عنوان: مروری بر کاربردهای Action و Func - قسمت اول
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۳:۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Refactoring

delegateها، نوع‌هایی هستند که ارجاعی را به یک متد دارند؛ بسیار شبیه به function pointers در C و CPP هستند، اما برخلاف آن‌ها، delegates شیءگرا بوده، به امضای متد اهمیت داده و همچنین کد مدیریت شده و امن به شمار می‌روند.
سیر تکاملی delegates را در مثال ساده زیر می‌توان ملاحظه کرد:

```
using System;

namespace ActionFuncSamples
{
    public delegate int AddMethodDelegate(int a);
    public class DelegateSample
    {
        public void UseDelegate(AddMethodDelegate addMethod)
        {
            Console.WriteLine(addMethod(5));
        }
    }

    public class Helper
    {
        public int CustomAdd(int a)
        {
            return ++a;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Helper helper = new Helper();

            // .NET 1
            AddMethodDelegate addMethod = new AddMethodDelegate(helper.CustomAdd);
            new DelegateSample().UseDelegate(addMethod);

            // .NET 2, anonymous delegates
            new DelegateSample().UseDelegate(delegate(int a) { return helper.CustomAdd(a); });

            // .NET 3.5
            new DelegateSample().UseDelegate(a => helper.CustomAdd(a));
        }
    }
}
```

معنای کلمه delegate، واگذاری مسئولیت است. به این معنا که ما در متد UseDelegate، نمی‌دانیم addMethod به چه نحوی تعریف خواهد شد. فقط می‌دانیم که امضای آن چیست.
در دات نت یک، یک وهله از شیء AddMethodDelegate ساخته شده و سپس متدی که امضایی متناسب و متناظر با آن را داشت، به عنوان متد انجام دهنده مسئولیت معرفی می‌شد. در دات نت دو، اندکی نحوه تعریف delegates با ارائه delegates بی‌نام، ساده‌تر شد و در دات نت سه و نیم با ارائه lambda expressions ، تعریف و استفاده از delegates باز هم ساده‌تر و زیباتر گردید.
به علاوه در دات نت 3 و نیم، دو Generic delegate به نام‌های Action و Func نیز ارائه گردیده‌اند که به طور کامل جایگزین تعریف طولانی delegates در کدهای پس از دات نت سه و نیم شده‌اند. تفاوت‌های این دو نیز بسیار ساده است:
اگر قرار است واگذاری قسمتی از کد را به متدی محول کنید که مقداری را بازگشت می‌دهد، از Func و اگر این متد خروجی ندارد از Action استفاده نمائید:

```
Action<int> example1 = x => Console.WriteLine("Write {0}", x);
example1(5);

Func<int, string> example2 = x => string.Format("{0:n0}", x);
Console.WriteLine(example2(5000));
```

در دو مثال فوق، نحوه تعریف inline یک Action و یا Func را ملاحظه می‌کنید. Action به متدی اشاره می‌کند که خروجی ندارد و در اینجا تنها یک ورودی int را می‌پذیرد. Func در اینجا به تابعی اشاره می‌کند که یک ورودی int را دریافت کرده و یک خروجی string را باز می‌گرداند.

پس از این مقدمه، در ادامه قصد داریم مثال‌های دنیای واقعی Action و Func را که در سال‌های اخیر بسیار متداول شده‌اند، بررسی کنیم.

مثال یک) ساده سازی تعاریف API ارائه شده به استفاده کنندگان از کتابخانه‌های ما

عنوان شد که کار delegates، واگذاری مسئولیت انجام کاری به کلاس‌های دیگر است. این مورد شما را به یاد کاربردهای interfaceها نمی‌اندازد؟

در interfaceها نیز یک قرارداد کلی تعریف شده و سپس کدهای یک کتابخانه، تنها با امضای متدها و خواص تعریف شده در آن کار می‌کنند و کتابخانه ما نمی‌داند که این متدها قرار است چه پیاده سازی خاصی را داشته باشند.

برای نمونه طراحی API زیر را درنظر بگیرید که در آن یک interface جدید تعریف شده که تنها حاوی یک متد است. سپس کلاس Runner از این interface استفاده می‌کند:

```
using System;

namespace ActionFuncSamples
{
    public interface ISchedule
    {
        void Run();
    }

    public class Runner
    {
        public void Exceute(ISchedule schedule)
        {
            schedule.Run();
        }
    }

    public class HelloSchedule : ISchedule
    {
        public void Run()
        {
            Console.WriteLine("Just Run!");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            new Runner().Exceute(new HelloSchedule());
        }
    }
}
```

در اینجا ابتدا باید این interface را در طی یک کلاس جدید (مثلا HelloSchedule) پیاده سازی کرد و سپس حاصل را در کلاس Runner استفاده نمود.

نظر شما در مورد این طراحی ساده شده چیست؟

```
using System;

namespace ActionFuncSamples
{
    public class Schedule
    {
        public void Exceute(Action run)
        {
            run();
        }
    }

    class Program
```

```
    {
        static void Main(string[] args)
        {
            new Schedule().Exceute(() => Console.WriteLine("Just Run!"));
        }
    }
}
```

با توجه به اینکه هدف از معرفی interface در طراحی اول، واگذاری مسئولیت نحوه تعریف متد Run به کلاسی دیگر است، به همین طراحی با استفاده از یک Action delegate نیز می‌توان رسید. مهم‌ترین مزیت آن، حجم بسیار کمتر کدنویسی استفاده کننده نهایی از API تعریف شده ما است. به علاوه امکان inline coding نیز فراهم گردیده است و در همان محل تعریف Action، بدنه آن‌را نیز می‌توان تعریف کرد.

بدیهی است delegates نمی‌توانند به طور کامل جای interfaceها را پر کنند. اگر نیاز است قرارداد تهیه شده بین ما و استفاده کنندگان از کتابخانه، حاوی بیش از یک متد باشد، استفاده از interfaceها بهتر هستند.

از دیدگاه بسیاری از طراحان API، اشیاء delegate معادل interface ایی با یک متد هستند و یک وهله از delegate معادل وهله‌ای از کلاسی است که یک interface را پیاده سازی کرده‌است.

علت استفاده بیش از حد interfaceها در سایر زبان‌ها برای ابتدایی‌ترین کارها، کمبود امکانات پایه‌ای آن زبان‌ها مانند نداشتن anonymous methods، lambda expressions و anonymous delegates هستند. به همین دلیل مجبورند همیشه و در همه‌جا از interfaceها استفاده کنند.

ادامه دارد ...

نظرات خوانندگان

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۷:۳۰

نمیشه همیشه اینطور گفت. بستگی به کاری داره که قرار هست انجام بشه. اینترفیس IComparable که فقط متد CompareTo رو داره، یک مثال نقض هست.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۷:۳۷

طراحی IComparable مربوط به زمان دات نت یک است. اگر آن زمان امکانات زبان مثل امروز بود، می‌شد از طراحی ساده‌تری استفاده کرد.
یک نمونه از طراحی‌های اخیر تیم دات نت رو میشه در WebGrid دید. در این طراحی برای نمونه جهت دریافت فرمول فرمت کردن مقدار یک cell، از Func استفاده کردن. می‌شد این رو با اینترفیس هم نوشت (چون قرار است کاری به خارج از کلاس محول شود و هر بار اطلاعاتی به آن ارسال و نتیجه‌ای جدید اخذ گردد؛ پیاده سازی آن با شما، نتیجه را فقط در اختیار WebGrid ما قرار دهید). اما جدا استفاده از آن تبدیل می‌شد به عذاب برای کاربر که به نحو زیبایی با Func و امکانات جدید زبان حل شده.

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۸:۵۱

فکر نمی‌کنم به خاطر دات نت 1 باشه. دلیلی فراتر از این وجود داره. با کمی جستجو، این لینک که بر اساس VS 2010 نوشته شده، در پاراگراف آخر دلیل منطقی‌تری رو ارائه میده. در مورد WebGrid که فرمودید، بحثش جداست. من از کامپونت‌های متن باز Telerik در بستر ASP.NET MVC استفاده می‌کنم و از انعطاف پذیری Action و Func در متدهای اون لذت می‌برم. حرف من در مورد تعریف واجب استفاده از Predefined Delegates به جای اینترفیس‌های تک متدی است.

```
One good example of using a single-method interface instead of a delegate is

IComparable

or the generic version,

<IComparable<T

.

IComparable

declares the

CompareTo

method, which returns an integer that specifies a less than, equal to,
or greater than relationship between two objects of the same type.

IComparable

can be used as the basis of a sort algorithm. Although using a delegate
 comparison method as the basis of a sort algorithm would be valid, it
is not ideal. Because the ability to compare belongs to the class and
the comparison algorithm does not change at run time, a single-method
interface is ideal.
```

نویسنده: وحید نصیری

تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۹:۱۱

- در مورد تعریف «واجب» کسی اینجا بحث نکرده. این هم یک دید طراحی است. آیا کسی می‌تونه بگه اولین طراحی مطرح شده در مطلب جاری اشتباه است؟ خیر. اما ضرورتی ندارد تا این اندازه صرفا جهت واگذاری مسئولیت انجام یک متد به کلاسی دیگر، اینقدر طراحی انجام شده زمخت و طولانی باشد.
- در متن MSDN فوق نوشته شده که استفاده از delegate در این حالت خاص نیز معتبر است؛ اما ایده‌آل نیست. دلیلی که آورده از نظر من ساختگی است. ضرورتی ندارد تعریف یک delegate معرفی شده در runtime عوض شود. یا عنوان کرده که IComparable پایه مرتب سازی یک سری از متدها است. خوب ... بله زمانیکه از روز اول اینطور طراحی کردید همه چیز به هم مرتبط خواهند بود.

پ.ن.
قسمت نظرات MSDN یک زمانی باز بود ولی ... بعد از مدتی پشیمان شدند و به نظر این قابلیت منسوخ شده در این سایت!