо метод через делегат
    }
    void SayHelloWorld()
    {
        Console.WriteLine("Hello World!");
    }
}
```

Багатоадресна передача (multicasting)

Multicasting – можливість делегата створювати ланцюжок викликів методів, які повинні автоматично викликатися при виклику делегата.

Для цього використовується оператор «+=»

Щоб видалити метод з ланцюжка – оператор «-=»

Обмеження – делегат повинен повертати тип даних – **void**.

Багатоадресна передача (multicasting). Приклад

```
using System;
delegate void del();
class Program
{
    static void Main()
    {
        Program pr = new Program(); // створюємо об'єкт
        // створюємо об'єкт делегата dl і задаємо його посилання на метод SayHelloWorld()
        del dl = new del(pr.SayHelloWorld);
        // додаємо до ланцюжка виклику методів ще 2 методи
        dl += new del(pr.SayHelloPNK);
        dl += new del(pr.SayHelloKN);
        dl();  // викликаємо 3 методи через делегат
    }

    void SayHelloWorld()
    {
        Console.WriteLine("Hello World!");
    }

    void SayHelloPNK()
```

Клас System.Delegate

Всі делегати являють собою класи, похідні від System.Delegate. Зазвичай члени цього класу не використовуються напряму.

Навіщо потрібні делегати?

1) Підтримують функціонування подій
2) Під час виконання програми дозволяють виконати метод, який точно невідомий в період компіляції.

Наприклад потрібно створити оболонку до якої могли б підключатися програмні компоненти. Уявіть графічну програму (на зразок стандартної утиліти Windows Paint). Використовуючи делегат, можна було б дозволити користувачеві підключати спеціальні кольорові світлофільтри або аналізатори зображень. Більш того, користувач міг би створювати "свої" послідовності цих фільтрів або аналізаторів. За допомогою делегатів організувати такий алгоритм дуже легко.

Приклад застосування делегата

1. Оголошуємо делегат в класі або за його межами з певною сигнатурою:

```
delegate void del();
```

2. Створюємо один (або декілька методів), сигнатура якого така сама, як в оголошеному делегаті

```
static void SayHelloWorld()
{
    Console.WriteLine("Hello World!");
}
```

Приклад застосування делегата

3. Створюємо об'єкт делегата dl і задаємо його посилання на метод SayHelloWorld():

```
del dl = new del(SayHelloWorld);
```

4. Викликаємо метод через делегат:

```
dl();
```

Приклад застосування делегата

```
using System;
delegate void del();
class Program
{
    static void Main()
    {
// створюємо об'єкт делегата dl і задаємо його посилання на метод SayHelloWorld()
        del dl = new del(SayHelloWorld);   // або del dl = SayHelloWorld;

        dl();  // викликаємо метод через делегат
    }

    static void SayHelloWorld()
    {
        Console.WriteLine("Hello World!");
    }
}
```

Анонімні функції

Метод, на який посилається делегат, нерідко використовується тільки для цієї мети. Іншими словами, єдиною підставою для існування методу є та обставина, що він може бути викликаний за допомогою делегата, але сам по собі він не викликається взагалі.

У подібних випадках можна скористатися **анонімною функцією**, щоб не створювати окремий метод.

Анонімна функція, по суті, являє собою безіменний кодовий блок, який передається конструктору делегата. Перевага анонімної функції полягає, зокрема, в її простоті. Завдяки їй відпадає необхідність оголошувати окремий метод, єдине призначення якого полягає в тому, що він передається делегату.

Починаючи з версії 3.0, в C# передбачено **два різновиди анонімних функцій** - **анонімні методи** і **лямбда-вирази**

Анонімні методи

Анонімні методи почали застосовуватися в C# ще у версії 2.0, а лямбда-вирази - у версії 3.0.

В цілому лямбда-вираз удосконалює принцип дії анонімного методу і в даний час вважається кращим для створення анонімної функції. Але анонімні методи широко застосовуються в існуючому коді C# і тому як і раніше є важливою складовою частиною C#. А оскільки анонімні методи передували появі лямбда-виразів, то чітке уявлення про них дозволяє краще зрозуміти особливості лямбда-виразів. До того ж анонімні методи можуть бути використані в цілому ряді випадків, де застосування лямбда-виразів виявляється неможливим.

Анонімний метод – один із способів створення безіменного блоку коду, пов'язаного з конкретним екземпляром делегата. Для створення анонімного методу досить вказати кодовий блок після ключового слова **delegate**. Покажемо, як це робиться, на конкретному прикладі:

Анонімні методи (приклад)

```
using System;
delegate void del();
class Program
{
    static void Main()
    {
     del dl = delegate()
         {
             Console.WriteLine("Hello World!");
         };
         dl();  // викликаємо анонімний метод через делегат
    }

    static void SayHelloWorld()
    {
        Console.WriteLine("Hello World!");
    }
}
```

Лямбда-вирази

Починаючи з C# 3.0, доступний новий синтаксис для призначення реалізації коду делегатам, що називається ***лямбда-виразами*** *(lambda expression)*. Лямбда-вирази можуть використовуватися скрізь, де є параметр типу делегата.

Синтаксис лямбда-виразів простіший синтаксису анонімних методів.

У всіх лямбда-виразах застосовується новий лямбда-оператор **=>**, який розділяє лямбда-вираз на дві частини. У лівій його частині вказується вхідний параметр (або кілька параметрів), а в правій частині - тіло лямбда-виразу. Оператор => іноді описується такими словами, як "переходить" або "стає".

Лямбда-вирази

У C# підтримуються два різновиди лямбда-виразів в залежності від тіла самого лямбда-виразу.

Так, якщо тіло лямбда-виразу складається з одного виразу, то утворюється ***одиночний лямбда-вираз***. У цьому випадку тіло виразу не заключається в фігурні дужки.

Якщо ж тіло лямбда-виразу складається з блоку операторів, укладених у фігурні дужки, то утворюється ***блочний лямбда-вираз***. При цьому блочний лямбда-вираз може містити цілий ряд операторів, в тому числі цикли, виклики методів і умовні оператори if. Обидва різновиди лямбда-виразів розглядаються далі окремо.

Одиночні лямбда-вирази

В одиночному лямбда-виразі частина, яка перебуває праворуч від оператора **=>**, впливає на параметр (або ряд параметрів), що вказується зліва. Поверненим результатом обчислення такого виразу є результат виконання лямбда-оператора. Нижче наведена загальна форма одиночного лямбда-виразу, що приймає єдиний параметр:

параметр => вираз

Якщо ж потрібно вказати кілька параметрів, то використовується наступна форма:

(список_параметров) => вираз

Таким чином, коли потрібно вказати два параметра або більше, їх слід помістити в дужки. Якщо ж вираз не вимагає параметрів, то слід використовувати порожні дужки.

Лямбда-вираз застосовується в два етапи. Спочатку оголошується тип делегата, сумісний з лямбда-виразом, а потім екземпляр делегата, якому присвоюється лямбда-вираз. Після цього лямбда-вираз обчислюється при зверненні до примірника делегата. Результатом його обчислення стає повернене значення.

Одиночні лямбда-вирази (приклад)

```
using System;
delegate void del(int i);
class Program
{
    static void Main()
    {
        // лямбда-вираз
        del dl = () => Console.WriteLine("Hello World!");

        dl();  // викликаємо через делегат
    }

    static void SayHelloWorld()
    {
        Console.WriteLine("Hello World!");
    }
}
```

Одиночні лямбда-вирази (приклад 2)

```
using System;
delegate int del(int i1, int i2);
class Program
{
    static void Main()
    {
        // лямбда-вираз
        del dl = (i1, i2) => i1+i2;

        dl(3, 5);  // викликаємо через делегат
    }

    static int Sum(int i1, int i2)
    {
        return i1 + i2;
    }
}
```

Блокові лямбда-вирази

Другим різновидом лямбда-виразів є **блоковий лямбда-вираз**. Для такого лямбда-виразу характерні розширені можливості виконання різних операцій, оскільки в його тілі допускається вказувати кілька операторів. Наприклад, в блочному лямбда-виразі можна використовувати цикли і умовні оператори **if**, оголошувати змінні і т.д. Створити блоковий лямбда-вираз неважко. Для цього досить помістити тіло виразу в фігурні дужки. Крім можливості використовувати кілька операторів, в іншому блочний лямбда-вираз, практично нічим не відрізняється від щойно розглянутого одиночного лямбда-виразу.

Блокові лямбда-вирази (приклад)

```
using System;
delegate int del(int i1, int i2);
class Program
{
    static void Main()
    {
        // блоковий лямбда-вираз
        del dl = (i1, i2) =>
        {
            int i3 = i1 + i2;
            if (i1>0) return i3;
            else return i1;
        };
        dl(3, 5);  // викликаємо через делегат
    }

    static int Sum(int i1, int i2)
    {
        int i3 = i1 + i2;
        if (i1 > 0) return i3;
        else return i1;
    }
}
```

Завдання для перевірки засвоєння знань

Дано метод:

```
static int Method1(string s, int i)
{
    int i2 = s.Length * 3;
    Console.WriteLine(i2);
    return i2;
}
```

1. Перетворіть цей метод в анонімний метод і викличте за допомогою делегата.
2. Перетворіть цей метод в лямбда-вираз і викличте за допомогою делегата.

Методи розширення

Методи розширення (extension methods) дозволяють додавати нові методи в уже існуючі типи без створення нового похідного класу. Ця функціональність буває особливо корисна, коли нам хочеться додати в певний тип новий метод, але сам тип (клас або структуру) ми змінити не можемо.

Наприклад, нам треба додати для типу **string** новий метод:

Методи розширення

```
using System;
class Program
{
    static void Main(string[] args)
    {
        string s = "Привіт світ";
        char c = 'і';
        // виклик методу розширення
        int i = s.SymbolCount('і');
        Console.WriteLine(i);
        Console.ReadLine();
    }
}
```

Методи розширення

Для того, щоб створити метод розширення, спочатку треба створити статичний клас, який і буде містити цей метод. В даному випадку це клас **StringExtension**. Потім оголошуємо статичний метод. У прикладі це **SymbolCount()**. Суть нашого методу розширення – підрахунок кількості певних символів в рядку.

```
using System;
0 references
class Program...
0 references
public static class StringExtension
{
    1 reference
    public static int SymbolCount(this string str, char c)...
}
```

Методи розширення

Власне метод розширення – це звичайний статичний метод, який в якості першого параметра завжди приймає таку конструкцію: this імя_типу назва_параметра, тобто в нашому випадку this string str. Так як наш метод буде відноситися до типу string, то ми і використовуємо даний тип.

```
public static class StringExtension
{
    1 reference
    public static int SymbolCount(this string str, char c)...
}
```

Методи розширення

Потім у всіх рядків ми можемо викликати даний метод:

```
int i = s.SymbolCount('i');
```

Причому нам вже не треба вказувати перший параметр. Значення для інших параметрів передаються в звичайному порядку.

Методи розширення

Слід враховувати, що методи розширення діють на рівні простору імен. Тобто, якщо додати в проект інший простір імен, то метод не буде застосовуватися до рядків, і в цьому випадку треба буде підключити простір імен методу через директиву **using**.