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.

عنوان: **مقید سازی پارامترهای نوع جنریک**
نویسنده: عثمان رحیمی
تاریخ: ۱۳۹۳/۱۱/۲۵ ۱۰:۱۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Generics

احتمالا در بیشتر مقالات (فارسی/انگلیسی) عبارات هایی مثل نمونه‌های زیر را دیده اید :

```
where T:clas
where T:struc
...
```

در این مقاله قصد داریم بپردازیم به «مقید سازی پارامترهای نوع جنریک» و اینکه چه کاربردی دارند و در چه زمانی بهتر است از آن‌ها استفاده کنیم و نحوه استفاده از آنها چگونه است. فرض میکنیم که خواننده‌ی محترم با مفاهیم جنریک آشنایی دارد. در صورتیکه با جنریک‌ها آشنا نیستید ابتدا مروری داشته باشید بر ج نریک‌ها و بعد این مقاله را مطالعه فرمایید؛ به این دلیل که موضوع مورد بحث بر پایه‌ی جنریک‌ها می‌باشد.

همانطور که مطلع هستید هر عنصری جنریکی را که تعریف میکنید حداقل دارای یک پارامتر نوع هست و در زمان بکارگیری آن جنریک باید نوع آن را مشخص نمایید. برای نمونه مثال زیر را در نظر بگیرید :

```
public  class MyCollection<T>
        {
            private List<T> collections = new List<T>();
            public void Add(T value)
            {
                collections.Add(value);
            }
        }
```

کلاس فوق یک کلاس جنریک است که در هنگام ساخت نمونه‌ای از آن، باید ابتدا data type نوعی را که که می‌خواهیم با آن کار کنیم، تعیین کنیم. برای مثال در کد فوق در هنگام ساخت نمونه‌ای از آن، نوع int را برای آن مشخص میکنیم و هر وقت بخواهیم متد Add آن را فراخوانی کنیم، فقط نوعی را قبول خواهد کرد که در ابتدا برای آن تعیین کرده ایم (int):

```
MyCollection<int> myintObj = new MyCollection<int>();
            myintObj.Add(12);
            myintObj.Add(33);
             myintObj.Add(33.3);// ERROR z
```

سؤال: می‌خواهیم فقط نوع‌هایی را بتوان به T نسبت داد که از نوع ارجاعی (reference type) هستن و یا فقط نوع هایی را به T نسبت داد که یک سازنده دارند؛ چگونه؟

ایجاد قیدها یا محدودیت‌ها بر روی پارامترهای جنریک‌ها شامل پنج حالت می‌باشد:

حالت اول : Where T:struct
در این حالت T باید یک ساختار باشد .

حالت دوم : where T:class
T باید یک نوع ارجاعی باشد. اگر در مثال فوق این قید را به آن اضافه کنیم، در هنگام ساخت نمونه‌ای از کلاس فوق، اگر یک نوع value type را به T نسبت دهیم، در هنگام وارد کردن یک نوع value type با خطا مواجه خواهیم شد. مثال:

```
public  class MyCollection<T> where T:class
        {
            private List<T> collections = new List<T>();
            public void Add(T value)
            {
                collections.Add(value);
            }
        }
```

و برای استفاده :

```
MyCollection<int> myintObj = new MyCollection<int>(); // ERROR , int is value type
```

حالت سوم : Where T:new()
نوعی که به T نسبت داده می‌شود باید یک سازنده‌ی پیش فرض داشته باشد.
داخل پرانتز : سازنده‌ی پیش فرض: زمانی که شما یک کلاس می‌نویسید اگر آن کلاس دارای هیچ سازنده‌ای نباشد، کامپایلر یک سازنده‌ی بدون پارامتر را به کلاس فوق اضافه می‌کند که کار آن مقدار دهی به فیلدهای کلاس است. در اینجا از مقادیر پیش فرض استفاده می‌شود. مثلا برای int مقدار صفر و برای string مقدار "" و به همین ترتیب.
اگر از مقدار دهی پیش فرض توسط کامپایلر خرسند نیستید، می‌توانید سازنده پیش فرض را تغییر داده و مطابق میل خود فیلدها را مقدار دهی اولیه کنید .

حالت چهارم : where T:NameOfBaseClass
نوعی که به T نسبت داده می‌شود باید از کلاس NameOfBaseClass ارث بری کرده باشد.

حالت پنجم : where T:NameOfInterface
همانند حالت چهارم می‌باشد؛ با این تفاوت: نوعی که به T نسبت داده می‌شود باید واسط NameOfInterface را پیاده سازی کرده باشد.

پنج حالت فوق نمونه‌هایی از ایجاد محدودیت بر روی پرامتر نوع اعضای جنریک بودند و اما در ادامه قصد داریم نکاتی را در این باب، بیان کنیم:

نکته اول : می‌توانید محدودیت‌های فوق را با هم ترکیب کنید برای اینکار آنها را با کاما از هم جدا کنید :

```
public  class MyCollection<T> where T:class,IDisposable,new()
        {
   //content
}
```

نوعی که به T نسبت داده می‌شود
باید از نوع ارجاعی باشد.
باید واسط IDisposable را پیاده سازی کرده باشد.
باید یک سازنده‌ی پیش فرض داشته باشد.

نکته دوم : زمانیکه از چندین محدودیت استفاده می‌کنید مثل مثال فوق، باید محدودیت ()new در آخرین جایگاه محدودیت‌ها قرار گیرد؛ در غیر اینصورت با خطای زمان ترجمه روبه رو خواهید شد .

نکته سوم : می‌توان محدودیت‌های فوق را علاوه بر کلاس، بر روی متدهای جنریک نیز اعمال کنید:

```
public void Swap<T>(ref T val1,ref T val2) where T:struct
            {
//content
            }
```

نکته چهارم : زمانیکه کلاس و یا متدهای شما بیش از یک نوع پارامتر از نوع جنریک را دریافت می‌کنند، باید محدودیت‌های مورد نظر را برای هر کدام به صورت جداگانه قید کنید. به طور مثال به کلاس زیر که دو پارمتر T و K را دارد، باید برای هر کدام جداگانه محدودیت‌های مورد نظر را اعمال کنیم (در صورت نیاز):

```
public  class MyCollection<T,K> where T:class where K:IDisposable,new()
```

```
        {
//content
}
```

عنوان: **ساختار داده‌های خطی Linear Data Structure قسمت دوم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۱۲/۰۱ ۲:۱۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Generics, Data Structures, Linked List

در قسمت قبلی به مقدمات و ساخت لیست‌های ایستا و پویا به صورت دستی پرداختیم و در این قسمت (مبحث پایانی) لیست‌های آماده در دات نت را مورد بررسی قرار می‌دهیم.

کلاس ArrayList

این کلاس همان پیاده سازی لیست‌های ایستایی را دارد که در مطلب پیشین در مورد آن صحبت کردیم و نحوه کدنویسی آن نیز بیان شد و امکاناتی بیشتر از آنچه که در جدول مطلب پیشین گفته بودیم در دسترس ما قرار می‌دهد. از این کلاس با اسم untyped dynamically-extendable array به معنی *آرایه پویا قابل توسعه بدون نوع* هم اسم می‌برند چرا که به هیچ نوع داده‌ای مقید نیست و می‌توانید یکبار به آن رشته بدهید، یکبار عدد صحیح، یکبار اعشاری و یکبار زمان و تاریخ، کد زیر به خوبی نشان دهنده‌ی این موضوع است و نحوه استفاده‌ی از این آرایه‌ها را نشان می‌دهد.

```
using System;
using System.Collections;

class ProgrArrayListExample
{
    static void Main()
    {
        ArrayList list = new ArrayList();
        list.Add("Hello");
        list.Add(5);
        list.Add(3.14159);
        list.Add(DateTime.Now);

        for (int i = 0; i < list.Count; i++)
        {
            object value = list[i];
            Console.WriteLine("Index={0}; Value={1}", i, value);
        }
    }
}
```

نتیجه کد بالا:

```
Index=0; Value=Hello
Index=1; Value=5
Index=2; Value=3.14159
Index=3; Value=29.02.2015 23:17:01
```

البته برای خواندن و قرار دادن متغیرها از آنجا که فقط نوع Object را برمی‌گرداند، باید یک تبدیل هم انجام داد یا اینکه از کلمه‌ی کلیدی dynamic استفاده کنید:

```
ArrayList list = new ArrayList();
list.Add(2);
list.Add(3.5f);
list.Add(25u);
list.Add(" ریال");
dynamic sum = 0;
for (int i = 0; i < list.Count; i++)
{
    dynamic value = list[i];
    sum = sum + value;
}
Console.WriteLine("Sum = " + sum);
// Output: Sum = 30.5ریال
```

مجموعه‌های جنریک Generic Collections

مشکل ما در حین کار با کلاس arrayList و همه کلاس‌های مشتق شده از system.collection.IList این است که نوع داده‌ی ما

تبدیل به Object می‌شود و موقعی‌که آن را به ما بر می‌گرداند باید آن را به صورت دستی تبدیل کرده یا از کلمه‌ی کلیدی dynamic استفاده کنیم. در نتیجه در یک شرایط خاص، هیچ تضمینی برای ما وجود نخواهد داشت که بتوانیم کنترلی بر روی نوع داده‌های خود داشته باشیم و به علاوه عمل تبدیل یا casting هم یک عمل زمان بر هست.
برای حل این مشکل، از جنریک‌ها استفاده می‌کنیم. جنریک‌ها می‌توانند با هر نوع داده‌ای کار کنند. در حین تعریف یک کلاس جنریک نوع آن را مشخص می‌کنیم و مقادیری که از آن به بعد خواهد پذیرفت، از نوعی هستند که ابتدا تعریف کرده‌ایم.
یک ساختار جنریک به صورت زیر تعریف می‌شود:

```
GenericType<T> instance = new GenericType<T>();
```

نام کلاس و به جای T نوع داده از قبیل int,bool,string را می‌نویسیم. مثال‌های زیر را ببینید:

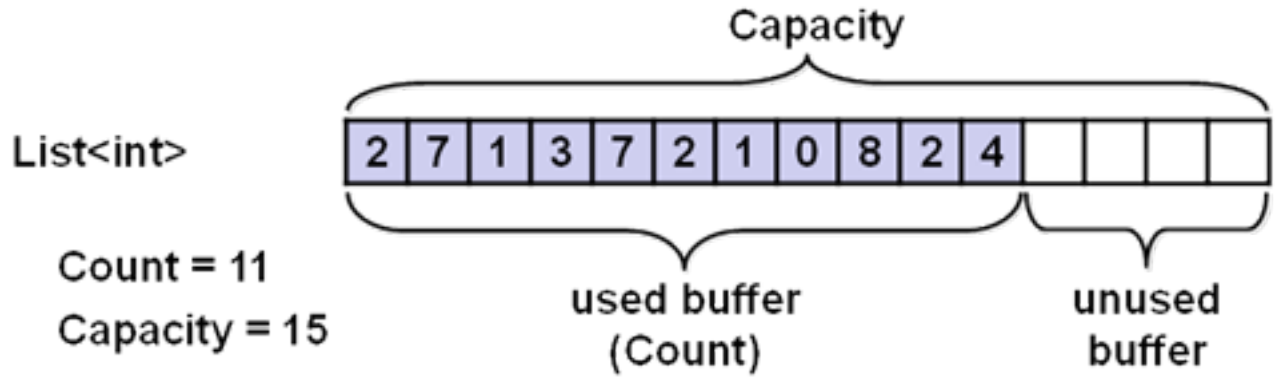
```
List<int> intList = new List<int>();
List<bool> boolList = new List<bool>();
List<double> realNumbersList = new List<double>();
```

کلاس جنریک List<T>

این کلاس مشابه همان کلاس ArrayList است و فقط به صورت جنریک پیاده سازی شده است.

```
List<int> intList = new List<int>();
```

تعریف بالا سبب ایجاد ArrayList ـی می‌باشد که تنها مقادیر int را دریافت می‌کند و دیگر نوع Object ـی در کار نیست. یک آرایه از نوع int ایجاد می‌کند و مقدار خانه‌های پیش فرضی را نیز در ابتدا، برای آن در نظر می‌گیرد و با افزودن هر مقدار جدید می‌بیند که آیا خانه‌ی خالی وجود دارد یا خیر. اگر وجود داشته باشد مقدار جدید، به خانه‌ی بعدی آخرین خانه‌ی پر شده انتقال می‌یابد و اگر هم نباشد، مقدار خانه از آن چه هست 2 برابر می‌شود. درست است عملیات resizing یا افزایش طول آرایه عملی زمان بر محسوب میشود ولی همیشه این اتفاق نمی‌افتد و با زیاد شدن مقادیر خانه‌ها این عمل کمتر هم می‌شود. هر چند با زیاد شدن خانه‌ها حافظه مصرفی ممکن است به خاطر زیاد شدن خانه‌های خالی بدتر هم بشود. فرض کنید بار اول خانه‌ها 16 تایی باشند که بعد می‌شوند 32 تایی و بعدا 64 تایی. حالا فرض کنید به خاطر یک عنصر، خانه‌ها یا ظرفیت بشود 128 تایی در حالی که طول آرایه (خانه‌های پر شده) 65 تاست و حال این وضعیت را برای موارد بزرگتر پیش بینی کنید. در این نوع داده اگر منظور زمان باشد نتجه خوبی را در بر دارد ولی اگر مراعات حافظه را هم در نظر بگیرید و داده‌ها زیاد باشند، باید تا حدامکان به روش‌های دیگر هم فکر کنید.



چه موقع از List<T> استفاده کنیم؟

استفاده از این روش مزایا و معایبی دارد که باید در توضیحات بالا متوجه شده باشید ولی به طور خلاصه:
استفاده از index برای دسترسی به یک مقدار، صرف نظر از اینکه چه میزان داده‌ای در آن وجود دارد، بسیار سریع انجام میگیرد.
جست و جوی یک عنصر بر اساس مقدار: جست و جو خطی است در نتیجه اگر مقدار مورد نظر در آخرین خانه‌ها باشد بدترین وضعیت ممکن رخ می‌دهد و بسیار کند عمل می‌کند. داده هر چی کمتر بهتر و هر چه بیشتر بدتر. البته اگر بخواهید مجموعه‌ای از مقدارهای برابر را برگردانید هم در بدترین وضعیت ممکن خواهد بود.

حذف و درج (منظور insert) المان‌ها به خصوص موقعی که انتهای آرایه نباشید، شیفت پیدا کردن در آرایه عملی کاملا کند و زمانبر است.
موقعی که عنصری را بخواهید اضافه کنید اگر ظرفیت آرایه تکمیل شده باشد، نیاز به عمل زمانبر افزایش ظرفیت خواهد بود که البته این عمل به ندرت رخ می‌دهد و عملیات افزودن Add هم هیچ وابستگی به تعداد المان‌ها ندارد و عملی سریع است.

با توجه به موارد خلاصه شده بالا، موقعی از لیست اضافه می‌کنیم که عملیات درج و حذف زیادی نداریم و بیشتر برای افزودن مقدار به انتها و دسترسی به المان‌ها بر اساس اندیس باشد.

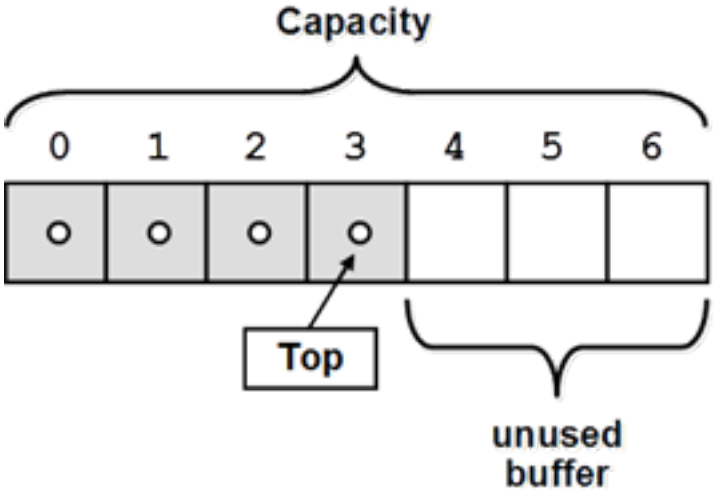
LinkedList<T>

یک کلاس از پیش آماده در دات نت که لیست‌های پیوندی دو طرفه را پیاده سازی می‌کند. هر المان یا گره یک متغیر جهت ذخیره مقدار دارد و یک اشاره گر به گره قبل و بعد. **چه موقع باید از این ساختار استفاده کنیم؟**

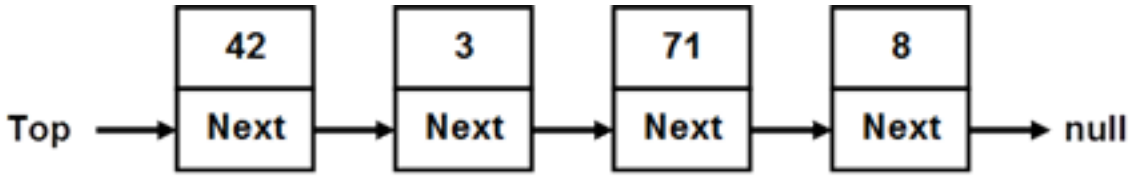
از مزایا و معایب آن :
افزودن به انتهای لیست به خاطر این که همیشه گره آخر در tail وجود دارد بسیار سریع است.
عملیات درج insert در هر موقعیتی که باشد اگر یک اشاره گر به آن محل باشد یک عملیات سریع است یا اینکه درج در ابتدا یاانتهای لیست باشد.
جست و جوی یک مقدار چه بر اساس اندیس باشد و چه مقدار، کار جست و جو کند خواهد بود. چرا که باید تمامی المان‌ها از اول به آخر اسکن بشن.
عملیات حذف هم به خاطر اینکه یک عمل جست و جو در ابتدای خود دارد، یک عمل کند است.

استفاده از این کلاس موقعی خوب است که عملیات‌های درج و حذف ما در یکی از دو طرف لیست باشد یا اشاره‌گری به گره مورد نظر وجود داشته باشد. از لحاظ مصرف حافظه به خاطر داشتن فیلدهای اشاره‌گر به جز مقدار، زیادتر از نوع List می‌باشد. در صورتی که دسترسی سریع به داده‌ها برایتان مهم باشد استفاده از List باز هم به صرفه‌تر است.

پشته Stack
یک سری مکعب را تصور کنید که روی هم قرار گرفته اند و برای اینکه به یکی از مکعب‌های پایینی بخواهید دسترسی داشته باشید باید تعدادی از مکعب‌ها را از بالا بردارید تا به آن برسید. یعنی بر خلاف موقعی که آن‌ها روی هم می‌گذاشتید و آخرین مکعب روی همه قرار گرفته است. حالا همان مکعب‌ها به صورت مخالف و معکوس باید برداشته شوند.
یک مثال واقعی‌تر و ملموس‌تر، یک کمد لباس را تصور کنید که مجبورید برای آن که به لباس خاصی برسید، باید آخرین لباس‌هایی را که در داخل کمد قرار داده‌اید را اول از همه از کمد در بیاورید تا به آن لباس برسید.
در واقع پشته چنین ساختاری را پیاده می‌کند که اولین عنصری که از پشته بیرون می‌آید، آخرین عنصری است که از آن درج شده است و به آن LIFO گویند که مخفف عبارت Last Input First Output آخرین ورودی اولین خروجی است. این ساختار از قدیمی‌ترین ساختارهای موجود است. حتی این ساختار در سیستم‌های داخل دات نت CLR هم به عنوان نگهدارنده متغیرها و پارامتر متدها استفاده می‌شود که به آن Program Execution Stack می‌گویند.
پشته سه عملیات اصلی را پیاده سازی می‌کند: **Push** جهت قرار دادن مقدار جدید در پشته، **POP** جهت بیرون کشیدن مقداری که آخرین بار در پشته اضافه شده و **Peek** جهت برگرداندن آخرین مقدار اضافه شده به پشته ولی آن مقدار از پشته حذف نمی‌شود.
این ساختار میتواند پیاده سازی‌های متفاوتی را داشته باشد ولی دو نوع اصلی که ما بررسی می‌کنیم، ایستا و پویا بودن آن است.
ایستا بر اساس آرایه است و پویا بر اساس لیست‌های پیوندی. شکل زیر پشته‌ای را به صورت استفاده از پیاده‌سازی ایستا با آرایه‌ها نشان می‌دهد و کلمه Top به بالای پشته یعنی آخرین عنصر اضافه شده اشاره می‌کند.



استفاده از لیست پیوندی برای پیاده سازی پشته:

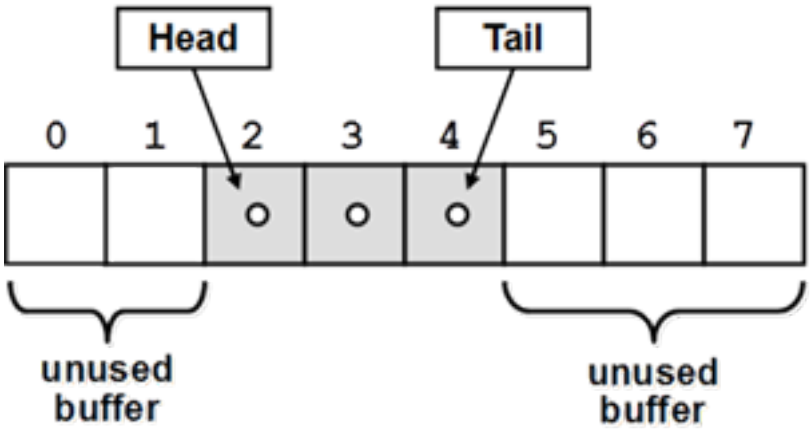


لیست پیوندی لازم نیست دو طرفه باشد و یک طرف برای کار با پشته مناسب است و دیگر لازم نیست که به انتهای لیست پیوندی عمل درج انجام شود؛ بلکه مقدار جدید به ابتدای آن اضافه شده و برای حذف گره هم اولین گره باید حذف شود و گره دوم به عنوان head شناخته می‌شود. همچنین لیست پیوندی نیازی به افزایش ظرفیت مانند آرایه‌ها ندارد.
ساختار پشته در دات نت توسط کلاس Stack از قبل آماده است:

```
Stack<string> stack = new Stack<string>();
stack.Push("A");
stack.Push("B");
stack.Push("C");
 while (stack.Count > 0)
    {
        string letter= stack.Pop();
        Console.WriteLine(letter);
    }
//خروجی
//C
//B
//A
```

صف Queue

ساختار صف هم از قدیمی‌ترین ساختارهاست و مثال آن در همه جا و در همه اطراف ما دیده می‌شود؛ مثل صف نانوایی، صف چاپ پرینتر، دسترسی به منابع مشترک توسط سیستمها. در این ساختار ما عنصر جدید را به انتهای صف اضافه می‌کنیم و برای دریافت مقدار، عنصر را از ابتدا حذف می‌کنیم. به این ساختار FIFO مخفف First Input First Output به معنی اولین ورودی و اولین خروجی هم می‌گویند.
ساختار ایستا که توسط آرایه‌ها پیاده سازی شده است:

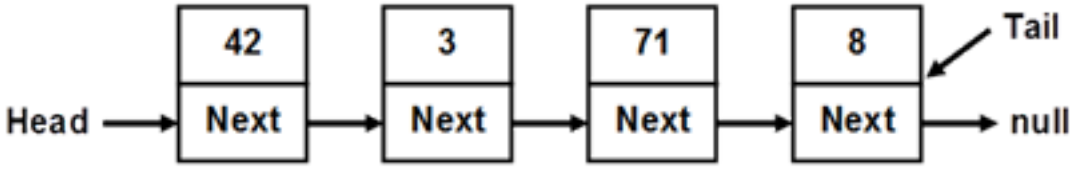


ابتدای آرایه مکانی است که عنصر از آنجا برداشته می‌شود و Head به آن اشاره می‌کند و tail هم به انتهای آرایه که جهت درج عنصر جدید مفید است. با برداشتن هر خانه‌ای که head به آن اشاره می‌کند، head یک خانه به سمت جلو حرکت می‌کند و زمانی که Head از tail بیشتر شود، یعنی اینکه دیگر عنصری یا المانی در صف وجود ندارد و head و Tail به ابتدای صف حرکت می‌کنند. در این حالت موقعی که المان جدیدی قصد اضافه شدن داشته باشد، افزودن، مجددا از اول صف آغاز می‌شود و به این صف‌ها، صف حلقوی می‌گویند.

عملیات اصلی صف دو مورد هستند enqueue که المان جدید را در انتهای صف قرار می‌دهد و dequeue اولین المان صف را بیرون می‌کشد.

پیاده سازی صف به صورت پویا با لیست‌های پیوندی

برای پیاده سازی صف، لیست‌های پیوندی یک طرفه کافی هستند:



در این حالت عنصر جدید مثل سابق به انتهای لیست اضافه می‌شود و برای حذف هم که از اول لیست کمک می‌گیریم و با حذف عنصر اول، متغیر Head به عنصر یا المان دوم اشاره خواهد کرد.

کلاس از پیش آمده صف در دات نت Queue<T> است و نحوه‌ی استفاده آن بدین شکل است:

```
static void Main()
{
    Queue<string> queue = new Queue<string>();
    queue.Enqueue("Message One");
    queue.Enqueue("Message Two");
    queue.Enqueue("Message Three");
    queue.Enqueue("Message Four");

    while (queue.Count > 0)
    {
        string msg = queue.Dequeue();
        Console.WriteLine(msg);
    }
```

```
}
خروجی//
//Message One
//Message Two
//Message Thre
//Message Four
```