عنوان: **ساخت فرم های generic در WPF و Windows Application**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۴/۰۹ ۸:۳۰
آدرس: www.dotnettips.info
برچسب‌ها: WPF, Generics

آیا می‌توان در یک پروژه های Windows App یا WPF، یک فرم پایه به صورت generic تعریف کنیم و سایر فرم‌ها بتوانند از آن ارث ببرند؟ در این پست به تشریح و بررسی این مسئله خواهیم پرداخت.
در پروژه هایی به صورت Smart UI کد نویسی شده اند و یا حتی قصد انجام پروژه با تکنولوژی‌های WPF یا Windows Application را دارید و نیاز دارید که فرم‌های خود را به صورت generic بسازید این مقاله به شما کمک خواهد کرد.

Windows Application#

یک پروژه از نوع Windows Application ایجاد می‌کنیم و یک فرم به نام FrmBase در آن خواهیم داشت. یک Label در فرم قرار دهید و مقدار Text آن را فرم اصلی قرار دهید.
در فرم مربوطه، فرم را به صورت generic تعریف کنید. به صورت زیر:

```
public partial class FrmBase<T> : Form where T : class
    {
        public FrmBase()
        {
            InitializeComponent();
        }
    }
```

بعد باید همین تغییرات را در فایل FrmBase.designer.cs هم اعمال کنیم:

```
partial class FrmBase<T> where T : class
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise,
false.</param>
        protected override void Dispose( bool disposing )
        {
            if ( disposing && ( components != null ) )
            {
                components.Dispose();
            }
            base.Dispose( disposing );
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.label1 = new System.Windows.Forms.Label();
            this.SuspendLayout();
            //
            // label1
            //
            this.label1.AutoSize = true;
            this.label1.Location = new System.Drawing.Point(186, 22);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(51, 13);
            this.label1.TabIndex = 0;
            this.label1.Text = "فرم اصلی";
            //
            // FrmBase
```

```
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(445, 262);
            this.Controls.Add(this.label1);
            this.Name = "FrmBase";
            this.Text = "Form1";
            this.ResumeLayout(false);
            this.PerformLayout();

        }

        #endregion

        private System.Windows.Forms.Label label1;
    }
```

یک فرم جدید بسازید و نام آن را FrmTest بگذارید. این فرم باید از FrmBase ارث ببرد. خب این کار را به صورت زیر انجام می‌دهیم:

```
public partial class FrmTest : FrmBase<String>
    {
        public FrmTest()
        {
            InitializeComponent();
        }
    }
```

پروژه را اجرا کنید. بدون هیچ گونه مشکلی برنامه اجرا می‌شود وفرم مربوطه را در حالت اجرا مشاهده خواهید کرد. اما اگر قصد باز کردن فرم FrmTest را در حالت design داشته باشید با خطای زیر مواجه خواهید شد:

با این که برنامه به راحتی اجرا می‌شود و خروجی آن قابل مشاهده است ولی امکان نمایش فرم در حالت design وجود ندارد. متاسفانه در Windows Appها برای تعریف فرم‌ها به صورت generic یا این مشکل روبرور هستیم. تنها راه موجود برای حل این مشکل استفاده از یک کلاس کمکی است. به صورت زیر:

```
public partial class FrmTest : FrmTestHelp
    {
        public FrmTest()
        {
            InitializeComponent();
        }
    }

    public class FrmTestHelp : FrmBase<String>
    {
    }
```

مشاهده می‌کنید که بعد از اعمال تغییرات بالا فرم FrmTest به راحتی Load می‌شود و در حالت designer هم می‌تونید از آن استفاده کنید.

WPF#

در پروژه‌های WPF ، راه حلی برای این مشکل در نظر گرفته شده است. در WPF، برای Window یا UserControl پایه نمی‌توان

Designer داشت. ابتدا باید فرم پایه را به صورت زیر ایجاد کنیم:

```
public class WindowBase<T> : Window where T : class
{
}
```

در این مرحله یک Window بسازید که از WindowBase ارث ببرد:

```
public partial class MainWindow: WindowBase<String>
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
```

در WPF باید تعاریف موجود برای Xaml و Code Behind یکی باشد. در نتیجه باید تغییرات زیر را در فایل Xaml نیز اعمال کنید:

```
<local:WindowBase x:Class="GenericWindows.MainWindow"

        x:TypeArguments="sys:String"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:GenericWindows"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</local:WindowBase>
```

همان طور که می‌بینید در ابتدای فایل به جای Window از local:WindowBase استفاده شده است. این نشان دهنده این است که فرم پایه برای این Window از نوع WindowBase است. برای مشخص کردن نوع generic هم می‌تونید از x:TypeArguments استفاده کنید که در این جا نوع آن را String انتخاب کردم.

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۴/۰۹ ۱۹:۶

با تشکر. لطفا یک مثال دنیای واقعی از این فرم جنریک بزنید.

نویسنده: مسعود م.پاکدل
تاریخ: ۱۳۹۲/۰۴/۱۰ ۹:۳۷

یک مثال پیاده سازی شده رو می‌تونید (^) اینجا مشاهده کنید.

نویسنده: محمدی راوری
تاریخ: ۱۳۹۲/۰۵/۰۹ ۱۴:۲۴

با سلام و تشکر از آموزش ارائه شده
من در ساخت برنامه مشکلی نداشتم و اون رو ساختم اما در مرحله ای که لازم بود تا نمایش فرم ساخته شده را فعال کنم با مشکل برخوردم.
اگر ممکنه راهنمایی کنید؛ با سپاس فراوان.

نویسنده: مسعود م.پاکدل
تاریخ: ۱۳۹۲/۰۵/۰۹ ۲۰:۲۶

مشکل در قسمت نمایش در حالت Design بوده است یا اجرا؟
اگر امکانش هست مشکل مربوطه را دقیق عنوان کنید.

عنوان: **استفاده از OfType یا Cast در Linq**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۴/۲۸ ۲۰:۳۰
آدرس: www.dotnettips.info
برچسب‌ها: LINQ, Generics, Casting, Collections

تقریبا تمام توسعه دهندگان دات نت با تکنولوژی Linq و Lambda Expressionها آشنایی دارند. همان طور که می‌دانیم Extension Methodهای موجود در فضای نام System.Linq فقط بر روی مجموعه ای از داده‌ها که اینترفیس IEnumerable<t> که در فضای نام System.Collections.Generic قرار دارد را پیاده سازی کرده باشند قابل اجرا هستند. مجموعه داده‌های جنریک فقط قابلیت نگهداری از یک نوع داده که به عنوان پارامتر T برای این مجموعه تعریف می‌شود را داراست.
نکته: البته در مجموعه هایی نظیر Dictionary یا سایر Collectionها امکان تعریف چند نوع داده به عنوان پارامتر وجود دارد. نکته مهم این است که داده‌های استفاده شده در این مجموعه ها، حتما باید از نوع پارامتر تعریف شده باشند.
اگر در یک مجموعه داده قصد داشته باشیم که داده هایی با نوع مختلف را ذخیره کنیم و در جای مناسب آن‌ها را بازیابی کرده و در برنامه استفاده نماییم چه باید کرد. به عنوان یک پیشنهاد می‌توان از مجموعه‌های موجود در فضای نام System.Collection بهره بگیریم. اما همان طور که واضح است این مجموعه از داده‌ها به صورت جنریک نمی‌باشند و امکان استفاده از Queryهای Linq در آن‌ها به صورت معمول امکان پذیر نیست. برای حل این مشکل در دات نت دو متد تعبیه شده است که وظیفه آن تبدیل این مجموعه از داده‌ها به مجموعه ای است که بتوان بر روی آن‌ها Queryهای از جنس Linq یا Lambda Expression را اجرا کرد.
Cast
OfType

#مثال 1
فرض کنید یک مجموعه مثل زیر داریم:

```
ArrayList myList = new ArrayList();

  myList.Add( "Value1" );
  myList.Add( "Value2" );
  myList.Add( "Value3" );

  var myCollection = myList.Cast<string>();
```

در مثال بالا یک Collection از نوع ArrayList ایجاد کردیم که در فضای نام System.Collection قرار دارد. شما در این مجموعه می‌توانید از هر نوع داده ای که مد نظرتان است استفاده کنید. با استفاده از اپراتور Cast توانستیم این مجموعه را به نوع مورد نظر خودمان تبدیل کنیم و در نهایت به یک مجموعه از IEnumerable<T> برسیم. حال امکان استفاده از تمام متدهای Linq امکان پذیر است.
#مثال دوم:

```
ArrayList myList = new ArrayList();

    myList.Add( "Value1" );
    myList.Add( 10 );
    myList.Add( 10.2 );

   var myCollection = myList.Cast<string>();
```

در مثال بالا در خط آخر با یک runtime Error مواجه خواهیم شد. دلیلش هم این است که ما از در ArrayList خود داده‌های غیر از string نظیر int یا double داریم. درنتیجه هنگام تبدیل داده‌های int یا double به string یک Exception رخ خواهد داد. در این گونه موارد که در لیست مورد نظر داده‌های غیر هم نوع وجود دارد باید متد OfType را جایگزین کنیم.

```
ArrayList myList = new ArrayList();

myList.Add( "Value1" );
myList.Add( 10 );
myList.Add( 10.2 );
```

```
var doubleNumber = myList.OfType<double>().Single();
var integerNumber = myList.OfType<int>().Single();
var stringValue = myList.OfType<string>().Single();
```

تفاوت بین متد Cast و OfType در این است که متد Cast سعی دارد تمام داده‌های موجود در مجموعه را به نوع مورد نظر تبدیل کند ولی متد OfType فقط داده‌های از نوع مشخص شده را برگشت خواهد داد. حتی اگر هیچ آیتمی از نوع مورد نظر در این مجموعه نباشد یک مجموعه بدون هیچ داده ای برگشت داده می‌شود.

عنوان: **آشنایی با جنریک‌ها #1**
نویسنده: امیر هاشم زاده
تاریخ: ۱۳۹۲/۱۰/۲۹ ۱۶:۱۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Generics, Generic Classes

طبق این معرفی ، جنریک‌ها باعث می‌شوند که نوع داده‌ای (data type) المان‌های برنامه در زمان استفاده از آن‌ها در برنامه مشخص شوند. به عبارت دیگر، جنریک به ما اجازه می‌دهد کلاس‌ها یا متدهایی بنویسیم که می‌توانند با هر نوع داده‌ای کار کنند.

نکاتی از جنریک‌ها:
برای به حداکثر رسانی استفاده مجدد از کد، type safety و کارایی است.
بیشترین استفاده مشترک از جنریک‌ها جهت ساختن کالکشن کلاس‌ها (collection classes) است.
تا حد ممکن از جنریک کالکشن کلاسها (generic collection classes) جدید فضای نام System.Collections.Generic بجای کلاس‌هایی مانند ArrayList در فضای نام System.Collections استفاده شود.
شما می‌توانید *اینترفیس جنریک* ، *کلاس جنریک* ، متد جنریک و *عامل* جنریک سفارشی خودتان تهیه کنید.
جنریک کلاس‌ها، ممکن است در دسترسی به متدهایی با نوع داده‌ای خاص محدود شود.
بوسیله reflection، می‌توانید اطلاعاتی که در یک جنریک در زمان اجرا (run-time) قرار دارد بدست آورید.

انواع جنریک ها:
کلاس‌های جنریک
اینترفیس‌های جنریک
متدهای جنریک
عامل‌های جنریک

در قسمت اول به معرفی کلاس جنریک می‌پردازیم.
کلاس‌های جنریک کلاس جنریک یعنی کلاسی که می‌تواند با چندین نوع داده کار کند برای آشنایی با این نوع کلاس به کد زیر دقت کنید:

```
using System;
using System.Collections.Generic;

namespace GenericApplication
{
    public class MyGenericArray<T>
    {
        // تعریف یک آرایه از نوع جنریک
        private T[] array;

        public MyGenericArray(int size)
        {
            array = new T[size + 1];
        }

        // بدست آوردن یک آیتم جنریک از آرایه جنریک
        public T getItem(int index)
        {
            return array[index];
        }

        // افزودن یک آیتم جنریک به آرایه جنریک
        public void setItem(int index, T value)
        {
            array[index] = value;
        }
    }
}
```

در کد بالا کلاسی تعریف شده است که می‌تواند بر روی آرایه‌هایی از نوع داده‌ای مختلف عملیات درج و حذف را انجام دهد. برای تعریف کلاس جنریک کافی است عبارت <T> بعد از نام کلاس خود اضافه کنید، سپس همانند سایر کلاس‌ها از این نوع داده ای در

کلاس استفاده کنید. در مثال بالا یک آرایه از نوع T تعریف شده است که این نوع، در زمان استفاده مشخص خواهد شد. (یعنی در زمان استفاده از کلاس مشخص خواهد شد که چه نوع آرایه ای ایجاد می‌شود)

در کد زیر نحوه استفاده از کلاس جنریک نشان داده شده است، همانطور که مشاهده می‌کنید نوع کلاس int و char در نظر گرفته شده است (نوع کلاس، زمان استفاده از کلاس مشخص می‌شود) و سپس آرایه هایی از نوع int و char ایجاد شده است و 5 آیتم از نوع int و char به آرایه‌های هم نوع افزوده شده است.

```
class Tester
{
        static void Main(string[] args)
        {
            // تعریف یک آرایه از نوع عدد صحیح
            MyGenericArray<int> intArray = new MyGenericArray<int>(5);

            // افزودن اعداد صحیح به آرایه ای از نوع عدد صحیح
            for (int c = 0; c < 5; c++)
            {
                intArray.setItem(c, c*5);
            }

            // بدست آوردن آیتم‌های آرایه ای از نوع عدد صحیح
            for (int c = 0; c < 5; c++)
            {
                Console.Write(intArray.getItem(c) + " ");
            }
            Console.WriteLine();

            // تعریف یک آرایه از نوع کاراکتر
            MyGenericArray<char> charArray = new MyGenericArray<char>(5);

            // افزودن کاراکترها به آرایه ای از نوع کاراکتر
            for (int c = 0; c < 5; c++)
            {
                charArray.setItem(c, (char)(c+97));
            }

            // بدست آوردن آیتم‌های آرایه ای از نوع کاراکتر
            for (int c = 0; c< 5; c++)
            {
                Console.Write(charArray.getItem(c) + " ");
            }
            Console.WriteLine();
            Console.ReadKey();
        }
}
```

زمانی که کد بالا اجرا می‌شود خروجی زیر بدست می‌آید:

```
0 5 10 15 20
a b c d e
```

عنوان: **آشنایی با جنریک‌ها #2**
نویسنده: امیر هاشم زاده
تاریخ: ۱۳۹۲/۱۰/۳۰ ۱۲:۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Generics, Generic Interfaces

قبل از ادامه آموزش مفاهیم جنریک، در نظر داشتن این نکته ضروری است که مطالبی که در این سری مقالات ارائه می‌شود در سطح مقدماتی است و قصد من آشنا نمودن برنامه نویسانی است که با این مفاهیم ناآشنا هستند ولی با مطالعه این مقاله می‌توانند کدهای تمیزتر و بهتری تولید کنند و همینطور این مفاهیم ساده، پایه‌ای باشد برای فراگیری سایر نکات تکمیلی و پیچیده‌تر جنریک‌ها.

در قسمت قبلی ، نحوه تعریف کلاس جنریک شرح داده شد و در سری دوم اشاره‌ای به مفاهیم و نحوه پیاده سازی اینترفیس جنریک می‌پردازیم.
مفهوم اینترفیس جنریک همانند مفهوم اینترفیس در دات نت است. با این تفاوت که برای آن‌ها یک نوع عمومی تعریف می‌شود و نوع آن‌ها در زمان اجرا تعیین خواهد شد و کلاس بر اساس نوع اینترفیس، اینترفیس را پیاده سازی می‌کند. برای درک بهتر به نحوه تعریف اینترفیس جنریک زیر دقت کنید:

```
public interface IBinaryOperations<T>
{
   T Add(T arg1, T arg2);
   T Subtract(T arg1, T arg2);
   T Multiply(T arg1, T arg2);
   T Divide(T arg1, T arg2);
}
```

در کد بالا اینترفیسی از نوع جنریک تعریف شده است که دارای چهار متد با چهار خروجی و پارامترهای چنریک می‌باشد که نوع خروجی‌ها و نوع پارامترهای ورودی در زمان استفاده از اینترفیس تعیین می‌شوند که البته در بالا بطور خاص بیان شده است.
اینترفیسی داریم که دو ورودی از هر نوعی دریافت می‌کند و چهار عملی اصلی را بر روی آن‌ها انجام داده و خروجی آن‌ها را از همان نوع پارامتر ورودی تولید می‌کند. (بجای اینترفیس‌های مختلف عملیات چهار عمل اصلی برای هر نوع داده (data type)، یک اینترفیس کلی برای تمام data type‌ها)
در کلاس زیر نحوه پیاده سازی اینترفیس از نوع int را مشاهده می‌کنید که چهار عملی اصلی را برروی داده هایی از نوع int انجام می‌شود و چهار خروجی از نوع int تولید می‌شود.

```
public class BasicMath : IBinaryOperations<int>
{
   public int Add(int arg1, int arg2)
   { return arg1 + arg2; }

   public int Subtract(int arg1, int arg2)
   { return arg1 - arg2; }

   public int Multiply(int arg1, int arg2)
   { return arg1 * arg2; }

   public int Divide(int arg1, int arg2)
   { return arg1 / arg2; }
}
```

بعد از پیاده سازی اینترفیس حال نوبت به استفاده از کلاس می‌رسد که زیر نیز نحوه استفاده از کلاس نمایش داده شده است:

```
static void Main(string[] args)
{
   Console.WriteLine("***** Generic Interfaces *****\n");
   BasicMath m = new BasicMath();
   Console.WriteLine("1 + 1 = {0}", m.Add(1, 1));
   Console.ReadLine();
}
```

و در صورتیکه بخواهید کلاسی چهار عمل اصلی را بر روی نوع داده double انجام دهد کافیست کلاسی اینترفیس نوع double را

پیاده سازی کرده باشد. مانند کد زیر:

```
public class BasicMath : IBinaryOperations<double>
{
   public double Add(double arg1, double arg2)
   { return arg1 + arg2; }
   ...
}
```

برداشتی آزاد از این مقاله .

عنوان: **آشنایی با جنریک‌ها #3**
نویسنده: امیر هاشم زاده
تاریخ: ۱۳۹۳/۰۱/۲۱ ۲۳:۴۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Generics

متدهای جنریک

متدهای جنریک، دارای پارامترهایی از نوع جنریک هستند و بوسیله‌ی آنها می‌توانیم نوع‌های (type) متفاوتی را به متد ارسال نمائیم. در واقع از متد، یک نمونه پیاده سازی کرده‌ایم، در حالیکه این متد را برای انواع دیگر هم می‌توانیم فراخوانی کنیم.

تعریف ساده دیگر

جنریک متدها اجازه می‌دهند متدهایی با نوع هایی که در زمان فراخوانی مشخص کرده ایم، داشته باشیم.

نحوه تعریف یک متد جنریک بشکل زیر است:

```
return-type method-name<type-parameters>(parameters)
```

قسمت مهم syntax بالا، **type-parameters** است. در آن قسمت می‌توانید یک یا چند نوع که بوسیله کاما از هم جدا می‌شوند را تعریف کنید. این typeها در return-value و نوع برخی یا همه پارامترهای ورودی جنریک متد، قابل استفاده هستند. به کد زیر توجه کنید:

```
public T1 PrintValue<T1, T2>(T1 param1, T2 param2)
{
    Console.WriteLine("values are: parameter 1 = " + param1 + " and parameter 2 = " + param2);
    return param1;
}
```

در کد بالا، دو پارامتر ورودی بترتیب از نوع T1 و T2 و پارامتر خروجی (return-type) از نوع T1 تعریف کرده‌ایم.

اعمال محدودیت بر روی جنریک متدها

در زمان تعریف یک جنریک کلاس یا جنریک متد، امکان اعمال محدودیت بر روی typeهایی را که قرار است به آن‌ها ارسال شود، داریم. یعنی می‌توانیم تعیین کنیم جنریک متد چه typeهایی را در زمان ایجاد یک وهله‌ی از آن بپذیرد یا نپذیرد. اگر نوعی که به جنریک متد ارسال می‌کنیم جزء محدودیت‌های جنریک باشد با خطای کامپایلر روبرو خواهیم شد. این محدودیت‌ها با کلمه کلیدی where اعمال می‌شوند.

```
public void MyMethod< T >()
       where T : struct
{
  ...
}
```

محدودیت‌های قابل اعمال بر روی جنریک ها

struct: نوع آرگومان ارسالی باید value-type باشد؛ بجز مقادیر غیر NULL.

```
class C<T> where T : struct {} // value type
```

class: نوع آرگومان ارسالی باید reference-type (کلاس، اینترفیس، عامل، آرایه) باشد.

```
class D<T> where T : class {} // reference type
```

new(): آرگومان ارسالی باید یک سازنده عمومی بدون پارامتر باشد. وقتی این محدوده کننده را با سایر محدود کننده‌ها به صورت همزمان استفاده می‌کنید، این محدوده کننده باید در آخر ذکر شود.

```
class H<T> where T : new() {} // no parameter constructor
```

```
public void MyMethod< T >()
       where T : IComparable, MyBaseClass, new ()
{
  ...
}
```

<base class name>: نوع آرگومان ارسالی باید از کلاس ذکر شده یا کلاس مشتق شده آن باشد.

```
class B {}
class E<T> where T : B {} // be/derive from base class
```

<interface name>: نوع آرگومان ارسالی باید اینترفیس ذکر شده یا پیاده ساز آن اینترفیس باشد.

```
interface I {}
class G<T> where T : I {} // be/implement interface
```

U: نوع آرگومان ارسالی باید از نوع یا مشتق شده U باشد.

```
class F<T, U> where T : U {} // be/derive from U
```

توجه: در مثال‌های بالا، محدوده کننده‌ها را برای جنریک کلاس‌ها اعمال کردیم که روش تعریف این محدودیت‌ها برای جنریک متدها هم یکسان است.

اعمال چندین محدودیت همزمان

برای اعمال چندین محدودیت همزمان بر روی یک آرگومان فقط کافی است محدودیت‌ها را پشت سرهم نوشته و آنها را بوسیله کاما از یکدیگر جدا نمایید.

```
interface I {}
class J<T>
  where T : class, I
```

در کلاس J بالا، برای آرگومان محدودیت **class** و **اینترفیس I** را اعمال کرده‌ایم.

این روش قابل تعمیم است:

```
interface I {}
class J<T, U>
  where T : class, I
  where U : I, new() {}
```

در کلاس J، آرگومان T با محدودیت‌های class و اینترفیس I و آرگومان U با محدودیت اینترفیس I و new() تعریف شده است و البته تعداد آرگومان‌ها قابل گسترش است.

حال سوال این است: **چرا از محدود کننده‌ها استفاده می‌کنیم؟**

کد زیر را در نظر بگیرید:

```
//this method returns if both the parameters are equal
public static bool Equals< T > (T t1, Tt2)
{
  return (t1 == t2);
}
```

متد بالا برای مقایسه دو نوع یکسان استفاده می‌شود. در مثال بالا در صورتیکه دو مقدار از نوع int با هم مقایسه نماییم جنریک متد بدرستی کار خواهد کرد ولی اگر بخواهیم دو مقدار از نوع string را مقایسه کنیم با خطای کامپایلر مواجه خواهیم شد. عمل مقایسه دو مقدار از نوع string که مقادیر در heap نگهداری می‌شوند بسادگی مقایسه دو مقدار int نیست. چون همانطور که می‌دانید int یک value-type و string یک reference-type است و برای مقایسه دو reference-type با استفاده از عملگر ==

تمهیداتی باید در نظر گرفته شود.
برای حل مشکل بالا 2 راه حل وجود دارد:
Runtime casting
استفاده از محدود کننده‌ها

casting در زمان اجرا، بعضی اوقات شاید مناسب باشد. در این مورد، CLR نوع‌ها را در زمان اجرا بدلیل کارکرد صحیح بصورت اتوماتیک cast خواهد کرد اما مطمئناً این روش همیشه مناسب نیست مخصوصاً زمانی که نوع‌های مورد استفاده در حال تحریف رفتار طبیعی عملگرها باشند (مانند آخرین نمونه بالا).

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.