عنوان: **Roslyn #6**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۳۱ ۱۶:۳۵
آدرس: www.dotnettips.info
گروه‌ها: Roslyn

معرفی Analyzers

پیشنیاز این بحث نصب مواردی است که در مطلب « شروع به کار با Roslyn » در قسمت دوم عنوان شدند:
الف) نصب SDK ویژوال استودیوی 2015
ب) نصب قالب‌های ایجاد پروژه‌های مخصوص Roslyn

البته این قالب‌ها چیزی بیشتر از ایجاد یک پروژه‌ی کلاس Library جدید و افزودن ارجاعاتی به بسته‌ی نیوگت Microsoft.CodeAnalysis، نیستند. اما درکل زمان ایجاد و تنظیم این نوع پروژه‌ها را خیلی کاهش می‌دهند و همچنین یک پروژه‌ی تست را ایجاد کرده و تولید بسته‌ی نیوگت و فایل VSIX را نیز بسیار ساده می‌کنند.

هدف از تولید Analyzers

بسیاری از مجموعه‌ها و شرکت‌ها، یک سری قوانین و اصول خاصی را برای کدنویسی وضع می‌کنند تا به کدهایی با قابلیت خوانایی بهتر و نگهداری بیشتر برسند. با استفاده از Roslyn و آنالیز کننده‌های آن می‌توان این قوانین را پیاده سازی کرد و خطاها و اخطارهایی را به برنامه نویس‌ها جهت رفع اشکالات موجود، نمایش داده و گوشزد کرد. بنابراین هدف از آنالیز کننده‌های Roslyn، سهولت تولید ابزارهایی است که بتوانند برنامه نویس‌ها را ملزم به رعایت استانداردهای کدنویسی کنند.
همچنین معلم‌ها نیز می‌توانند از این امکانات جهت ارائه‌ی نکات ویژه‌ای به تازه‌کاران کمک بگیرند. برای مثال اگر این قسمت از کد اینگونه باشد، بهتر است؛ مثلا بهتر است فیلدهای سطح کلاس، خصوصی تعریف شوند و امکان دسترسی به آن‌ها صرفا از طریق متدهایی که قرار است با آن‌ها کار کنند صورت گیرد.
این آنالیز کنندها به صورت پویا در حین تایپ کدها در ویژوال استودیو فعال می‌شوند و یا حتی به صورت خودکار در طی پروسه‌ی Build پروژه نیز می‌توانند ظاهر شده و خطاها و اخطارهایی را گزارش کنند.

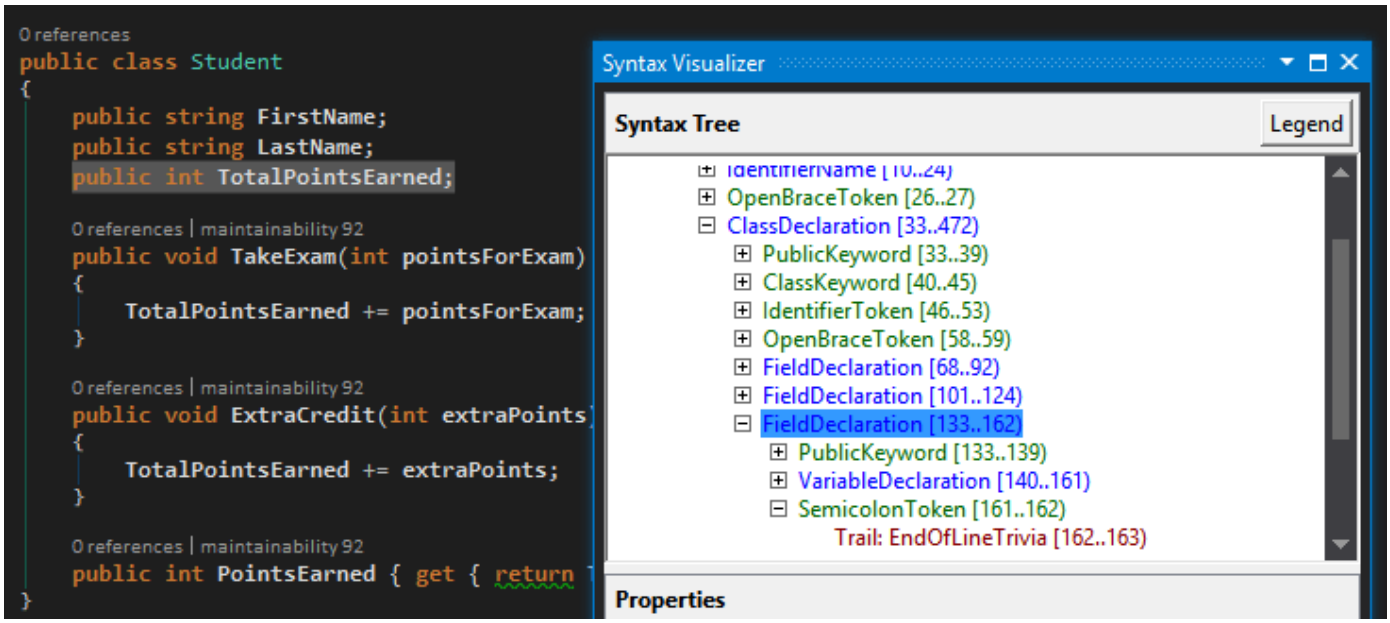
بررسی مثال معتبری که می‌تواند بهتر باشد

در اینجا یک کلاس نمونه را مشاهده می‌کنید که در آن فیلدهای کلاس به صورت public تعریف شده‌اند.

```
public class Student
    {
        public string FirstName;
        public string LastName;
        public int TotalPointsEarned;

        public void TakeExam(int pointsForExam)
        {
            TotalPointsEarned += pointsForExam;
        }

        public void ExtraCredit(int extraPoints)
        {
            TotalPointsEarned += extraPoints;
        }


        public int PointsEarned { get { return TotalPointsEarned; } }
    }
```

هرچند این کلاس از دید کامپایلر بدون مشکل است و کامپایل می‌شود، اما از لحاظ اصول کپسوله سازی اطلاعات دارای مشکل است و نباید جمع امتیازات کسب شده‌ی یک دانش آموز به صورت مستقیم و بدون مراجعه‌ی به متدهای معرفی شده، از طریق فیلدهای عمومی آن قابل تغییر باشد.

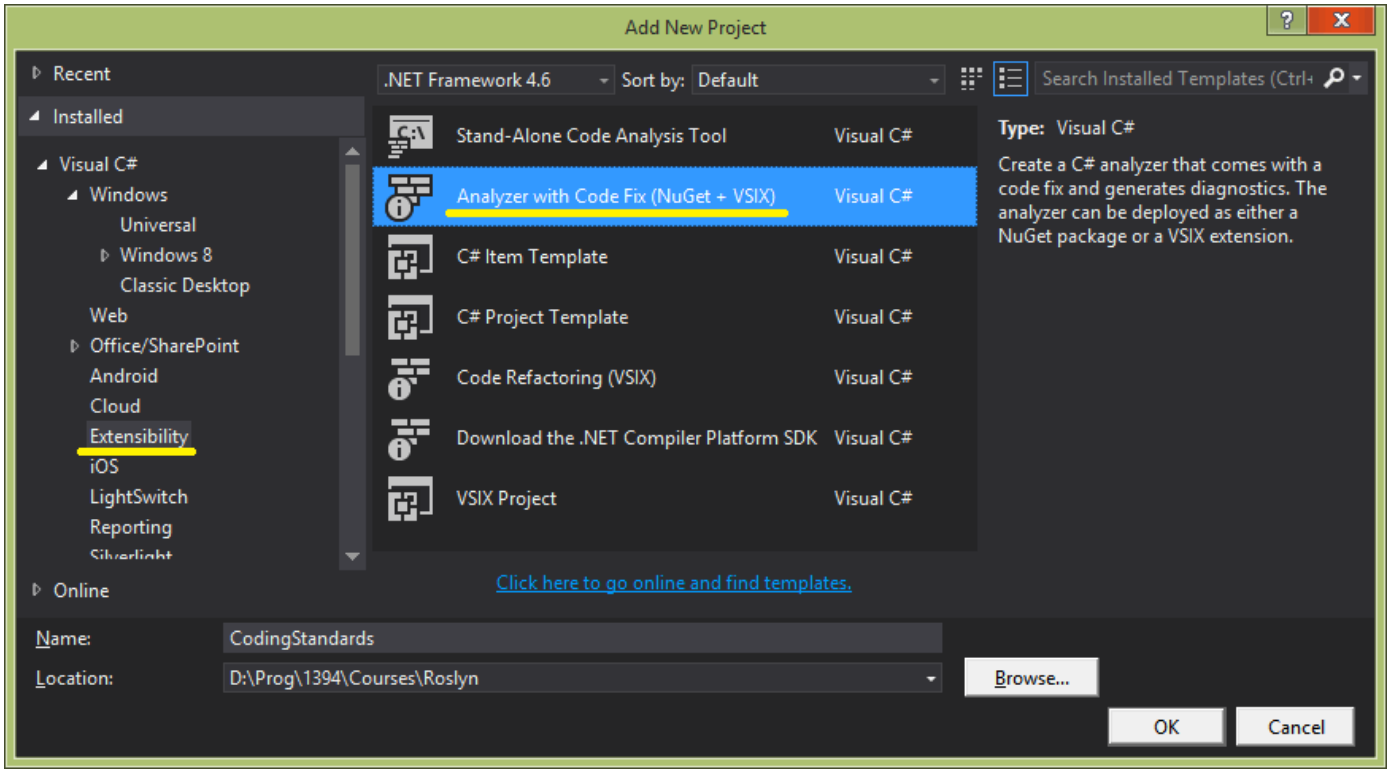
بنابراین در ادامه هدف ما این است که یک Roslyn Analyzer جدید را طراحی کنیم تا از طریق آن هشدارهایی را جهت تبدیل فیلدهای عمومی به خصوصی، به برنامه نویس نمایش دهیم.



با اجرای افزونه‌ی Syntax visualizer از مسیر View->Other windows->، تصویر فوق نمایان خواهد شد. بنابراین در اینجا نیاز است FieldDeclarationها را یافته و سپس tokenهای آن‌ها را بررسی کنیم و مشخص کنیم که آیا نوع یا Kind آن‌ها public است (PublicKeyword) یا خیر؟ اگر بلی، آن مورد را به صورت یک Diagnostic جدید گزارش می‌دهیم.

ایجاد اولین Roslyn Analyzer

پس از نصب پیشنیازهای بحث، به شاخه‌ی قالب‌های extensibility در ویژوال استودیو مراجعه کرده و یک پروژه‌ی جدید از نوع Analyzer with code fix را آغاز کنید.



قالب Stand-alone code analysis tool آن دقیقا همان برنامه‌های کنسول بحث شده‌ی در قسمت‌های قبل است که تنها ارجاعی را به بسته‌ی نیوگت Microsoft.CodeAnalysis به صورت خودکار دارد.

قالب پروژه‌ی Analyzer with code fix علاوه بر ایجاد پروژه‌های Test و VSIX جهت بسته بندی آنالایزر تولید شده، دارای دو فایل DiagnosticAnalyzer.cs و CodeFixProvider.cs پیش فرض نیز هست. این دو فایل قالب‌هایی را جهت شروع به کار تهیه‌ی آنالیز کننده‌های مبتنی بر Roslyn ارائه می‌دهند. کار DiagnosticAnalyzer آنالیز کد و ارائه‌ی خطاهایی جهت نمایش به ویژوال استودیو است و CodeFixProvider این امکان را مهیا می‌کند که این خطای جدید عنوان شده‌ی توسط آنالایزر، چگونه باید برطرف شود و راه‌کار بازنویسی Syntax tree آن‌را ارائه می‌دهد.

همین پروژه‌ی پیش فرض ایجاد شده نیز قابل اجرا است. اگر بر روی F5 کلیک کنید، یک کپی جدید و محصور شده‌ی ویژوال استودیو را باز می‌کند که در آن افزونه‌ی در حال تولید به صورت پیش فرض و محدود نصب شده‌است. اکنون اگر پروژه‌ی جدیدی را جهت آزمایش، در این وهله‌ی محصور شده‌ی ویژوال استودیو باز کنیم، قابلیت اجرای خودکار آنالایزر در حال توسعه را فراهم می‌کند. به این ترتیب کار تست و دیباگ آنالایزرها با سهولت بیشتری قابل انجام است.

این پروژه‌ی پیش فرض، کار تبدیل نام فضاهای نام را به upper case، به صورت خودکار انجام می‌دهد (که البته بی‌معنا است و صرفا جهت نمایش و ارائه‌ی قالب‌های شروع به کار مفید است).

نکته‌ی دیگر آن، تعریف تمام رشته‌های مورد نیاز آنالایزر در یک فایل resource به نام Resources.resx است که در جهت بومی سازی پیام‌های خطای آن می‌تواند بسیار مفید باشد.

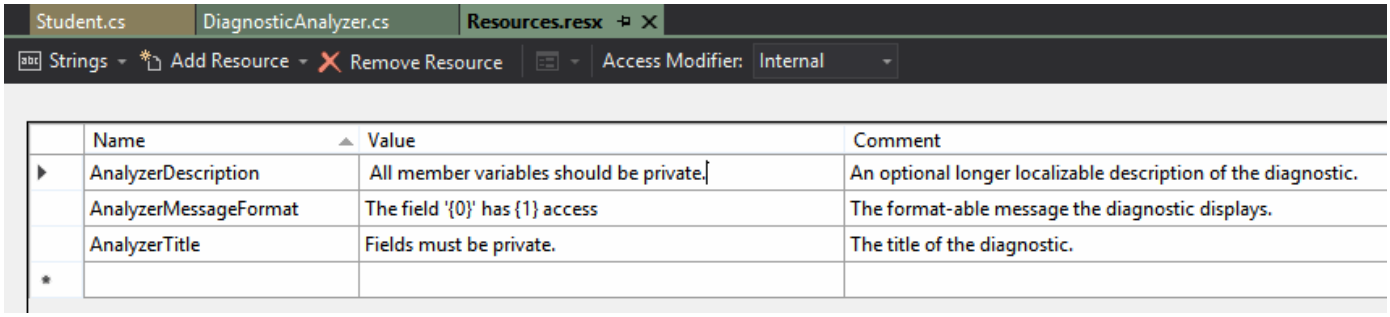
در ادامه کدهای فایل DiagnosticAnalyzer.cs را به صورت ذیل تغییر دهید:

```
using System.Collections.Immutable;
using System.Linq;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Diagnostics;

namespace CodingStandards
{
    [DiagnosticAnalyzer(LanguageNames.CSharp)]
    public class CodingStandardsAnalyzer : DiagnosticAnalyzer
    {
        public const string DiagnosticId = "CodingStandards";
```

```
        // You can change these strings in the Resources.resx file. If you do not want your analyzer to
be localize-able, you can use regular strings for Title and MessageFormat.
        internal static readonly LocalizableString Title = new
LocalizableResourceString(nameof(Resources.AnalyzerTitle), Resources.ResourceManager,
typeof(Resources));
        internal static readonly LocalizableString MessageFormat = new
LocalizableResourceString(nameof(Resources.AnalyzerMessageFormat), Resources.ResourceManager,
typeof(Resources));
        internal static readonly LocalizableString Description = new
LocalizableResourceString(nameof(Resources.AnalyzerDescription), Resources.ResourceManager,
typeof(Resources));
        internal const string Category = "Naming";

        internal static DiagnosticDescriptor Rule =
            new DiagnosticDescriptor(
                DiagnosticId,
                Title,
                MessageFormat,
                Category,
                DiagnosticSeverity.Error,
                isEnabledByDefault: true,
                description: Description);

        public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics
        {
            get { return ImmutableArray.Create(Rule); }
        }

        public override void Initialize(AnalysisContext context)
        {
            // TODO: Consider registering other actions that act on syntax instead of or in addition to
symbols
            context.RegisterSyntaxNodeAction(analyzeFieldDeclaration, SyntaxKind.FieldDeclaration);
        }

        static void analyzeFieldDeclaration(SyntaxNodeAnalysisContext context)
        {
            var fieldDeclaration = context.Node as FieldDeclarationSyntax;
            if (fieldDeclaration == null) return;
            var accessToken = fieldDeclaration
                                .ChildTokens()
                                .SingleOrDefault(token => token.Kind() == SyntaxKind.PublicKeyword);

            // Note: Not finding protected or internal
            if (accessToken.Kind() != SyntaxKind.None)
            {
                // Find the name of the field:
                var name = fieldDeclaration.DescendantTokens()
                               .SingleOrDefault(token =>
token.IsKind(SyntaxKind.IdentifierToken)).Value;
                var diagnostic = Diagnostic.Create(Rule, fieldDeclaration.GetLocation(), name,
accessToken.Value);
                context.ReportDiagnostic(diagnostic);
            }
        }
    }
}
```

توضیحات:

اولین کاری که در این کلاس انجام شده، خواندن سه رشته‌ی AnalyzerDescription (توضیحی در مورد آنالایزر)، AnalyzerMessageFormat (پیامی که به کاربر نمایش داده می‌شود) و AnalyzerTitle (عنوان پیام) از فایل Resources.resx است. این فایل را گشوده و محتوای آن‌را مطابق تنظیمات ذیل تغییر دهید:



سپس کار به متد Initialize می‌رسد. در اینجا برخلاف مثال‌های قسمت‌های قبل، context مورد نیاز، توسط پارامترهای override شده‌ی کلاس پایه DiagnosticAnalyzer فراهم می‌شوند. برای مثال در متد Initialize، این فرصت را خواهیم داشت تا به ویژوال استودیو اعلام کنیم، قصد آنالیز فیلدها یا FieldDeclaration را داریم. پارامتر اول متد RegisterSyntaxNodeAction یک delegate یا Action است. این Action کار فراهم آوردن context کاری را برعهده دارد که نحوه‌ی استفاده‌ی از آن‌را در متد analyzeFieldDeclaration می‌توانید ملاحظه کنید.

سپس در اینجا نوع نود در حال آنالیز (همان نودی که کاربر در ویژوال استودیو انتخاب کرده‌است یا در حال کار با آن است)، به نوع تعریف فیلد تبدیل می‌شود. سپس توکن‌های آن استخراج شده و بررسی می‌شود که آیا یکی از این توکن‌ها کلمه‌ی کلیدی public هست یا خیر؟ اگر این فیلد عمومی تعریف شده بود، نام آن‌را یافته و به عنوان یک Diagnostic جدید بازگشت و گزارش می‌دهیم.

ایجاد اولین Code fixer

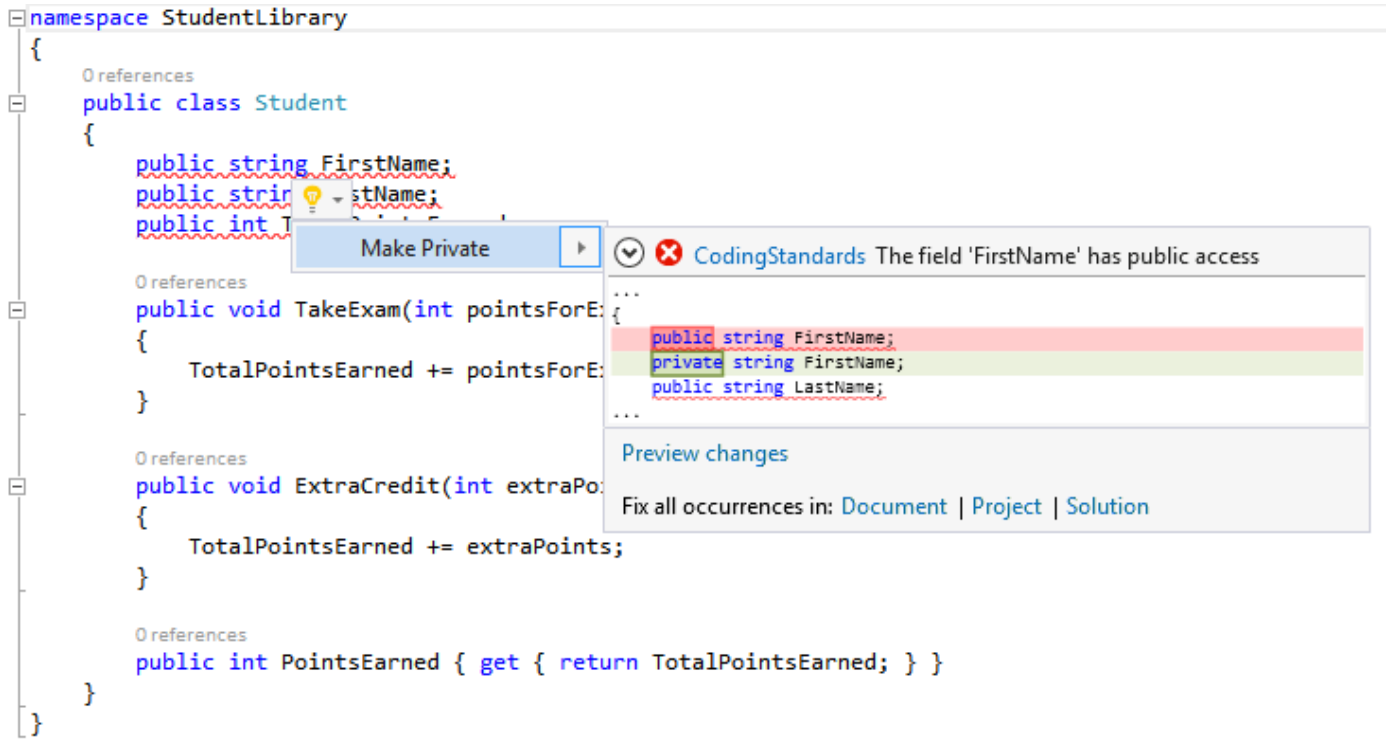
در ادامه فایل CodeFixProvider.cs پیش فرض را گشوده و تغییرات ذیل را به آن اعمال کنید. در اینجا مهم‌ترین تغییر صورت گرفته نسبت به قالب پیش فرض، اضافه شدن متد makePrivateDeclarationAsync بجای متد MakeUppercaseAsync از پیش موجود آن است:

```
using System.Collections.Immutable;
using System.Composition;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CodeFixes;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace CodingStandards
{
    [ExportCodeFixProvider(LanguageNames.CSharp, Name = nameof(CodingStandardsCodeFixProvider)),
Shared]
    public class CodingStandardsCodeFixProvider : CodeFixProvider
    {
        public sealed override ImmutableArray<string> FixableDiagnosticIds
        {
            get { return ImmutableArray.Create(CodingStandardsAnalyzer.DiagnosticId); }
        }

        public sealed override FixAllProvider GetFixAllProvider()
        {
            return WellKnownFixAllProviders.BatchFixer;
        }

        public sealed override async Task RegisterCodeFixesAsync(CodeFixContext context)
        {
            var root = await
context.Document.GetSyntaxRootAsync(context.CancellationToken).ConfigureAwait(false);

            // TODO: Replace the following code with your own analysis, generating a CodeAction for
each fix to suggest
            var diagnostic = context.Diagnostics.First();
            var diagnosticSpan = diagnostic.Location.SourceSpan;
```

```
            // Find the type declaration identified by the diagnostic.
            var declaration = root.FindToken(diagnosticSpan.Start)
                                  .Parent.AncestorsAndSelf().OfType<FieldDeclarationSyntax>()
                                  .First();

            // Register a code action that will invoke the fix.
            context.RegisterCodeFix(
                CodeAction.Create("Make Private",
                c => makePrivateDeclarationAsync(context.Document, declaration, c)),
                diagnostic);
        }

        async Task<Document> makePrivateDeclarationAsync(Document document, FieldDeclarationSyntax
declaration, CancellationToken c)
        {
            var accessToken = declaration.ChildTokens()
                .SingleOrDefault(token => token.Kind() == SyntaxKind.PublicKeyword);

            var privateAccessToken = SyntaxFactory.Token(SyntaxKind.PrivateKeyword);

            var root = await document.GetSyntaxRootAsync(c);
            var newRoot = root.ReplaceToken(accessToken, privateAccessToken);

            return document.WithSyntaxRoot(newRoot);
        }
    }
}
```

اولین کاری که در یک code fixer باید مشخص شود، تعیین FixableDiagnosticIds آن است. یعنی کدام آنالایزرهای از پیش تعیین شده‌ای قرار است توسط این code fixer مدیریت شوند که در اینجا همان Id آنالایزر قسمت قبل را مشخص کرده‌ایم. به این ترتیب ویژوال استودیو تشخیص می‌دهد که خطای گزارش شده‌ی توسط CodingStandardsAnalyzer قسمت قبل، توسط کدام code fixer موجود قابل رفع است.
کاری که در متد RegisterCodeFixesAsync انجام می‌شود، مشخص کردن اولین مکانی است که مشکلی در آن گزارش شده‌است. سپس به این مکان منوی Make Private با متد متناظر با آن معرفی می‌شود. در این متد، اولین توکن public، مشخص شده و سپس با یک توکن private جایگزین می‌شود. اکنون این syntax tree بازنویسی شده بازگشت داده می‌شود. با Syntax Factory در قسمت سوم آشنا شدیم.

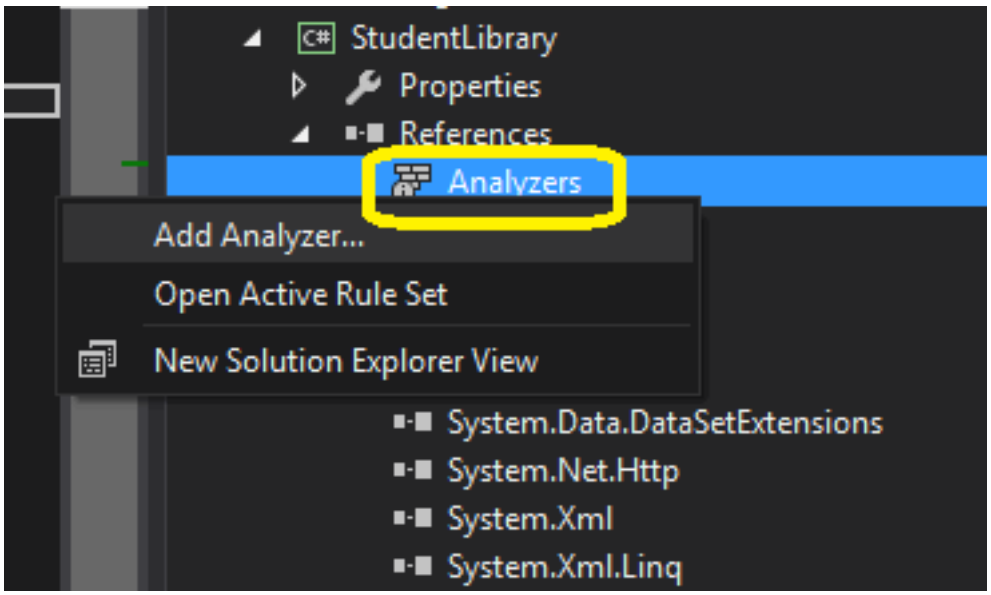
خوب، تا اینجا یک analyzer و یک code fixer را تهیه کرده‌ایم. برای آزمایش آن دکمه‌ی F5 را فشار دهید تا وهله‌ای جدید از ویژوال استودیو که این آنالایزر جدید در آن نصب شده‌است، آغاز شود. البته باید دقت داشت که در اینجا باید پروژه‌ی **Vsix**. CodingStandards را به عنوان پروژه‌ی آغازین ویژوال استودیو معرفی کنید؛ چون پروژه‌ی class library آنالایزرها را نمی‌توان مستقیما اجرا کرد. همچنین یکبار کل solution را نیز build کنید.
پس از اینکه وهله‌ی جدید ویژوال استودیو شروع به کار کرد (بار اول اجرای آن کمی زمانبر است؛ زیرا باید تنظیمات وهله‌ی ویژه‌ی اجرای افزونه‌ها را از ابتدا اعمال کند)، همان پروژه‌ی Student ابتدای بحث را در آن باز کنید.



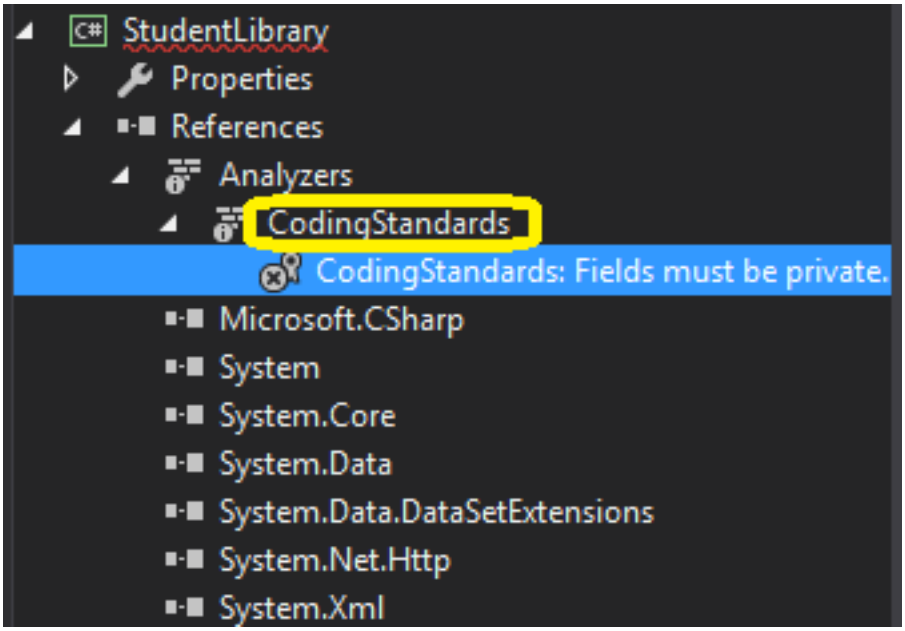
نتیجه‌ی اعمال این افزونه‌ی جدید را در تصویر فوق ملاحظه می‌کنید. زیر سطرهای دارای فیلد عمومی، خط قرمز کشیده شده‌است (به علت تعریف DiagnosticSeverity.Error). همچنین حالت فعلی و حالت برطرف شده را نیز با رنگ‌های قرمز و سبز می‌توان مشاهده کرد. کلیک بر روی گزینه‌ی make private، سبب اصلاح خودکار آن سطر می‌گردد.

روش دوم آزمایش یک Roslyn Analyzer

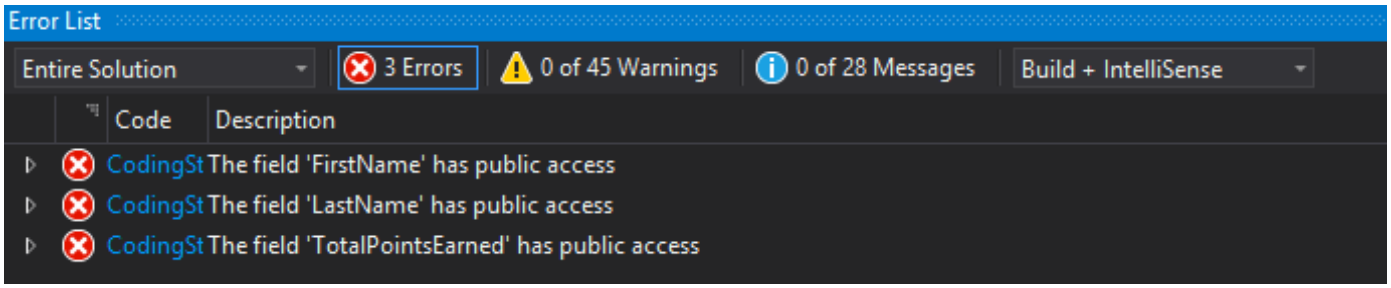
همانطور که از انتهای بحث قسمت دوم به‌خاطر دارید، این آنالایزرها را می‌توان به کامپایلر نیز معرفی کرد. روش انجام اینکار در ویژوال استودیوی 2015 در تصویر ذیل نمایش داده شده‌است.



نود references را باز کرده و سپس بر روی گزینه‌ی analyzers کلیک راست نمائید. در اینجا گزینه‌ی Add analyzer را انتخاب کنید. در صفحه‌ی باز شده بر روی دکمه‌ی browse کلیک کنید. در اینجا می‌توان فایل اسمبلی موجود در پوشه‌ی CodingStandards\bin\Debug را به آن معرفی کرد.



بلافاصله پس از معرفی این اسمبلی، آنالایزر آن شناسایی شده و همچنین فعال می‌گردد.



در این حالت اگر برنامه را کامپایل کنیم، با خطاهای جدید فوق متوقف خواهیم شد و برنامه کامپایل نمی‌شود (به علت تعریف DiagnosticSeverity.Error).