عنوان: **انجام اعمال ریاضی بر روی Generics**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۰۱ ۰:۴۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Generics

کامپایلر سی‌شارپ اگر نتواند نوع‌های عملوندها را در حین بکارگیری عملگرها تشخیص دهد، اجازه‌ی استفاده از عملگر را نخواهد داد و کار کامپایل، با یک خطا خاتمه می‌یابد. برای نمونه مثال زیر را در نظر بگیرید:

```
public interface ICalculator<T>
    {
        T Add(T operand1, T operand2);
    }

    public class Calculator<T> : ICalculator<T>
    {
        public T Add(T operand1, T operand2)
        {
            return operand1 + operand2;
        }
    }
```

در اینجا چون کامپایلر نمی‌داند که عملگر + بر روی چه نوع‌هایی قرار است اعمال شود (به علت جنریک تعریف شدن این نوع‌ها و مشخص نبودن اینکه آیا این نوع، اصلا عملگر + دارد یا خیر)، با صدور خطای زیر، عملیات کامپایل را متوقف می‌کند:

```
Operator '+' cannot be applied to operands of type 'T' and 'T'
```

برای حل این مساله، چندین روش مطرح شده‌است که در ادامه تعدادی از آن‌ها را مرور خواهیم کرد.

روش اول: واگذار کردن استراتژی عملیات ریاضی به یک کلاس خارجی

این راه حلی است که توسط اعضای تیم سی‌شارپ در روزهای ابتدایی معرفی جنریک‌ها مطرح شده‌است. فرض کنید می‌خواهیم لیستی از جنریک‌ها را با هم جمع بزنیم:

```
public class Calculator2<T>
    {
        public T Sum(List<T> list)
        {
            T sum = 0;
            for (int i = 0; i < list.Count; i++)
                sum += list[i];
            return sum;
        }
    }
```

این کد نیز قابل کامپایل نبوده و امکان اعمال عملگر + بر روی نوع ناشناخته‌ی T میسر نیست.

```
public interface ICalculator<T>
    {
        T Add(T operand1, T operand2);
    }

    public class Int32Calculator : ICalculator<int>
    {
        public int Add(int operand1, int operand2)
        {
            return operand1 + operand2;
        }
    }

    public class AlgorithmLibrary<T> where T : new()
    {
        private readonly ICalculator<T> _calculator;
        public AlgorithmLibrary(ICalculator<T> calculator)
        {
```

```
            _calculator = calculator;
        }

        public T Sum(List<T> items)
        {
            var sum = new T();
            for (var i = 0; i < items.Count; i++)
            {
                sum = _calculator.Add(sum, items[i]);
            }
            return sum;
        }
    }
```

در راه حل ارائه شده، یک اینترفیس عمومی که متد جمع را تعریف کرده‌است، مشاهده می‌کنیم. سپس این اینترفیس در سازنده‌ی کتابخانه‌ی الگوریتم‌های برنامه تزریق شده‌است. اکنون کدهای AlgorithmLibrary بدون مشکل کامپایل می‌شوند. هر زمان که نیاز به استفاده از آن بود، بر اساس نوع T، پیاده سازی خاصی را باید ارائه داد. برای مثال در اینجا Int32Calculator پیاده سازی نوع int را انجام داده‌است. برای استفاده از آن نیز خواهیم داشت:

```
var result = new AlgorithmLibrary<int>(new Int32Calculator()).Sum(new List<int> { 1, 2, 3 });
```

البته این نوع پیاده سازی را که کار اصلی آن واگذاری عملیات جمع، به یک کلاس خارجی است، توسط Func نیز می‌توان خلاصه‌تر کرد:

```
public class Algorithms<T> where T : new()
    {
        public T Calculate(Func<T, T, T> add, IEnumerable<T> numbers)
        {
            var sum = new T();
            foreach (var number in numbers)
            {
                sum = add(sum, number);
            }
            return sum;
        }
    }
```

استفاده از Action و Func نیز یکی دیگر از روش‌های تزریق وابستگی‌ها است که در اینجا بکار گرفته شده‌است. برای استفاده از آن خواهیم داشت:

```
var result = new Algorithms<int>().Calculate((a, b) => a + b, new[] { 1, 2, 3 });
```

آرگومان اول روش جمع زدن را مشخص می‌کند و آرگومان دوم، لیستی است که باید اعضای آن جمع زده شوند.

روش دوم: استفاده از واژه‌ی کلیدی dynamic

با استفاده از واژه‌ی کلیدی dynamic می‌توان بررسی نوع داده‌ها را به زمان اجرا موکول کرد. به این ترتیب دیگر کامپایلر مشکلی با کامپایل قطعه کد ذیل نخواهد داشت:

```
public class Calculator<T> : ICalculator<T>
    {
        public T Add(T operand1, T operand2)
        {
            return (dynamic)operand1 + operand2;
        }
    }
```

و مثال زیر نیز به خوبی کار می‌کند:

```
var test = new Calculator<int>().Add(1, 2);
```

البته بدیهی است که نوع تعریف شده در اینجا باید دارای عملگر + باشد. در غیر اینصورت در زمان اجرا برنامه با یک خطا خاتمه خواهد یافت.
روش فوق نسبت به حالتی که بر اساس نوع T تصمیم‌گیری شود و از عملگر + متناظری استفاده گردد، خوانایی بهتری دارد:

```
public T Add(T t1, T t2)
{
    if (typeof(T) == typeof(double))
    {
        var d1 = (double)t1;
        var d2 = (double)t2;
        return (T)(d1 + d2);
    }
    else if (typeof(T) == typeof(int)){
        var i1 = (int)t1;
        var i2 = (int)t2;
        return (T)(i1 + i2);
    }
    else ...
}
```

روش سوم: استفاده از Expression Trees

روش زیر بسیار شبیه است به حالتیکه از Func در روش اول استفاده شد. در اینجا این Func به صورت پویا تولید و سپس صدا زده می‌شود:

```
using System;
using System.Linq.Expressions;

namespace GenericsArithmetic
{
    public class Solution3
    {
        public T Add<T>(T a, T b)
        {
            var paramA = Expression.Parameter(typeof(T), "a");
            var paramB = Expression.Parameter(typeof(T), "b");

            var body = Expression.Add(paramA, paramB);
            var add = Expression.Lambda<Func<T, T, T>>(body, paramA, paramB).Compile();
            return add(a, b);
        }
    }
}
```

البته این مثال، یک مثال ابتدایی در این مورد است. بر همین مبنا و ایده، یک کتابخانه‌ی با کارآیی بالا، تحت عنوان Generic Operators که جزو Misc utils می‌باشد، تهیه شده‌است.
به کمک کتابخانه‌ی Generic Operators، کدهای جمع زدن اعضای یک لیست جنریک به صورت ذیل خلاصه می‌شوند:

```
public static T Sum<T>(this IEnumerable<T> source)
{
    T sum = Operator<T>.Zero;
    foreach (T value in source)
    {
            sum = Operator.Add(sum, value);
    }
    return sum;
}
```

نظرات خوانندگان

نویسنده: سجاد
تاریخ: ۱۳۹۳/۰۴/۰۸ ۱۲:۵۶

c++ احتیاجی به این نوع پیاده سازی‌های دشوار با استفاده از روش‌های غیرمعمول رو نداره. هرچند خودم هم یکی از طرفدارهای پروپا قرص c# هستم ولی genericهای c# در مقابل templateهای c++ کمبود دارند. هرچند همیشه **عاشق** c# بودم ولی generic های c# هیچوقت انتظارات منو برآورده نکرد.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۰۸ ۱۳:۱۸

C# generics مانند C++ templates نیستند . آرگومان‌های C# generics در زمان اجرا دریافت و پردازش می‌شوند، در حالیکه C++ templates مانند یک compiler macro عمل کرده و در زمان کامپایل و پیش از اجرا به صورت کامل دریافت، بررسی و الحاق خواهند شد. به همین جهت است که C++ templates می‌توانند برای مثال تشخیص دهند، آرگومان مورد استفاده، دارای عملگر + هست یا خیر.
پردازش در زمان اجرای آرگومان‌های جنریک این مزیت را به همراه دارد که بتوانید بدون نیاز به الحاق سورس آرگومان‌های مورد استفاده (چون برخلاف C++ templates ، ریز اطلاعات آن‌ها کامپایل نمی‌شوند)، کتابخانه‌ای را برای عموم منتشر کنید.