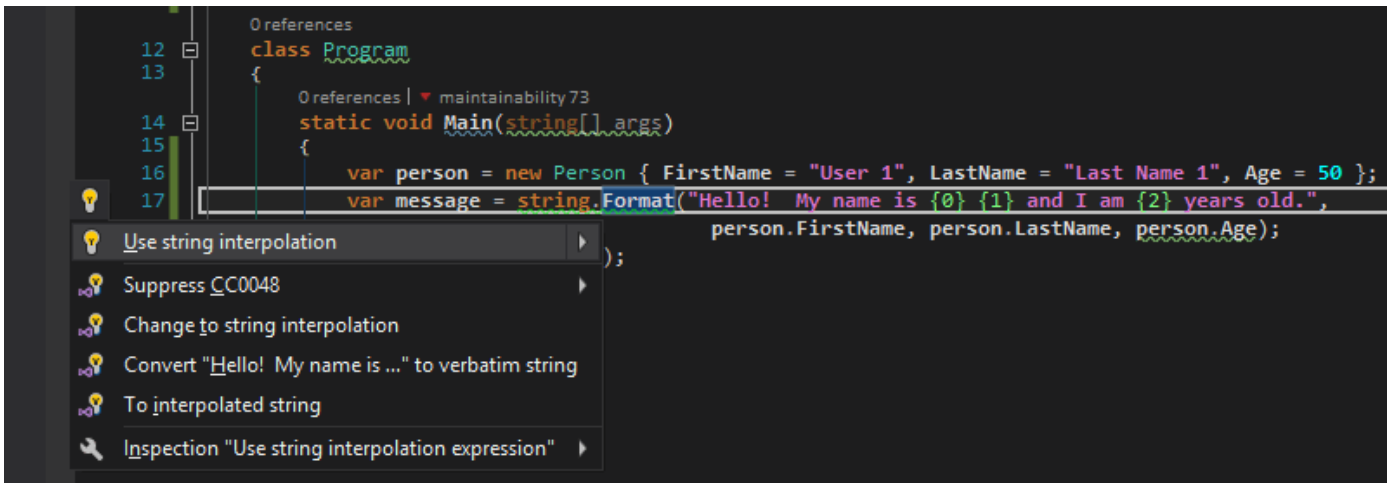
عنوان: **C# 6 - String Interpolation**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۷/۱۱ ۱۳:۴۵
آدرس: www.dotnettips.info
گروه‌ها: C#, C# 6.0

تا پیش از C# 6 یکی از روش‌های توصیه شده‌ی جهت اتصال رشته‌ها به هم، استفاده از متدهایی مانند string.Format و StringBuilder.AppendFormat بود:

```
using System;

namespace CS6NewFeatures
{
    class Person
    {
        public string FirstName { set; get; }
        public string LastName { set; get; }
        public int Age { set; get; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var person = new Person { FirstName = "User 1", LastName = "Last Name 1", Age = 50 };
            var message = string.Format("Hello!  My name is {0} {1} and I am {2} years old.",
                                        person.FirstName, person.LastName, person.Age);
            Console.Write(message);
        }
    }
}
```

مشکل این روش، کاهش خوانایی آن با بالا رفتن تعداد پارامترهای متد Format است و همچنین گاهی از اوقات فراموش کردن مقدار دهی بعضی از آن‌ها و یا حتی ذکر ایندکس‌هایی غیر معتبر که در زمان اجرا، برنامه را با یک خطا متوقف می‌کنند.
در C# 6 جهت رفع این مشکلات، راه حلی به نام String interpolation ارائه شده‌است و اگر افزونه‌ی ReSharper یا یکی از افزونه‌های Roslyn را نصب کرده باشید، به سادگی امکان تبدیل کدهای قدیمی را به فرمت جدید آن خواهید یافت:



در این حالت کد قدیمی فوق، به کد ذیل تبدیل خواهد شد:

```
static void Main(string[] args)
{
    var person = new Person { FirstName = "User 1", LastName = "Last Name 1", Age = 50 };
    var message = $"Hello!  My name is {person.FirstName} {person.LastName} and I am {person.Age} years
old.";
```

```
    Console.Write(message);
}
```

در اینجا ابتدا یک $ در ابتدای رشته قرار گرفته و سپس هر متغیر به داخل {} انتقال یافته‌است. همچنین دیگر نیازی هم به ذکر string.Format نیست.
عملیاتی که در اینجا توسط کامپایلر صورت خواهد گرفت، تبدیل این کدهای جدید مبتنی بر String interpolation به همان string.Format قدیمی در پشت صحنه‌است. بنابراین این قابلیت جدید C# 6 را به کدهای قدیمی خود نیز می‌توانید اعمال کنید. فقط کافی است VS 2015 را نصب کرده باشید و دیگر شماره‌ی دات نت فریم ورک مورد استفاده مهم نیست.

امکان انجام محاسبات با String interpolation

زمانیکه $ در ابتدای رشته قرار گرفت، عبارات داخل {}ها توسط کامپایلر محاسبه و جایگزین می‌شوند. بنابراین می‌توان چنین محاسباتی را نیز انجام داد:

```
var message2 = $"{Environment.NewLine}Test {DateTime.Now}, {3*2}";
Console.Write(message2);
```

بدیهی اگر $ ابتدای رشته فراموش شود، اتفاق خاصی رخ نخواهد داد.

تغییر فرمت عبارات نمایش داده شده توسط String interpolation

همانطور که با string.Format می‌توان نمایش سه رقم جدا کننده‌ی هزارها را فعال کرد و یا تاریخی را به نحوی خاص نمایش داد، در اینجا نیز همان قابلیت‌ها برقرار هستند و باید پس از ذکر یک : عنوان شوند:

```
var message3 = $"{Environment.NewLine}{1000000:n0} {DateTime.Now:dd-MM-yyyy}";
Console.Write(message3);
```

حالت کلی و استاندارد آن در متد string.Format به صورت {index[,alignment][:formatString]} است.

سفارشی سازی String interpolation

اگر متغیر رشته‌ای معرفی شده‌ی توسط $ را با یک var مشخص کنیم، نوع آن به صورت پیش فرض، از نوع string خواهد بود. برای نمونه در مثال‌های فوق، message و message2 از نوع string تعریف می‌شوند. اما این رشته‌های ویژه را می‌توان از نوع IFormattable و یا FormattableString نیز تعریف کرد.
در حقیقت رشته‌های آغاز شده‌ی با $ از نوع IFormattable هستند و اگر نوع متغیر آن‌ها ذکر نشود، به صورت خودکار به نوع FormattableString که اینترفیس IFormattable را پیاده سازی می‌کند، تبدیل می‌شوند. بنابراین پیاده سازی این اینترفیس، امکان سفارشی سازی خروجی string interpolation را میسر می‌کند. برای نمونه می‌خواهیم در مثال message2، نحوه‌ی نمایش تاریخ را سفارشی سازی کنیم.

```
class MyDateFormatProvider : IFormatProvider
{
    readonly MyDateFormatter _formatter = new MyDateFormatter();

    public object GetFormat(Type formatType)
    {
        return formatType == typeof(ICustomFormatter) ? _formatter : null;
    }

    class MyDateFormatter : ICustomFormatter
    {
        public string Format(string format, object arg, IFormatProvider formatProvider)
        {
            if (arg is DateTime)
                return ((DateTime)arg).ToString("MM/dd/yyyy");
```

```
            return arg.ToString();
        }
    }
}
```

در اینجا ابتدا کار با پیاده سازی اینترفیس IFormatProvider شروع می‌شود. متد GetFormat آن همیشه به همین شکل خواهد بود و هر زمانیکه نوع ارسالی به آن ICustomFormatter بود، یعنی یکی از اجزای {} دار در حال آنالیز است و خروجی مدنظر آن همیشه از نوع ICustomFormatter است که نمونه‌ای از پیاده سازی آن‌را جهت سفارشی سازی DateTime ملاحظه می‌کنید.
پس از پیاده سازی این سفارشی کننده‌ی تاریخ، نحوه‌ی استفاده‌ی از آن به صورت ذیل است:

```
static string formatMyDate(FormattableString formattable)
{
     return formattable.ToString(new MyDateFormatProvider());
}
```

ابتدا یک متد static را تعریف کنید که ورودی آن از نوع FormattableString باشد؛ از این جهت که رشته‌های شروع شده‌ی با $ نیز از همین نوع هستند. سپس سفارشی سازی پردازش {}ها در قسمت ToString آن انجام می‌شود و در اینجا می‌توان یک IFormatProvider جدید را معرفی کرد.
در ادامه برای اعمال این سفارشی سازی، فقط کافی است متد formatMyDate را به رشته‌ی مدنظر اعمال کنیم:

```
var message2 = formatMyDate($"{Environment.NewLine}Test {DateTime.Now}, {3*2}");
Console.Write(message2);
```

و اگر تنها می‌خواهید فرهنگ جاری را عوض کنید، از روش ساده‌ی زیر استفاده نمائید:

```
public static string faIr(IFormattable formattable)
{
    return formattable.ToString(null, new CultureInfo("fa-Ir"));
}
```

در اینجا با اعمال متد faIr به عبارت شروع شده‌ی با $، فرهنگ ایران به رشته‌ی جاری اعمال خواهد شد.
نمونه‌ی کاربردی‌تر آن اعمال InvariantCulture به String interpolation است:

```
static string invariant(FormattableString formattable)
{
    return formattable.ToString(CultureInfo.InvariantCulture);
}
```

یک نکته: همانطور که عنوان شد این قابلیت جدید با نگارش‌های قبلی دات نت نیز سازگار است؛ اما این کلاس‌های جدید را در این نگارش‌ها نخواهید یافت. برای رفع این مشکل تنها کافی است این کلاس‌های یاد شده را به صورت دستی در فضای نام اصلی آن‌ها تعریف و اضافه کنید. یک مثال

غیرفعال سازی String interpolation

اگر می‌خواهید در رشته‌ای که با $ شروع شده، بجای محاسبه‌ی عبارتی، دقیقا خود آن‌را نمایش دهید (و { را escape کنید)، از {{}} استفاده کنید:

```
var message0 = $"Hello! My name is {person.FirstName} {{person.FirstName}}";
```

در این مثال اولین {} محاسبه خواهد شد و دومی خیر.

پردازش عبارات شرطی توسط String interpolation

همانطور که عنوان شد، امکان ذکر یک عبارت کامل هم در بین {} وجود دارد (محاسبات، ذکر یک عبارت LINQ، ذکر یک متد و امثال آن). اما در این میان اگر یک عبارت شرطی مدنظر بود، باید بین () قرار گیرد:

```
Console.Write($"{(person.Age>50 ? "old": "young")}");
```

علت اینجا است که کامپایلر سی‌شارپ، : بین {} را به format specifier تفسیر می‌کند. نمونه‌ی آن‌را پیشتر با مثال «تغییر فرمت عبارات نمایش داده شده» ملاحظه کردید. ذکر : در اینجا به معنای شروع مشخص سازی فرمتی است که قرار است به این حاصل اعمال شود. برای تغییر این رفتار پیش فرض، کافی است عبارت مدنظر را بین () ذکر کنیم تا تمام آن به صورت یک عبارت سی‌شارپ تفسیر شود.

عنوان: C# 6 - The nameof Operator
نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۴/۰۷/۱۲ ۱۲:۲۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Refactoring, WPF, C# 6.0

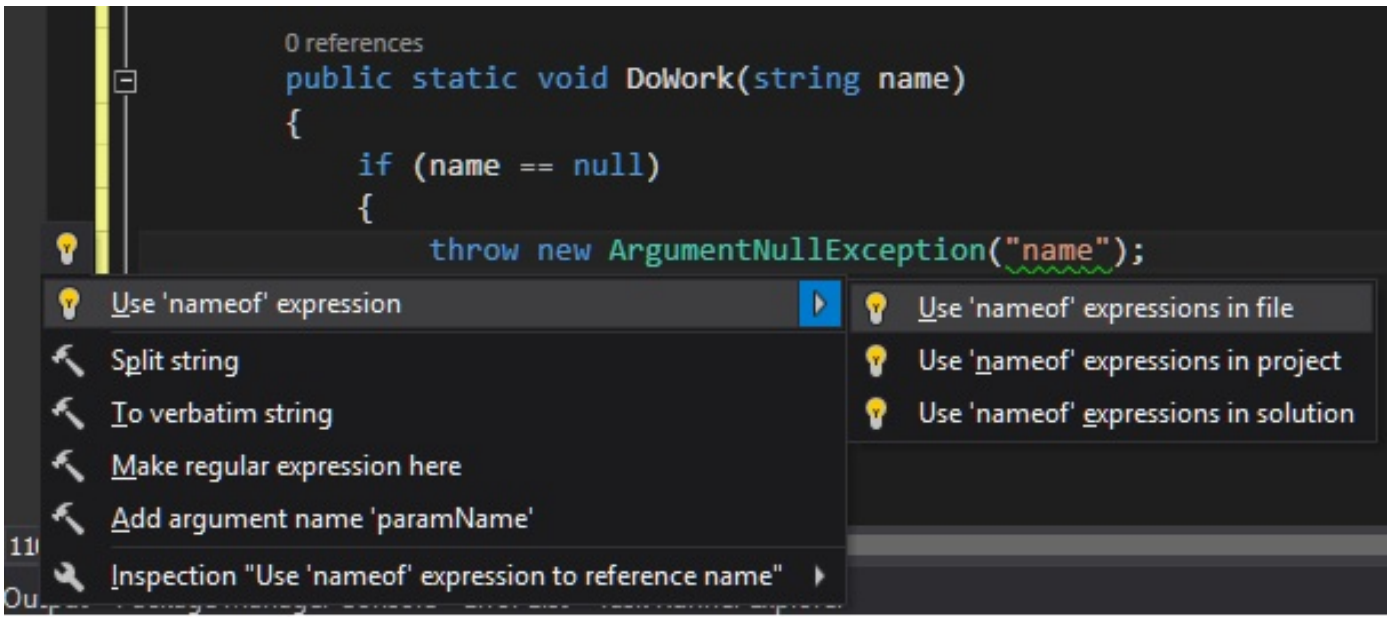
یکی دیگر از قابلیت‌های جذاب نسخه‌ی جدید سی‌شارپ، عملگر nameof است. هدف اصلی آن ارائه کدهایی با قابلیت Refactoring بهتر است؛ زیرا به جای نوشتن نام فیلدها و یا متدها در صورت نیاز به صورت hard-coded، می‌توانیم از این عملگر استفاده کنیم. به عنوان مثال در زمان صدور استثنایی از نوع ArgumentNullException باید نام آرگومان را به سازنده‌ی این کلاس پاس دهیم. متاسفانه یکی از مشکلاتی که با رشته‌ها در حالت کلی وجود دارد این است که امکان دیباگ در زمان کامپایل را از دست خواهیم داد و با تغییر هر المنت، تغییرات به صورت خودکار به رشته پاس داده شده، به سازنده‌ی کلاس ArgumentNullException اعمال نخواهد شد:

```
public static void DoWork(string name)
{
        if (name == null)
        {
            throw new ArgumentNullException("name");
        }
}
```

اما با استفاده از عملگر nameof، کد امن‌تری را خواهیم داشت؛ زیرا همیشه نام واقعی آرگومان به سازنده‌ی کلاس ArgumentNullException پاس داده می‌شود:

```
public static void DoWork(string name)
{
        if (name == null)
        {
            throw new ArgumentNullException(nameof(name));
        }
}
```

اگر ReSharper را نصب کرده باشید، به شما پیشنهاد می‌دهد که از nameof به جای یک رشته‌ی جادویی (magic string) استفاده نمائید:



یک مثال دیگر می‌تواند در زمان فراخوانی رخدادهای مربوط به OnPropertyChanged باشد. در اینجا باید نام خصوصیتی را که تغییر یافته است، به آن پاس دهیم:

```
public string Name
        {
            get { return _name; }
            set
            {
                _name = value;
                OnPropertyChanged("Name");
            }
        }
```

اما با کمک عملگر nameof می‌توانیم قسمت فراخوانی متد OnPropertyChanged را به اینصورت نیز بازنویسی کنیم:

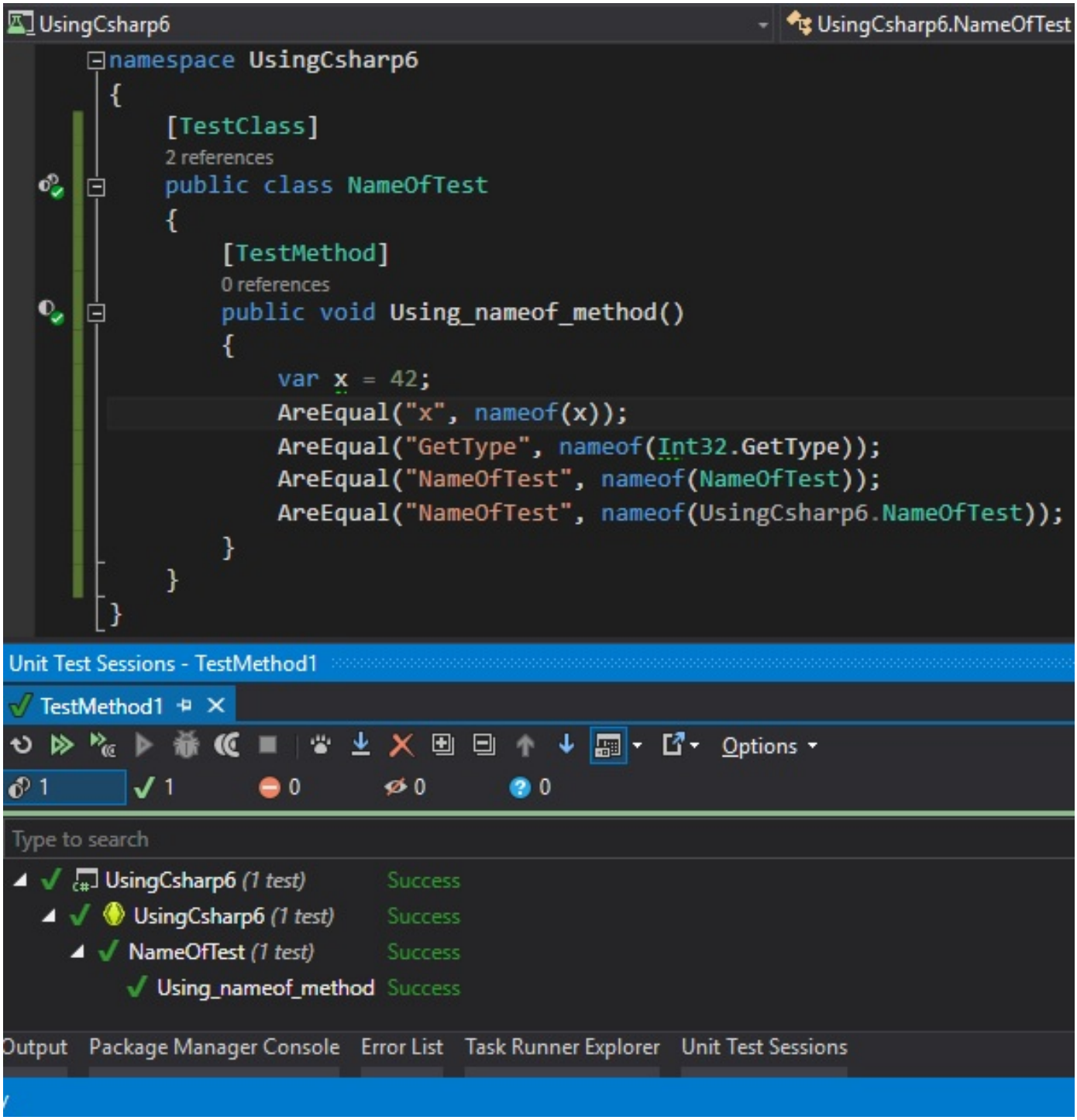
```
OnPropertyChanged(nameof(Name));
```

ممکن است عنوان کنید قبلاً در سی‌شارپ 5 هم می‌توانستیم از ویژگی [CallerMemberName] استفاده کنیم، پس دیگر نیازی به استفاده از عملگر nameof نخواهد بود. اما تفاوت کلیدی این است که CallerMemberName در زمان اجرا نام فیلد فراخوان را دریافت میکند (run time)، در حالیکه با استفاده از عملگر nameof می‌توانید در زمان کامپایل به نام فیلد دسترسی داشته باشید (compile time).

محدودیت‌های عملگر nameof این عملگر حالت‌هایی را که مشاهده می‌کنید، فعلاً پشتیبانی نخواهد کرد:

```
nameof(f()); // where f is a method - you could use nameof(f) instead
nameof(c._Age); // where c is a different class and _Age is private. Nameof can't break accessor rules.
nameof(List<>); // List<> isn't valid C# anyway, so this won't work
nameof(default(List<int>)); // default returns an instance, not a member
nameof(int); // int is a keyword, not a member- you could do nameof(Int32)
nameof(x[2]); // returns an instance using an indexer, so not a member
nameof("hello"); // a string isn't a member
nameof(1 + 2); // an int isn't a member
```

برای آزمایش عملگر nameof می‌توانیم یک تست را در حالت‌های زیر بنویسیم:



همانطور که مشاهده می‌کنید، همه‌ی حالت‌های فوق با موفقیت پاس شده‌اند.

عنوان: **C# 6 - Null-conditional operators**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۷/۱۲ ۱۴:۳۵
آدرس: www.dotnettips.info
گروه‌ها: C#, C# 6.0

برنامه نویس‌های سی‌شارپ پیشتر با null-coalescing operator یا ?? آشنا شده بودند. برای مثال

```
string data = null;
var result = data ?? "value";
```

در این حالت اگر data یا سمت چپ عملگر، نال باشد، مقدار value (سمت راست عملگر) بازگشت داده خواهد شد؛ که در حقیقت خلاصه شده‌ی چند سطر ذیل است:

```
if (data == null)
{
    data = "value";
}
var result = data;
```

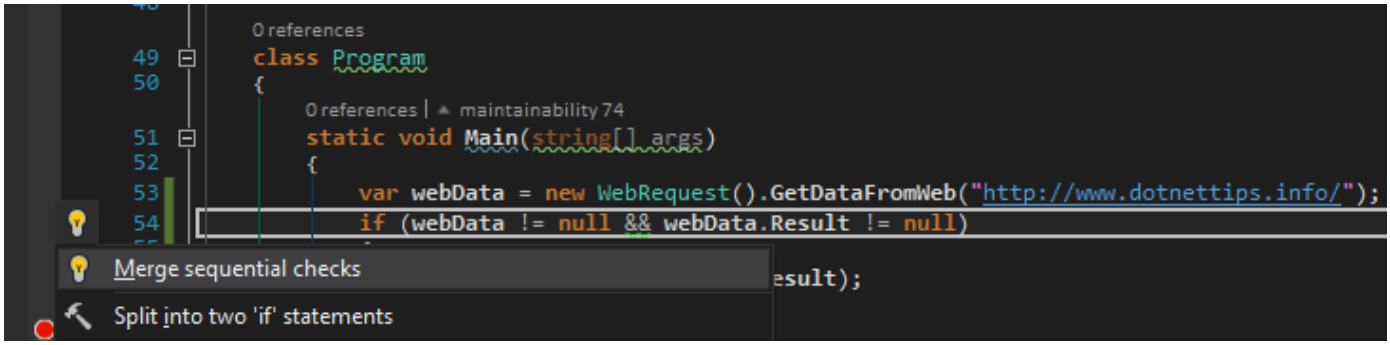
در سی شارپ 6، جهت تکمیل عملگرهای کار با مقادیر نال و بالا بردن productivity برنامه نویس‌ها، عملگر دیگری به نام Null-conditional operator و یا .? به این مجموعه اضافه شده‌است. در این حالت ابتدا مقدار سمت چپ عملگر بررسی خواهد شد. اگر مقدار آن مساوی نال بود، در همینجا کار خاتمه یافته و نال بازگشت داده می‌شود. در غیر اینصورت کار بررسی زنجیره‌ی جاری ادامه خواهد یافت.
برای مثال بسیاری از نتایج بازگشتی از متدها، چند سطحی هستند:

```
class Response
{
    public string Result { set; get; }
    public int Code { set; get; }
}


class WebRequest
{
    public Response GetDataFromWeb(string url)
    {
        // ...
        return new Response { Result = null };
    }
}
```

در اینجا روش مرسوم کار با کلاس درخواست اطلاعات از وب به صورت ذیل است:

```
var webData = new WebRequest().GetDataFromWeb("http://www.dotnettips.info/");
if (webData != null && webData.Result != null)
{
    Console.WriteLine(webData.Result);
}
```

چون می‌خواهیم به خاصیت Result دسترسی پیدا کنیم، نیاز است دو مرحله وضعیت خروجی متد و همچنین خاصیت Result آن‌را جهت مشخص سازی نال نبودن آن‌ها، بررسی کنیم و اگر برای مثال خاصیت Result نیز خود متشکل از یک کلاس دیگر بود که در آن برای مثال StatusCode نیز ذکر شده بود، این بررسی به سه سطح یا بیشتر نیز ادامه پیدا می‌کرد.
در این حالت اگر اشاره‌گر را به محل && انتقال دهیم، افزونه‌ی ReSharper پیشنهاد یکی کردن این بررسی‌ها را ارائه می‌دهد:



به این ترتیب تمام چند سطح بررسی نال، به یک عبارت بررسی .? دار، خلاصه خواهد شد:

```
if (webData?.Result != null)
{
    Console.WriteLine(webData.Result);
}
```

در اینجا ابتدا بررسی می‌شود که آیا webData نال است یا خیر؟ اگر نال بود همینجا کار خاتمه پیدا می‌کند و به بررسی Result نمی‌رسد. اگر نال نبود، ادامه‌ی زنجیره تا به انتها بررسی می‌شود.
البته باید دقت داشت که برای تمام سطوح باید از .? استفاده کرد (برای مثال response?.Results?.Status)؛ در غیر اینصورت همانند سابق در صورت استفاده‌ی از دات معمولی، به یک null reference exception می‌رسیم.

کار با متدها و Delegates

این عملگر جدید مقایسه‌ی با نال را بر روی متدها (علاوه بر خواص و فیلدها) نیز می‌توان بکار برد. برای مثال خلاصه شده‌ی فراخوانی ذیل:

```
if (x != null)
{
   x.Dispose();
}
```

با استفاده از Null Conditional Operator به این صورت است:

```
x?.Dispose();
```

و یا بکار گیری آن بر روی delegates (روش قدیمی):

```
var copy = OnMyEvent;
if (copy != null)
{
   copy(this, new EventArgs());
}
```

نیز با استفاده از متد Invoke به نحو ذیل قابل انجام است و نکته جالب یک سطر کد ذیل علاوه بر ساده شدن آن:

```
OnMyEvent?.Invoke(this, new EventArgs());
```

Thread-safe بودن آن نیز می‌باشد. زیرا در این حالت کامپایلر delegate را به یک متغیر موقتی کپی کرده و سپس فراخوانی‌ها را انجام می‌دهد. اگر انجام این کپی موقت صورت نمی‌گرفت، در حین فراخوانی آن از طریق چندین ترد مختلف، ممکن بود یکی از

مشترکین delegate از آن قطع اشتراک می‌کرد و در این حالت فراخوانی تردی دیگر در همان لحظه، سبب کرش برنامه می‌شد.

استفاده از Null Conditional Operator بر روی Value types

الف) مقایسه با نال

کد ذیل را درنظر بگیرید:

```
var code = webData?.Code;
```

در اینجا Code یک value type از نوع int است. در این حالت با بکارگیری Null Conditional Operator، خروجی این حاصل، از نوع Nullable<int> و یا int? درنظر گرفته خواهد شد و با توجه به اینکه عبارات 0<null و همچنین 0>null هر دو false هستند، مقایسه‌ی این خروجی با 0 بدون مشکل انجام می‌شود. برای مثال مقایسه‌ی ذیل از نظر کامپایلر یک عبارت معتبر است و بدون مشکل کامپایل می‌شود:

```
if (webData?.Code > 0)
{

}
```

ب) بازگشت مقدار پیش فرض دیگری بجای نال

اگر نیاز بود بجای null مقدار پیش فرض دیگری را بازگشت دهیم، می‌توان از null-coalescing operator سابق استفاده کرد:

```
int count = response?.Results?.Count ?? 0;
```

در این مثال خاصیت CountT در اصل از نوع int تعریف شده‌است؛ اما بکارگیری .? سبب Nullable شدن آن خواهد شد. بنابراین امکان بکارگیری عملگر ?? یا null-coalescing operator نیز بر روی این متغیر وجود دارد.

ج) دسترسی به مقدار Value یک متغیر nullable

نمونه‌ی دیگر آن قطعه کد ذیل است:

```
int? x = 10;
//var value = x?.Value; // invalid
Console.WriteLine(x?.ToString());
```

در اینجا برخلاف متغیر Code که از ابتدا nullable تعریف نشده‌است، متغیر x نال پذیر است. اما باید دقت داشت که با تعریف .? دیگر نیازی به استفاده از خاصیت Value این متغیر nullable نیست؛ زیرا .? سبب محاسبه و بازگشت خروجی آن می‌شود. بنابراین در این حالت، سطر دوم غیرمعتبر است (کامپایل نمی‌شود) و سطر سوم معتبر.

کار با indexer property و بررسی نال

اگر به عنوان بحث دقت کرده باشید، یک s جمع در انتهای **s** Null-conditional operator ذکر شده‌است. به این معنا که این عملگر مقایسه‌ی با نال، صرفا یک شکل و فرم .? را ندارد. مثال ذیل در حین کار با آرایه‌ها و لیست‌ها بسیار مشاهده می‌شود:

```
if (response != null && response.Results != null && response.Results.Addresses != null
  && response.Results.Addresses[0] != null && response.Results.Addresses[0].Zip == "63368")
{

}
```

در اینجا به علت بکارگیری indexer بر روی Addresses، دیگر نمی‌توان از عملگر .? که صرفا برای فیلدها، خواص، متدها و delegates طراحی شده‌است، استفاده کرد. به همین منظور، عملگر بررسی نال دیگری به شکل [...]? برای این بررسی طراحی

شده‌است:

```
if(response?.Results?.Addresses?[0]?.Zip == "63368")
{

}
```

به این ترتیب 5 سطح بررسی نال فوق، به یک عبارت کوتاه کاهش می‌یابد.

موارد استفاده‌ی ناصحیح از عملگرهای مقایسه‌ی با نال

خوب، عملگر .? کار مقایسه‌ی با نال را خصوصا در دسترسی‌های چند سطحی به خواص و متدها بسیار ساده می‌کند. اما آیا باید در همه جا از آن استفاده کرد؟ آیا باید از این پس کلا استفاده از دات را فراموش کرد و بجای آن از .? در همه جا استفاده کرد؟
مثال ذیل را درنظر بگیرید:

```
public void DoSomething(Customer customer)
{
    string address = customer?.Employees
                  ?.SingleOrDefault(x => x.IsAdmin)?.Address?.ToString();
    SendPackage(address);
}
```

در این مثال در تمام سطوح آن از .? بجای دات استفاده شده‌است و بدون مشکل کامپایل می‌شود. اما این نوع فراخوانی سبب خواهد شد تا یک سری از مشکلات موجود کاملا مخفی شوند؛ خصوصا اعتبارسنجی‌ها. برای مثال در این فراخوانی اگر مشتری نال باشد یا اگر کارمندانی را نداشته باشد، آدرسی بازگشت داده نمی‌شود. بنابراین حداقل دو سطح بررسی و اعتبارسنجی عدم وجود مشتری یا عدم وجود کارمندان آن در اینجا مخفی شده‌اند و دیگر مشخص نیست که علت بازگشت نال چه بوده‌است.
روش بهتر انجام اینکار، بررسی وضعیت customer و انتقال مابقی زنجیره‌ی LINQ به یک متد مجزای دیگر است:

```
public void DoSomething(Customer customer)
{
   Contract.Requires(customer != null);
   string address = customer.GetAdminAddress();
   SendPackage(address);
}
```

نظرات خوانندگان

نویسنده: امیر ح کریمی
تاریخ: ۱۳۹۴/۰۷/۱۹ ۱۴:۳۶

با سلام
برای چک کردن مقادیر نال پی در پی واقعا کاربردی است
البته موردی که ابتدای مطلب اومده اشکال کوچکی دارد :

```
string data = null;
var result = data ?? "value";
```

و

```
if (data == null)
{
    data = "value";
}
var result = data;
```

یکی نیستند چون در کد دوم مقدار data تغییر می‌کند(در صورتیکه برابر نال باشد).

عنوان: **C# 6 - Exception Filtering**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۷/۱۵ ۱۵:۲۵
آدرس: www.dotnettips.info
گروه‌ها: C#, C# 6.0

سی‌شارپ نیز مانند بسیاری از زبان‌های شیءگرای دیگر، امکان فیلتر کردن استثناءها را بر اساس نوع آن‌ها، دارا است. برای مثال:

```
try
{
    // some code to check  ...
}
catch (InvalidOperationException ex)
{
    // do your handling for invalid operation ...
}
catch (IOException ex)
{
    // do your handling for IO error ...
}
```

در اینجا می‌توان بر اساس نوع استثنای مدنظر، چندین catch را نوشت و مدیریت کرد. اما گاهی از اوقات شاید بهتر باشد بجای مدیریت کلی یک نوع از استثناءها، فقط نوعی خاص را صرفا بر اساس شرایطی مشخص، مدیریت کرد. این قابلیت، تحت عنوان Exception Filtering به C# 6 اضافه شده‌است و شکل کلی آن به صورت ذیل است:

```
catch (SomeException ex) when (someConditionIsMet)
{
    // Your handler logic
}
```

در این حالت ابتدا نوع استثناء بررسی می‌شود و سپس شرطی که در قسمت when ذکر شده‌است. اگر هر دو با هم برقرار بودند، آنگاه این استثنای خاص مدیریت خواهد شد؛ در غیر اینصورت، از مدیریت این نوع استثناء صرفنظر می‌گردد. این قابلیت، از ابتدای ارائه‌ی CLR وجود داشته‌است، اما C#6 تازه شروع به استفاده‌ی از آن کرده‌است (و VB.NET از چند نگارش قبل).

علاوه بر این در اینجا می‌توان چندین بدنه‌ی catch مجزا را به ازای یک نوع استثنای مشخص به همراه whenهای متفاوتی نیز تعریف کرد و از این لحاظ محدودیتی وجود ندارد. فقط در این حالت باید به تقدم و تاخرها دقت داشت. برای نمونه در مثال ذیل، ترکیب چندین شرط متفاوت را بر اساس یک نوع مشخص استثناء، مشاهده می‌کنید. در اینجا اگر برای نمونه شرط ذکر شده‌ی در قسمت when مربوط به catch اولی صادق باشد، همینجا کار خاتمه می‌یابد و سایر catchها بررسی نمی‌شوند:

```
catch (SomeDependencyException ex) when (condition1 && condition2)
{

}
catch (SomeDependencyException ex) when (condition1)
{

}
catch (SomeDependencyException ex)
{

}
```

مورد آخر، حالت catch all را دارد و در صورت شکست دو catch قبلی اجرا می‌شود. اما باید دقت داشت که اگر این catch all بدون شرط و بدون قسمت when را در ابتدا ذکر کنیم، دیگر کار به بررسی سایر catchهای این نوع استثنای خاص نخواهد رسید:

```
catch (SomeDependencyException ex)
{

}
catch (SomeDependencyException ex) when (condition1 && condition2)
```

```
{

}
catch (SomeDependencyException ex) when (condition1)
{

}
```

در مثال فوق هیچگاه دو catch تعریف شده‌ی پس از catch all اولی اجرا نمی‌شوند.

لاگ کردن استثناءها در C# 6 بدون مدیریت آن‌ها

به مثال ذیل دقت کنید:

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex) when (Log(ex, "An error occurred"))
{
    // this catch block will never be reached
}

...

static bool Log(Exception ex, string message, params object[] args)
{
   Debug.Print(message, args);
   return false;
}
```

در قسمت when می‌توان هر متدی که true یا false را برگرداند، فراخوانی کرد. در این مثال، متدی تعریف شده‌است که false بر می‌گرداند. یعنی این استثناء کلی از نوع Exception هرچند به ظاهر دارای قسمت when است و مدیریت شده‌است، اما چون خروجی متد Log قسمت when آن مساوی false است، مدیریت نخواهد شد. یعنی در اینجا می‌توان بدون مدیریت یک استثناء، اطلاعات کامل آن‌را لاگ کرد!

تفاوت C# 6 - Exception Filtering با if/else نوشتن در بدنه‌ی catch چیست؟

تا اینجا به این نتیجه رسیدیم که کدهای if/else دار داخل بدنه‌ی catch کدهای قدیمی را مانند کد ذیل:

```
try
{
    var request = WebRequest.Create("http://www.google.coom/");
    var response = request.GetResponse();
}
catch (WebException we)
{
    if (we.Status == WebExceptionStatus.NameResolutionFailure)
    {
        //handle DNS error
        return;
    }
    if (we.Status == WebExceptionStatus.ConnectFailure)
    {
        //handle connection error
        return;
    }

    throw;
}
```

می‌توان به شکل جدید C# 6 به همراه when نوشت و تبدیل کرد:

```
try
```

```
{
    var request = WebRequest.Create("http://www.google.coom/");
    var response = request.GetResponse();
}
catch (WebException we) when (we.Status == WebExceptionStatus.NameResolutionFailure)
{
    //Handle NameResolutionFailure Separately
}
catch (WebException we) when (we.Status == WebExceptionStatus.ConnectFailure)
{
    //Handle ConnectFailure Separately
}
```

اما باید دقت داشت که تفاوت مهم قطعه کد دوم، در مباحث Stack unwinding است. در مثال اولی که if/else داخل بدنه‌ی catch نوشته شده‌است، اطلاعات local محل فراخوانی متدی را که سبب بروز استثناء شده‌است، از دست خواهیم داد؛ اما در مثال دوم خیر.

به این معنا که exception filters سبب Stack unwinding نمی‌شوند. با هربار ورود به بدنه‌ی catch، اصطلاحا عملیات Stack unwinding صورت می‌گیرد. یعنی اطلاعات stack مربوط به متدهای پیش از فراخوانی متدی که سبب بروز استثناء شده‌است، از بین می‌روند. به این ترتیب تشخیص مقادیر متغیرهایی که سبب بروز این استثناء شده‌اند نیز میسر نخواهد بود و دیگر نمی‌توان با قطعیت عنوان کرد که چه مقادیری و چه اطلاعاتی سبب بروز این مشکل شده‌اند. اما در حالت exception filters در قسمت when آن هنوز وارد بدنه‌ی catch نشده‌ایم. در اینجا دسترسی کاملی به اطلاعات stack جاری و مقادیر متغیرهای محلی که سبب بروز این استثناء شده‌اند وجود دارد.

تفاوت stack با stack trace چیست؟ stack قطعه‌ای از حافظه‌است که اطلاعاتی در مورد نحوه‌ی فراخوانی متدها، آدرس بازگشتی آن‌ها، آرگومان و همچنین متغیرهای محلی آن‌ها را دارا است. اما stack trace تنها یک رشته‌است و بیانگر نام متدهایی است که هم اکنون بر روی stack قرار دارند. احتمالا پیشتر خوانده بودید که فراخوانی throw داخل بدنه‌ی catch سبب حفظ stack trace می‌شود و اگر throw ex صورت گیرد، این اطلاعات از دست می‌روند و بازنویسی می‌شوند. اما در C# 6 امکان حفظ کل اطلاعات stack به همراه exception filtering میسر شده‌است.

عنوان: **C# 6 - Expression-Bodied Members**
نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۴/۰۷/۱۸ ۱۶:۵۵
آدرس: www.dotnettips.info
گروه‌ها: C#, C# 6.0

در ادامه مطالب منتشر شده در رابطه با قابلیت‌های جدید سی‌شارپ 6، در این مطلب به بررسی یکی دیگر از این قابلیت‌ها، با نام Expression-Bodied Members خواهیم پرداخت. در واقع در سی‌شارپ 6، هدف، ساده‌سازی سینتکس و افزایش بهره‌وری برنامه‌نویس می‌باشد. در نسخه‌های قبلی سی‌شارپ برای یکسری از اعمال روتین می‌بایستی روالی‌هایی را مدام تکرار می‌کردیم؛ به عنوان مثال در تعریف پراپرتی‌های یک کلاس در حالت get-only باید هر بار توسط return مقداری را برگردانیم:

```
public class Person
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string FullName
   {
       get
       {
                return FirstName + " " + LastName;
       }
   }
}
```

نوشتن پراپرتی‌هایی همانند FullName منجر به نوشتن خطوط کد اضافه‌تری خواهد شد، هرچند می‌توان این حالت را با برداشتن خطوط اضافی بهبود بخشید:

```
public string FullName
{
       get { return FirstName + " " + LastName; }
}
```

اما در سی‌شارپ 6 میتوان آن را توسط expression body به یک خط کاهش داد!

استفاده از expression body برای پراپرتی‌های get-only (فقط خواندنی):

اگر در کلاس‌هایتان پراپرتی‌های get-only دارید، به راحتی می‌توانید بدنه‌ی پراپرتی را با استفاده از expression syntax خلاصه‌نویسی کنید. در واقع شما با استفاده از سینتکس lambda expression اقدام به نوشتن بدنه پراپرتی‌های موردنظرتان می‌کنید. یعنی به جای نوشتن کدی مانند:

```
{ get { return your expression; } }
```

به راحتی می‌توانید از سینتکس زیر استفاده نمائید:

```
=> your expression;
```

به عنوان مثال، میتوان پراپرتی FullName را در کلاس Person با کمک قابلیت expression body به صورت زیر بازنویسی کنیم:

```
public class Person
{
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public string FullName => FirstName + " " + LastName;
}
```

با کد فوق به راحتی توانستیم قسمت‌های اضافه‌ای را حذف کنیم. اکنون ممکن است بپرسید آیا این تغییر در performance برنامه

تاثیری دارد؟ خیر؛ زیرا سینتکس فوق دقیقاً همان کد ILی را تولید خواهد کرد که در حالت عادی تولید می‌شود. همچنین delegateی را تولید نخواهد کرد؛ بلکه تنها از سینتکس lambda expression برای خلاصه‌نویسی بدنه پراپرتی استفاده می‌کند. در حال حاضر برای حالت setter سینتکسی ارائه نشده است.

استفاده از expression body برای Indexerها:

همچنین از این قابلیت برای Indexerها نیز میتوان استفاده کرد، مثلاً به جای نوشتن کد زیر:

```
public string this[int number]
{
            get
            {
                if (number >= 0 && number < _values.Length)
                {
                    return _values[number];
                }
                return "Error";
            }
}
```

می‌توانیم کد فوق را به این صورت خلاصه‌نویسی کنیم:

```
public string this[int number] => (number >= 0 && number < _values.Length) ? _values[number] : "Error";
```

نکته: توجه داشته باشید که در هر دو حالت فوق تنها می‌توانیم برای get از expression body استفاده کنیم، هنوز سینتکسی برای حالت set ارائه نشده است.

استفاده از expression body برای متدها:

برای متدها نیز می‌توانیم از قابلیت عنوان شده استفاده نمائیم، به عنوان مثال اگر داخل کلاس Person متد زیر را داشته باشیم:

```
public override string ToString()
{
      return FirstName;
}
```

می‌توانیم آن را به صورت زیر بنویسیم:

```
public override string ToString() => FirstName;
```

همانطور که مشاهده می‌کنید به جای نوشتن curly braces یا {} از lambda arrow یا => استفاده کرده‌ایم. در اینجا عبارت سمت راست lambda arrow نمایانگر بدنه‌ی متد است. همچنین برای متدهای دارای پارامتر نیز به این صورت عمل می‌کنیم:

```
public int DoubleTheValue(int someValue) => someValue * 2;
```

یک عضو از کلاس که به صورت expression body نوشته شده باشد، expression bodied member نامیده می‌شود. این عضو از کلاس در ظاهر شبیه به عبارات لامبدای ناشناس (anonymous lambda expression) است. اما یک expression bodied member باید دارای نام، مقدار بازگشتی و بدنه متد باشد.
تقریباً تمامی access modifierها در این حالت قابلیت استفاده را دارند. تنها متدهای abstract نمی‌توانند استفاده شوند.

محدودیت‌های Expression Bodied Members

یکی از محدودیت‌های استفاده از expression body داشتن چندین خط دستور برای بدنه متدهایمان است. در اینحالت باید از روش سابق (statement body) استفاده نمائید.
یکی دیگر از محدودیت‌ها عدم امکان استفاده از if, else, switch است. به عنوان مثال نمی‌توان کد زیر را با داشتن if و else به صورت expression body نوشت:

```
public override string ToString()
{
	if (MiddleName != null)
	{
		return FirstName + " " + MiddleName + " " + LastName;
	}
	else
	{
		return FirstName + " " + LastName;
	}
}
```

برای حالت فوق به عنوان یک روش جایگزین می‌توان از conditional operator استفاده کرد:

```
public override string ToString() =>
                    (MiddleName != null)
                    ? FirstName + " " + MiddleName + " " + LastName
                    : FirstName + " " + LastName;
```

همچنین نمی‌توان از for, foreach, while, do در expression body استفاده کرد، به جای آن می‌توان از عبارت‌های LINQ برای بدنه تابع استفاده کرد. به عنوان مثال متد زیر:

```
public IEnumerable<int> SmallNumbers()
{
    for (int i = 0; i < 10; i++)
        yield return i;
}
```

را می‌توان در حالت expression body به این صورت نوشت:

```
public IEnumerable<int> SmallNumbers() => from n in Enumerable.Range(0, 10)
                                                                    select n;
```

و یا به این صورت:

```
public IEnumerable<int> SmallNumbers() => Enumerable.Range(0, 10).Select(n => n);
```

همانطور که عنوان شد، استفاده از expression body در قسمت پراپرتی‌ها تنها محدود به پراپرتی‌های get-only (فقط خواندنی) میباشد.

استفاده از این قابلیت برای متدهای سازنده
استفاده در رخدادها
استفاده در finalizers

نکته: اگر می‌خواهید expression bodied member شما هم initializer داشته باشد و همچنین یک read only auto property باشد، باید مقداری سینتکس آن را تغییر دهید. همانطور که می‌دانید auto propertyها نیازی به backing field ندارند؛ بلکه در زمان کامپایل به صورت خودکار تولید خواهند شد. در نتیجه برای مقداردهی اولیه به backing fieldها می‌توانیم درون سازنده کلاس آنها را initialize کنیم:

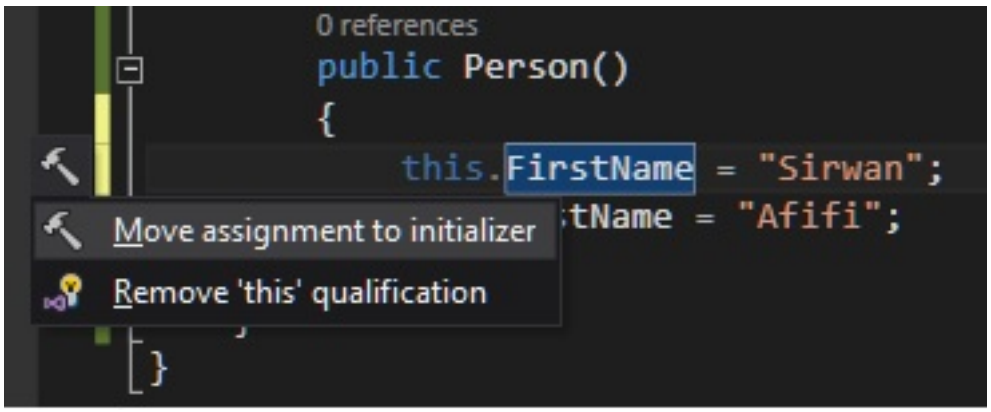
```
public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public Person()
        {
            this.FirstName = "Sirwan";
            this.LastName = "Afifi";
        }
    }
```

برای نوشتن پراپرتی‌های فوق به صورت expression body می‌توانیم به این صورت عمل کنیم:

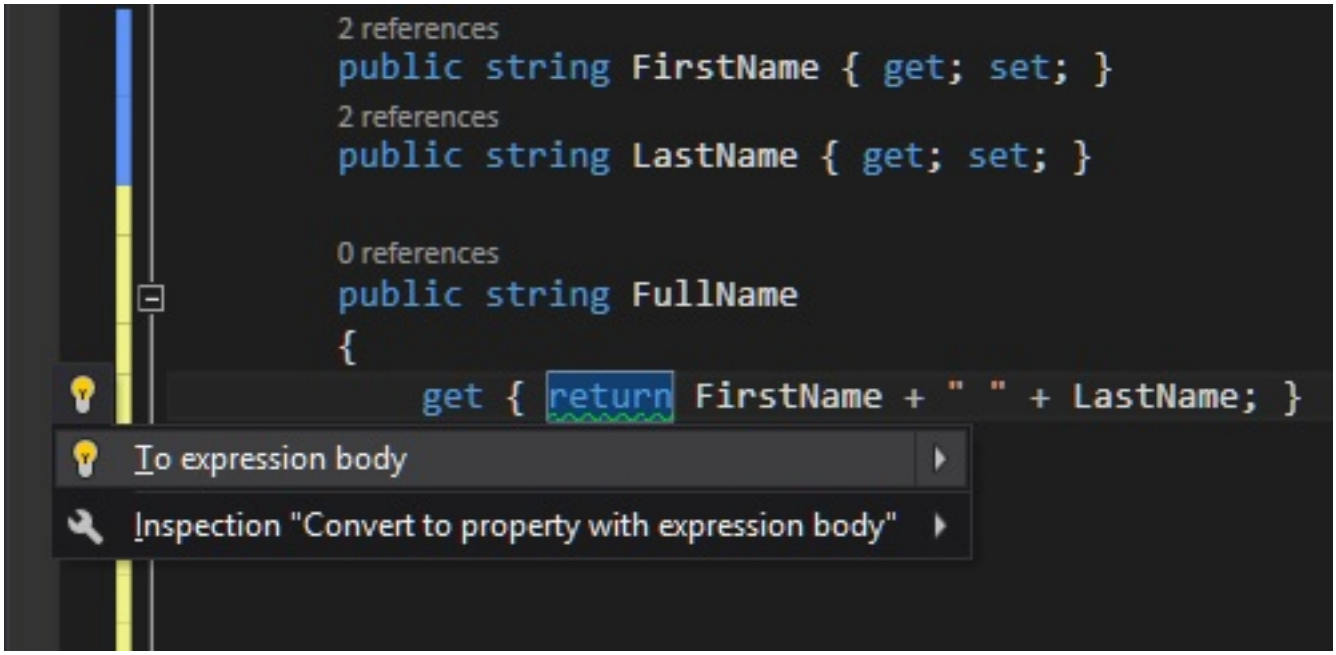
```
public string FirstName { get; set; } = "Sirwan";
public string LastName { get; set; } = "Afifi";
```

اگر ReSharper را نصب کرده باشید، به شما پیشنهاد می‌دهد که از expression body استفاده نمائید: :

برای حالت فوق:



برای پراپرتی‌ها:


2 references
public string FirstName { get; set; }
2 references
public string LastName { get; set; }
0 references
public string FullName
{
get { return FirstName + " " + LastName; }
To expression body
Inspection "Convert to property with expression body"

عنوان: **C# 6 - Index Initializers**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۷/۲۱ ۱۰:۳۰
آدرس: www.dotnettips.info
گروه‌ها: C#, C# 6.0

زمان زیادی از ارائه‌ی امکان Collection Initializer برای ایجاد یک متغیر از نوع Collection می‌گذرد؛ برای نمونه به مثال زیر توجه کنید:

```
enum USState {...}
var AreaCodeUSState = new Dictionary<string, USState>
    {
        {"408", USState.California},
        {"701", USState.NorthDakota},
        ...
    };
```

در پشت صحنه، کامپایلر، Collection Initializer را می‌گیرد، با استفاده از یک *Dictionary<TKey, TValue>* و با فراخوانی متد Add آن بر روی لیست Collection Initializer شروع به درج آن در دیکشنری ساخته شده می‌کند. Collection Initializer فقط بر روی کلاس هایی که در آن‌ها *IEnumerable* پیاده سازی شده باشد امکان پذیر است چرا که کامپایلر کار اضافه کردن مقادیر اولیه را به IEnumerable.Add() می‌سپارد.

اکنون در C# 6.0 ما می‌توانیم از Index Initializer استفاده کنیم:

```
enum USState {...}
var AreaCodeUSState = new Dictionary<string, USState>
    {
        ["408"] = USState.California,
        ["701"] = USState.NorthDakota,
        ...
    };
```

اولین تفاوتی که این دو روش با هم دارند این است که در حالت استفاده‌ی از Index Initializer پس از کامپایل، IEnumerable.Add() فراخوانی نمی‌شود. این تفاوت بسیار مهم است و کار اضافه کردن مقادیر اولیه را با استفاده از کلید (Key) ویژه انجام می‌دهد.
شبه کد مثال بالا به صورت زیر می‌شود:

Collection Initializer

```
create a Dictionary<string, USState>
add to new Dictionary the following items:
     "408", USState.California
     "701", USState.NorthDakota
```

Index Initializer

```
create a Dictionary<string, USState> then
using AreaCodeUSState's default Indexed property
    set the Value of Key "408" to USState.California
    set the Value of Key "701" to USState.NorthDakota
```

حال به مثال زیر توجه کنید:

Collection Initializer

```
enum USState {...}
var AreaCodeUSState = new Dictionary<string, USState>
```

```
        {
            { "408", USState.Confusion},
            { "701", USState.NorthDakota },
            { "408", USState.California},
            ...
        };
Console.WriteLine( AreaCodeUSState.Where(x => x.Key == "408").FirstOrDefault().Value );
```

Index Initializer

```
enum USState {...}
var AreaCodeUSState = new Dictionary<string, USState>
    {
        ["408"] = USState.Confusion,
        ["701"] = USState.NorthDakota,
        ["408"] = USState.California,
        ...
    };
Console.WriteLine( AreaCodeUSState2.Where(x => x.Key == "408").FirstOrDefault().Value );  // output =
California
```

هر دو کد بالا با موفقیت کامپایل و اجرا می‌شود، اما در زمان اجرای Collection Initializer هنگامیکه می‌خواهد مقدار دوم "408" را اضافه کند با استثناء *ArgumentException* متوقف می‌شود چرا که کلید "408" از قبل وجود دارد.
اما در زمان اجرا، Index Initializer به صورت کامل و بدون خطا این کار را انجام می‌دهد و در کلید "408" مقدار USState.Confusion قرار می‌گیرد. سپس "701" مقدار USState.NorthDakota و بعد از استفاده‌ی مجدد از کلید "408" مقدار USState.California جایگزین مقدار قبلی می‌شود.

```
var fibonaccis = new List<int>
    {
        [0] = 1,
        [1] = 2,
        [3] = 5,
        [5] = 13
    }
```

این کد هم معتبر است و هم کامپایل می‌شود. البته معتبر است، ولی صحیح نیست. List<T> اجازه‌ی تخصیص اندیسی فراتر از اندازه‌ی فعلی را نمی‌دهد.
تلاش برای تخصیص مقدار 1 با کلید 0 به List<int>، سبب بروز استثناء *ArguementOutOfRangeException* می شود. وقتی List<T>.Add(item) فراخوانی می‌شود اندازه‌ی لیست یک واحد افزایش می‌یابد. بنابراین باید دقت داشت که Index Initializer از ()Add. استفاده نمی‌کند؛ در عوض با استفاده از خصوصیت اندیس پیش فرض، مقداری را برای یک کلید تعیین می‌کند.
برای چنین حالتی بهتر است از همان روش قدیمی Collection Initializer استفاده کنیم:

```
var fibonaccis = new List<int>()
    {
        1,
        3,
        5,
        13
    };
```

عنوان: **ویژگی Static Using Statements در سی شارپ 6**
نویسنده: مانی مهدوی
تاریخ: ۱۳۹۴/۰۷/۲۱ ۲۰:۵
آدرس: www.dotnettips.info
گروه‌ها: C#, C# 6.0

مروری بر کاربردهای مختلف دستور Using تا پیش از ارائه‌ی سی شارپ 6

-1 اضافه کردن فضاهای نام مختلف، برای سهولت دسترسی به اعضای آن:

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

-2 تعریف نام مستعار (alias name) برای نوع داده‌ها و فضای نام‌ها

```
using BLL = DotNetTipsBLLLayer;//نام مستعار برای فضای نام
using EmployeeDomain = DotNetTipsBLLLayer.Employee;//نام مستعار برای یک نوع داده
```

-3 تعریف یک بازه و مشخص کردن زمان تخریب یک شیء و آزاد سازی حافظه‌ی تخصیص داده شده:

```
using (var sqlConnection = new SqlConnection())
            {
                //کد
            }
```

در سی شارپ 6 ، **Static** Using Statements برای بهبود کدنویسی و تمیزتر نوشتن کدها ارائه شده‌است.
در ابتدا نحوه‌ی عملکرد اعضای static را مرور می‌کنیم. متغیرها و متدهایی که با کلمه‌ی کلیدی static معرفی می‌شوند، اعلام می‌کنند که برای استفاده‌ی از آنها به نمونه سازی کلاس آن‌ها احتیاجی نیست و برای استفاده‌ی از آنها کافی است نام کلاس را تایپ کرده (بدون نوشتن new) و متد و یا خصوصیت مورد نظر را فراخوانی کنیم.
با معرفی ویژگی جدید Static Using Statement نوشتن نام کلاس برای فراخوانی اعضای استاتیک نیز حذف می‌شود.
اتفاق خوبی است اگر بتوان اعضای استاتیک را همچون Data Typeهای موجود در سی شارپ استفاده کرد. مثلا بتوان به جای Console.WrriteLine() نوشت WriteLine() **نحوه استفاده از این ویژگی:** در ابتدای فایل و بخش معرفی کتابخانه‌ها بدین شکل عمل می‌کنیم using **static** namespace. **className** .
در بخش className، نام کلاس استاتیک مورد نظر خود را می‌نویسیم .
مثال :

```
using static System.Console;
using static System.Math;

namespace dotnettipsUsingStatic
{
    class Program
    {
        static void Main(string[] args)
        {

            Write(" *** Cal Area *** ");
            int r = int.Parse(ReadLine());
            var result = Pow(r, 2) * PI;
            Write($"Area  is : {result}");
            ReadKey();
       }
    }
}
```

همان طور که در کدهای فوق می‌بینید، کلاس‌های Console و Math، در ابتدای فایل با استفاده از ویژگی جدید سی شارپ 6 معرفی شده‌اند و در بدنه برنامه تنها با فراخوانی نام متدها و خصوصیت‌ها از آنها استفاده کرده ایم.

استفاده از ویژگی using static و Enum:

فرض کنید می‌خواهیم یک نوع داده‌ی شمارشی را برای نمایش جنسیت تعریف کنیم:

```
enum Gender
    {
        Male,
        Female
    }
```

تا قبل از سی شارپ 6 برای استفاده‌ی از نوع داده شمارشی بدین شکل عمل می‌کردیم:

```
var gender = Gender.Male;
```

و اکنون بازنویسی استفاده‌ی از Enum به کمک ویژگی جدید static using statement :

در قسمت معرفی فضاهای نام بدین شکل عمل می‌کنیم:

```
using static dotnettipsUsingStatic.Gender;
```

و در برنامه کافیست مستقیما نام اعضای Enum را ذکر کنیم .

```
var gender = Male;//تخصیص نوع داده شمارشی
WriteLine($"Employee Gender is : {Male}");//استفاده مستقیم از نوع داده شمارشی
```

استفاده از ویژگی using static و متدهای الحاقی :

تا قبل از ارائه سی شارپ 6 اگر نیاز به استفاده‌ی از یک متد الحاقی خاص همچون where در فضای نام System.Linq.Enumeable داشتیم می‌بایستی فضای نام System.Linq را به طور کامل اضافه می‌کردیم و راهی برای اضافه کردن یک فضای نام خاص درون فضای نام بزرگتر وجود نداشت.

اما با قابلیت جدید اضافه شده می‌توانیم بخشی از یک فضای نام را اضافه کنیم:

```
using static System.Linq.Enumerable;
```

متدهای استاتیک و متدهای الحاقی در زمان استفاده از ویژگی using static:

فرض کنید کلاس static ای بنام MyStaticClass داشته باشیم که متد Print1 و Print2 در آن تعریف شده باشند:

```
public static class MyStaticClass
    {
        public static void Print1(string parameter)
        {
            WriteLine(parameter);
        }
        public static void  Print2(this string parameter)
        {
            WriteLine(parameter);
        }

    }
```

برای استفاده از متدهای تعریف شده به شکل زیر عمل می‌کنیم :

```
فراخوانی تابع استاتیک//
Print1("Print 1");//روش اول
MyStaticClass.Print1("Prtint 1");//روش دوم
فراخوانی متد الحاقی استاتیک//
MyStaticClass.Print2("Print 2");
"print 2".Print2();
```

ویژگی‌های جدید ارائه شده در سی شارپ 6 برای افزایش خوانایی برنامه‌ها و تمیزتر شدن کدها اضافه شده‌اند. در مورد ویژگی‌های ارائه شده در مقاله‌ی جاری این نکته مهم است که گاهی قید کردن نام کلاس‌ها خود سبب افزایش خوانایی کدها می‌شود .