عنوان: **Roslyn #1**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۲۶ ۱۴:۳۵
آدرس: www.dotnettips.info
گروه‌ها: Visual Studio, Roslyn

معرفی Roslyn

سکوی کامپایلر دات نت یا Roslyn (با تلفظ «رازلین») بازنویسی مجدد کامپایلرهای VB.NET و C# توسط همین زبان‌ها است. این سکوی کامپایلر به همراه یک سری کتابخانه و اسمبلی ارائه می‌شود که امکان آنالیز زبان‌های مدیریت شده را به صورت مستقل و یا یکپارچه‌ی با ویژوال استودیو، فراهم می‌کنند. برای نمونه در VS.NET 2015 تمام سرویس‌های زبان‌های موجود، با Roslyn API جایگزین و بازنویسی شده‌اند. نمونه‌هایی از این سرویس‌های زبان‌ها، شامل Intellisense و مرور کدها مانند go to references and definitions، به همراه امکانات Refactoring می‌شوند. به علاوه به کمک Roslyn می‌توان یک کامپایلر و ابزارهای مرتبط با آن، مانند FxCop را تولید کرد و یا در نهایت یک فایل اسمبلی نهایی را از آن تحویل گرفت.

چرا مایکروسافت Roslyn را تولید کرد؟

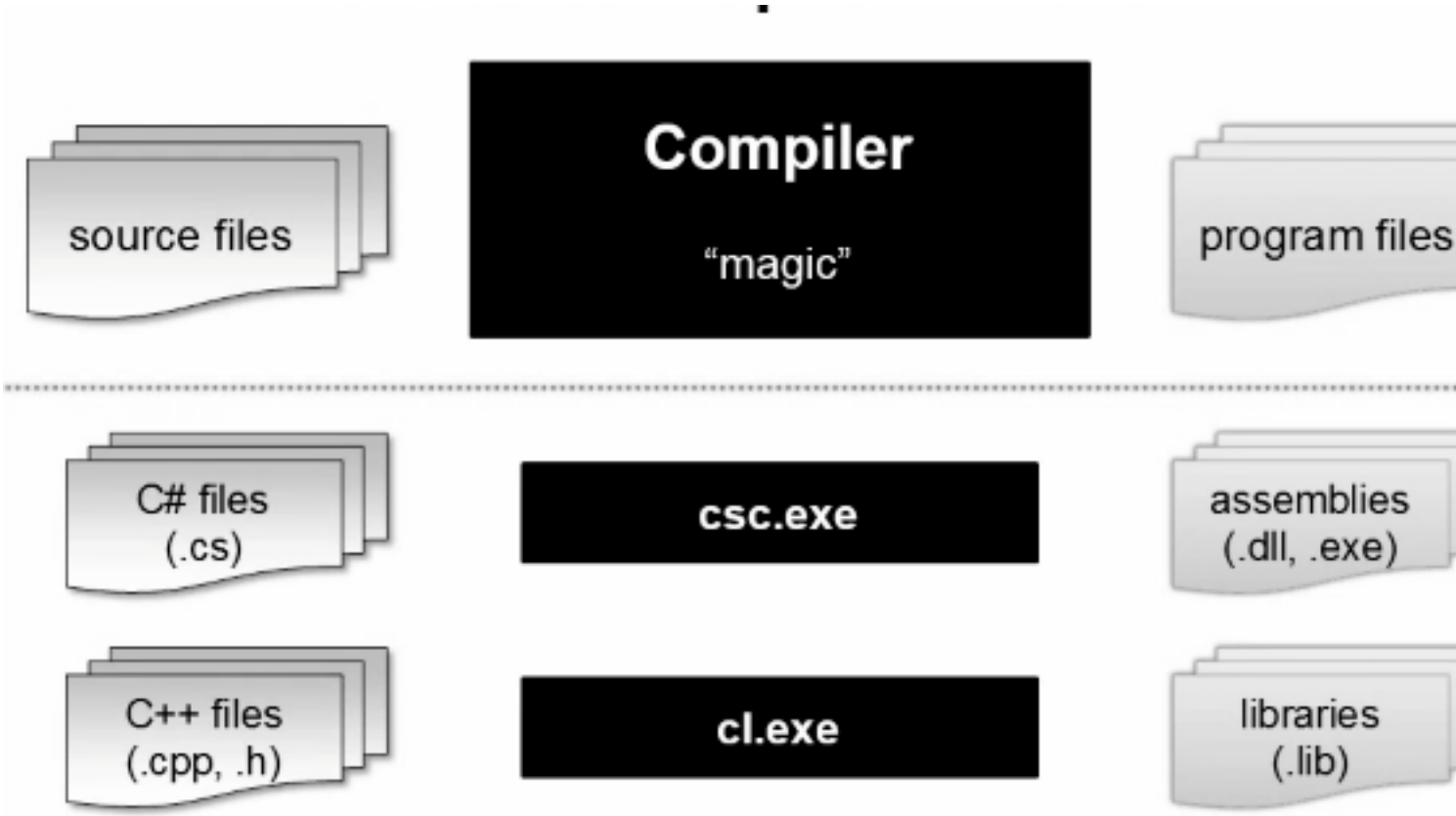
پیش از پروژه‌ی Roslyn، کامپایلرهای VB.NET و C# با زبان C++ نوشته شده بودند؛ از این جهت که در اواخر دهه‌ی 90 که کار تولید سکوی دات نت در حال انجام بود، هنوز امکانات کافی برای نوشتن این کامپایلرها با زبان‌های مدیریت شده وجود نداشت و همچنین زبان محبوب کامپایلر نویسی در آن دوران نیز C++ بود. این انتخاب در دراز مدت مشکلاتی مانند کاهش انعطاف پذیری و productivity تیم کامپایلر نویس را با افزایش تعداد سطرهای کامپایلر نوشته شده به همراه داشت و افزودن ویژگی‌های جدید را به زبان‌های VB.NET و C# سخت‌تر و سخت‌تر کرده بود. همچنین در اینجا برنامه نویس‌های تیم کامپایلر مدام مجبور بودند که بین زبان‌های مدیریت شده و مدیریت نشده سوئیچ کنند و امکان استفاده‌ی همزمان از زبان‌هایی را که در حال توسعه‌ی آن هستند، نداشتند.
این مسایل سبب شدند تا در طی بیش از یک دهه، چندین نوع کامپایلر از صفر نوشته شوند:
- کامپایلرهای خط فرمانی مانند csc.exe و vbc.exe
- کامپایلر پشت صحنه‌ی ویژوال استودیو (برای مثال کشیدن یک خط قرمز زیر مشکلات دستوری موجود)
- کامپایلر snippetها در immediate window ویژوال استودیو

هر کدام از این کامپایلرها هم برای حل مسایلی خاص طراحی شده‌اند. کامپایلرهای خط فرمانی، با چندین فایل ورودی، به همراه ارائه‌ی تعدادی زیادی خطا و اخطار کار می‌کنند. کامپایلر پشت صحنه‌ی ویژوال استودیوهای تا پیش از نسخه‌ی 2015، تنها با یک تک فایل در حال استفاده، کار می‌کند و همچنین باید به خطاهای رخ داده نیز مقاوم باشد و بیش از اندازه گزارش خطا ندهد. برای مثال زمانیکه کاربر در حالت تایپ یک سطر است، بدیهی است تا اتمام کار، این سطر فاقد ارزش دستوری صحیحی است و کامپایلر باید به این مساله دقت داشته باشد و یا کامپایلر snippetها تنها جهت ارزیابی یک تک سطر از دستورات وارد شده، طراحی شده‌است.

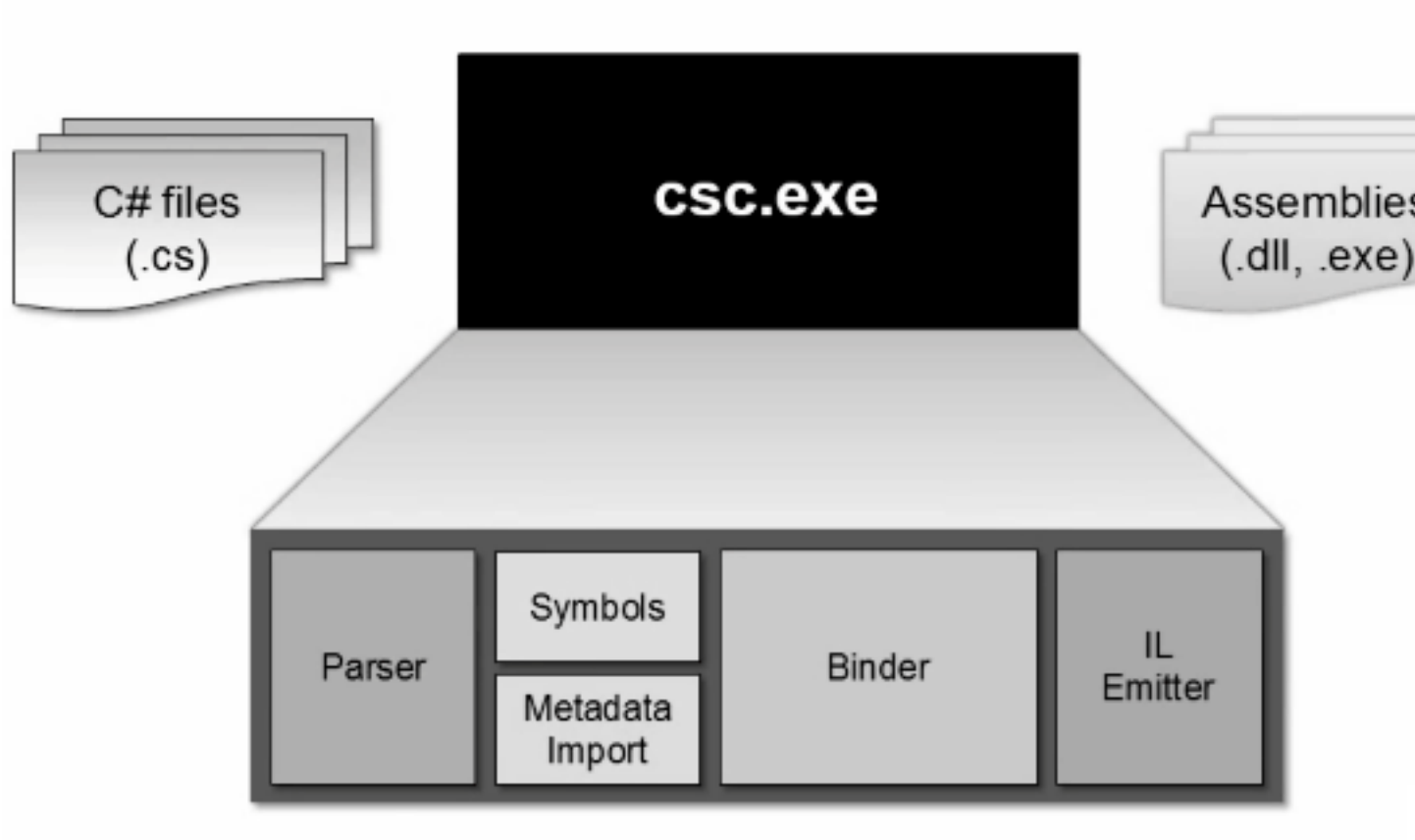
با توجه به این مسایل، مایکروسافت از بازنویسی سکوی کامپایلر دات نت این اهداف را دنبال می‌کند:
- بالا بردن سرعت افزودن قابلیت‌های جدید به زبان‌های موجود
- سبک کردن حجم کاری کامپایلر نویسی و کاهش تعداد آن‌ها به یک مورد
- بالا بردن دسترسی پذیری به API کامپایلرها
برای مثال اکنون برنامه نویس‌ها بجای اینکه یک فایل cs را به کامپایلر csc.exe ارائه کنند و یک خروجی باینری دریافت کنند، امکان دسترسی به syntax trees، semantic analysis و تمام مسایل پشت صحنه‌ی یک کامپایلر را دارند.
- ساده سازی تولید افزونه‌های مرتبط با زبان‌های مدیریت شده.
اکنون برای تولید یک آنالیز کننده‌ی سفارشی، نیازی نیست هر توسعه دهنده‌ای شروع به نوشتن امکانات پایه‌ای یک کامپایلر کند. این امکانات به صورت یک API عمومی در دسترس برنامه نویس‌ها قرار گرفته‌اند.
- آموزش مسایل درونی یک کامپایلر و همچنین ایجاد اکوسیستمی از برنامه نویس‌های علاقمند در اطراف آن.
همانطور که اطلاع دارید، Roslyn به صورت سورس باز در GitHub در دسترس عموم است .

تفاوت Roslyn با کامپایلرهای سنتی

اکثر کامپایلرهای موجود به صورت یک جعبه‌ی سیاه عمل می‌کنند. به این معنا که تعدادی فایل ورودی را دریافت کرده و در نهایت یک خروجی باینری را تولید می‌کنند. اینکه در این میان چه اتفاقاتی رخ می‌دهد، از دید استفاده کننده مخفی است.

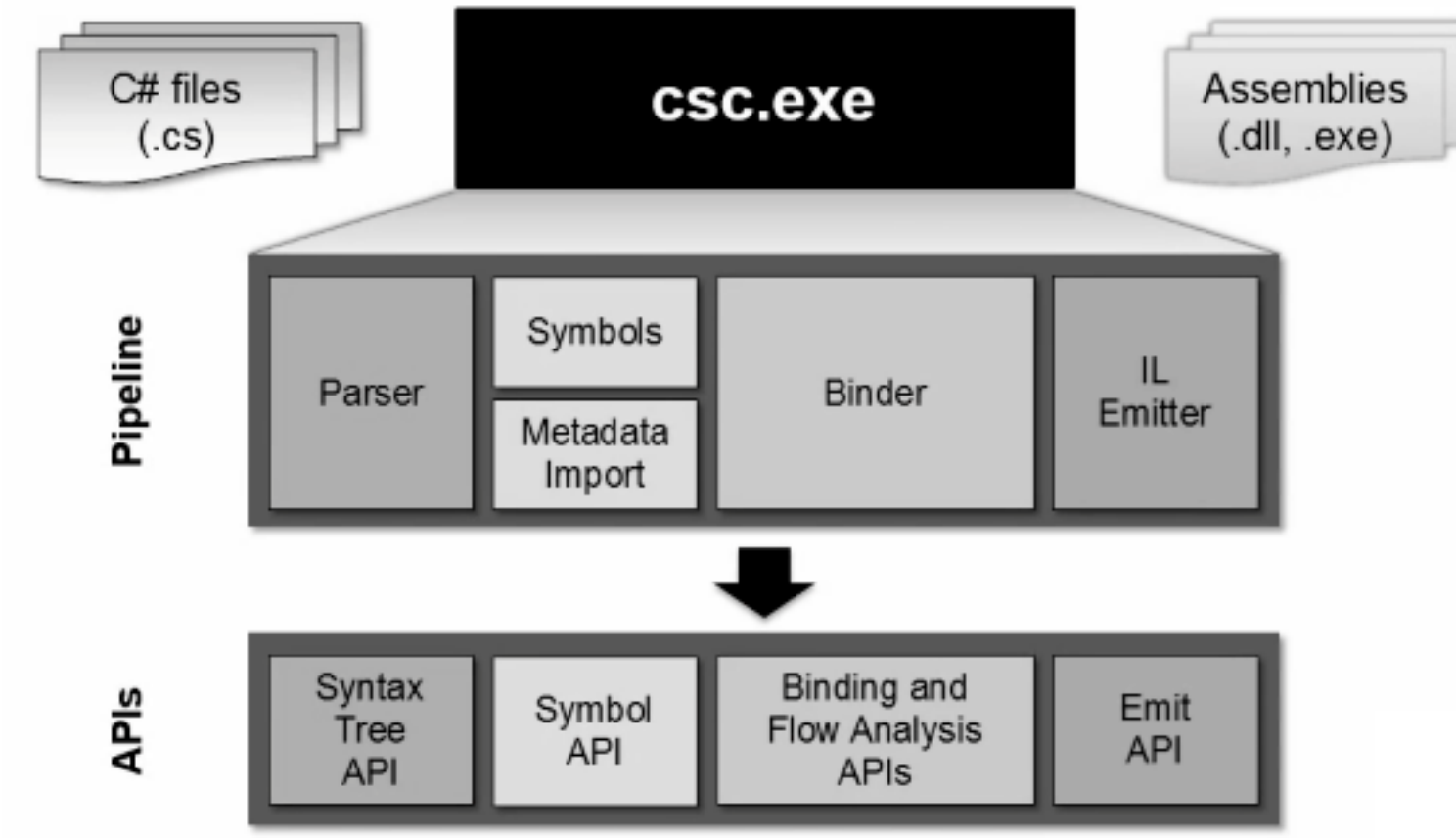


نمونه‌ای از این کامپایلرهای جعبه سیاه را در تصویر فوق مشاهده می‌کنید. در اینجا شاید این سؤال مطرح شود که در داخل جعبه‌ی سیاه کامپایلر سی‌شارپ، چه اتفاقاتی رخ می‌دهد؟

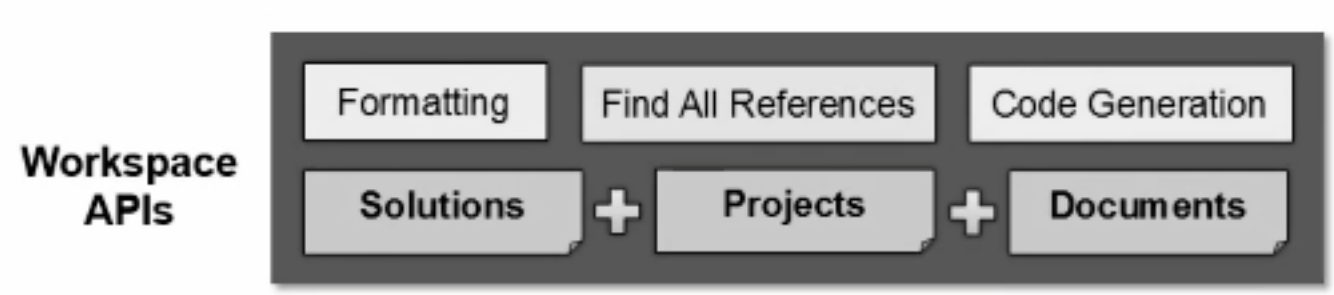


خلاصه‌ی مراحل رخ داده در کامپایلر سی‌شارپ را در تصویر فوق ملاحظه می‌کنید. در اینجا ابتدا کار parse اطلاعات متنی دریافتی شروع می‌شود و از روی آن syntax tree تولید می‌شود. در مرحله‌ی بعد مواردی مانند ارجاعاتی به mscorlib و امثال آن پردازش می‌شوند. در مرحله‌ی binder کار پردازش حوزه‌ی دید متغیرها، اشیاء و اتصال آن‌ها به هم انجام می‌شود. در مرحله‌ی آخر، کار تولید کدهای IL و اسمبلی باینری نهایی صورت می‌گیرد.

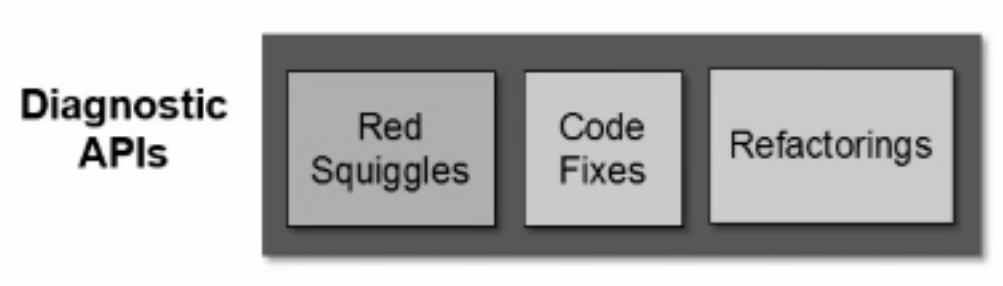
با معرفی Roslyn، این جعبه‌ی سیاه، به صورت یک API عمومی در دسترس برنامه نویس‌ها قرار گرفته‌است:



همانطور که مشاهده می‌کنید، هر مرحله‌ی کامپایل جعبه‌ی سیاه، به یک API عمومی Roslyn نگاشت شده‌است. برای مثال Parser به Syntax tree API نگاشت شده‌است. به علاوه این API صرفا به موارد فوق خلاصه نمی‌شود و همانطور که پیشتر نیز ذکر شد، برای اینکه بتواند جایگزین سه نوع کامپایلر موجود شود، به همراه Workspace API نیز می‌باشد:



Roslyn امکان کار با یک Solution و فایل‌های آن را دارد و شامل سرویس‌های زبان‌های مورد نیاز در ویژوال استودیو نیز می‌شود. برفراز Workspace API، یک مجموعه API دیگر به نام Diagnostics API تدارک دیده شده‌است تا برنامه نویس‌ها بتوانند امکانات Refactoring جانبی را توسعه داده و یا در جهت بهبود کیفیت کدهای نوشته شده، اخطارهایی را به برنامه نویس‌ها تحت عنوان Code fixes و آنالیز کننده‌ها، ارائه دهند.

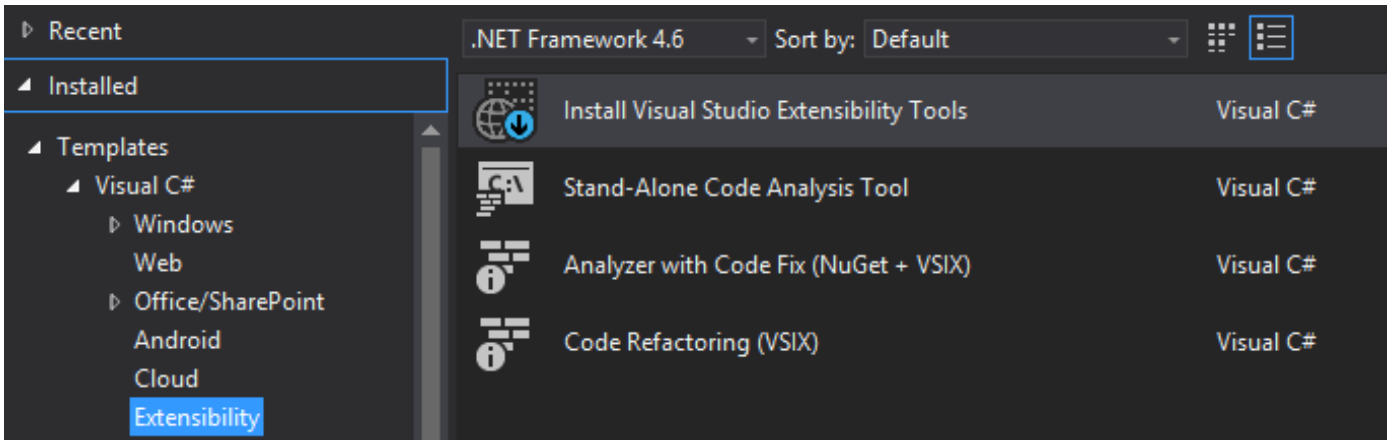

Diagnostic APIs
Red Squiggles
Code Fixes
Refactorings

عنوان: **Roslyn #2**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۲۷ ۱۳:۵
آدرس: www.dotnettips.info
گروه‌ها: Roslyn

شروع به کار با Roslyn

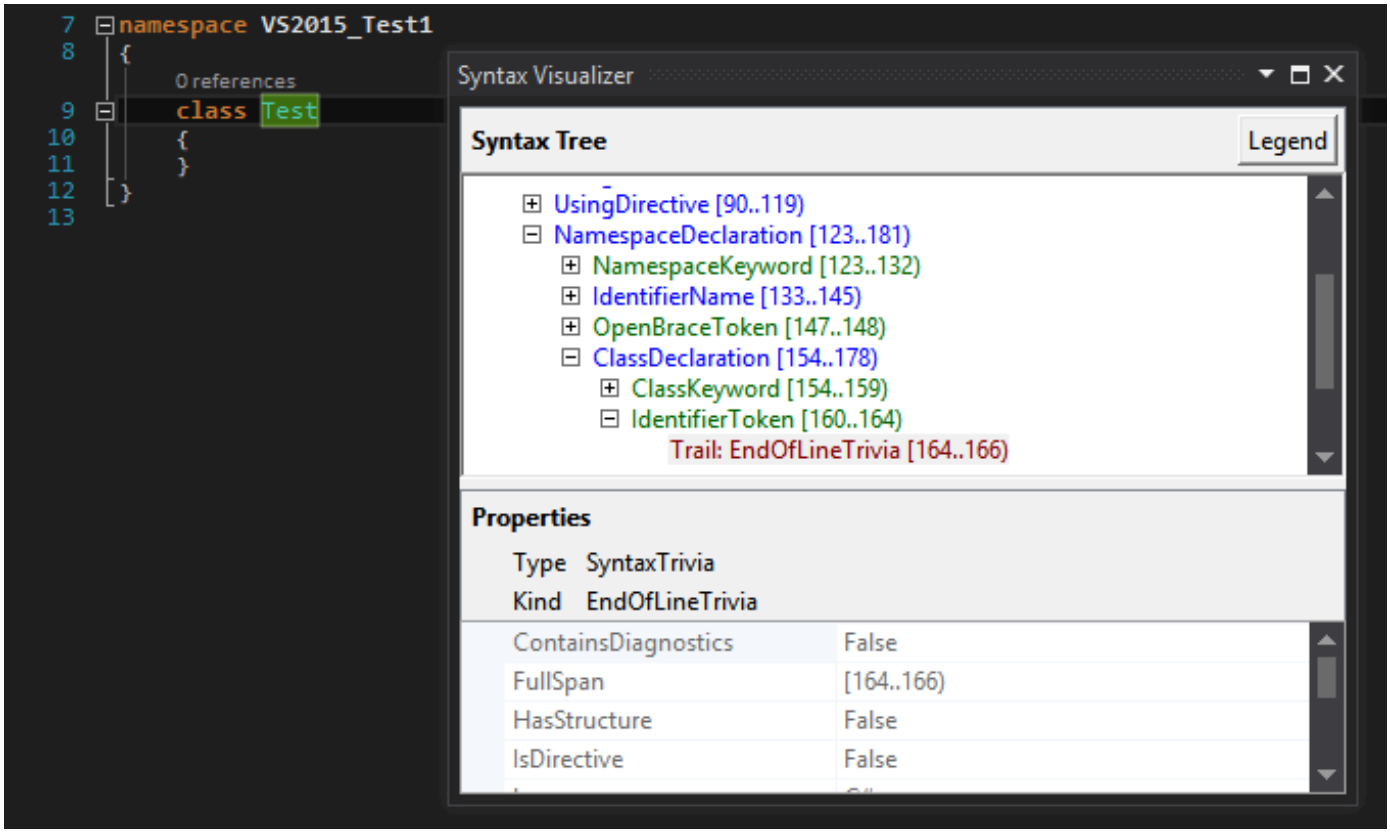
Roslyn از زمان ارائه‌ی نگارش Visual Studio 14 CTP3 با ویژوال استودیو یکپارچه شد. بنابراین اگر از نگارش نهایی آن یعنی Visual Studio 2015 استفاده می‌کنید، اولین پیشنیاز کار با آن را در اختیار دارید. البته نگارش پیش نمایش آن نیز برای VS 2013 ارائه شد که در این تاریخ منسوخ شده درنظر گرفته می‌شود و دیگر به روز رسانی نخواهد شد. بنابراین نیاز است حتما Visual Studio 2015 را نصب کنید.
در حین نصب Visual Studio 2015 نیز باید گزینه‌ی نصب SDK آن‌را انتخاب کرده باشید، تا امکان تولید فایل‌های VSIX مرتبط را پیدا کنید. از این جهت که قالب‌های پروژه‌ی Roslyn، امکان تولید افزونه‌های آنالیز کدها را نیز میسر می‌کنند.
علاوه بر این‌ها نیاز است Syntax Visualizer و قالب‌های پروژه‌ی مخصوص Roslyn را نیز جداگانه نصب کنید. برای این منظور به آدرس ذیل مراجعه کرده و افزونه‌ی آن‌را نصب کنید.
.NET Compiler Platform SDK

پس از نصب این افزونه، دو قابلیت جدید به Visual studio اضافه می‌شوند:
الف) در منوی view، ذیل گزینه‌ی other windows، گزینه‌ی جدیدی به نام Syntax visualizer اضافه شده‌است.
ب) ذیل گزینه‌ی extensibility پنجره‌ی new project، تعدادی قالب پروژه‌ی جدید مخصوص Roslyn نصب شده‌اند.



البته باید دقت داشت که این قالب‌های جدید برای نمایش در این لیست، حداقل نیاز به انتخاب دات نت 4.5.2 را دارند.

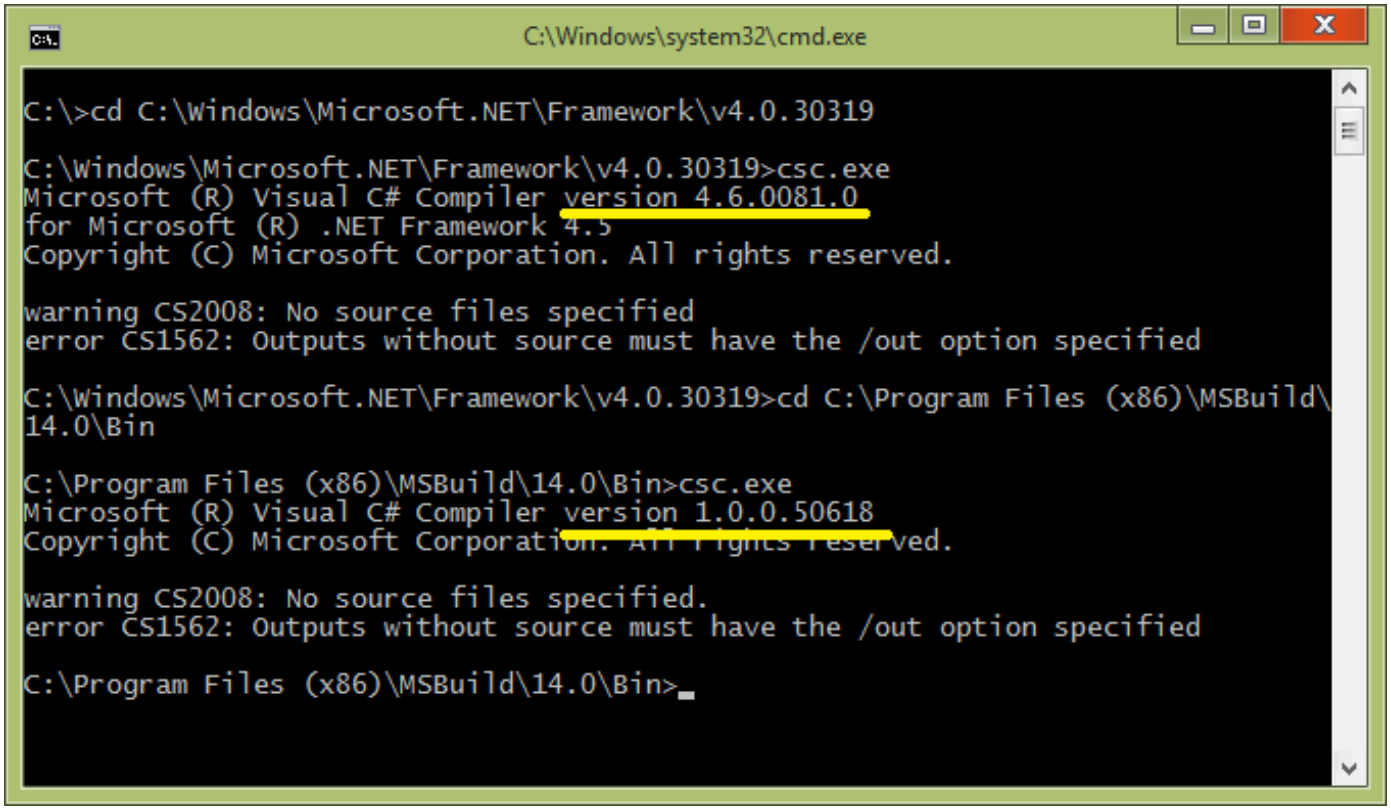
از Syntax visualizer جهت مشاهده‌ی ریز جزئیات ساختار دستوری کدهای جاری استفاده می‌شود:



زمانیکه گزینه‌ی View->Other windows->Syntax visualizer را اجرا کردید، اندکی صبر کنید تا بارگذاری شود و سپس اشاره‌گر ویرایشگر را به قسمتی دلخواه حرکت دهید. در این حالت می‌توان ساختار کدها را بر اساس تفسیر Roslyn مشاهده کرد. همچنین اگر در Syntax visualizer یک نود را انتخاب کنید، بلافاصله معادل آن در ادیتور ویژوال استودیو انتخاب می‌شود. از این ابزار جهت تولید آنالایزرهای مبتنی بر Roslyn زیاد استفاده می‌شود.

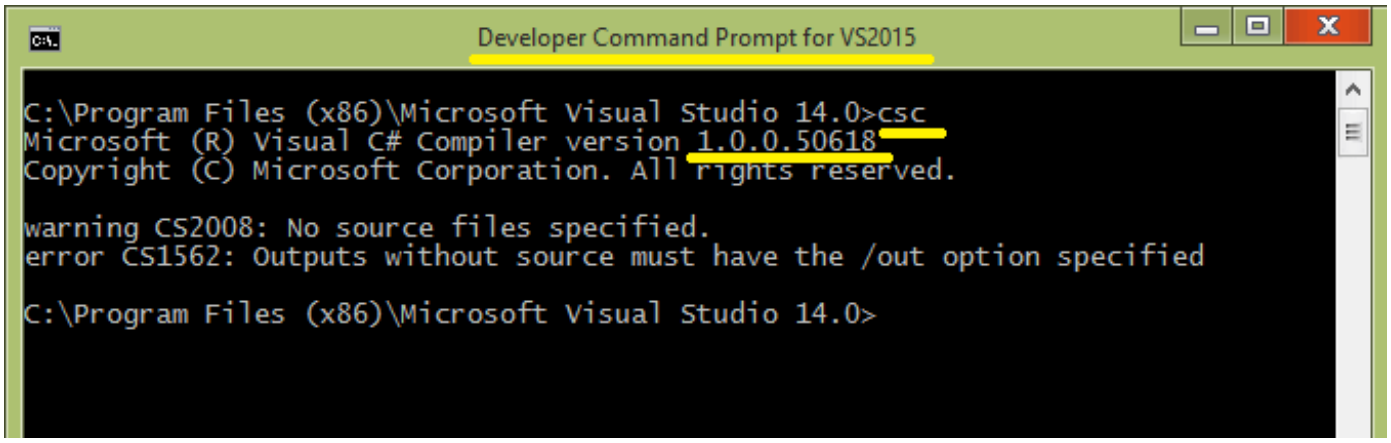
تغییرات خط فرمان Visual Studio 2015 نسبت به نگارش‌های پیشین آن

خط فرمان Visual Studio 2015 به همراه کامپایلر قدیمی خط فرمان C# 5 و همچنین کامپایلر جدید مبتنی بر Roslyn مخصوص C# 6 است. این کامپایلرها را در مسیرهای ذیل می‌توانید پیدا کنید:
کامپایلر جدید مبتنی بر Roslyn در مسیر C:\Program Files (x86)\MSBuild\14.0\Bin قرار دارد و کامپایلر قدیمی C# 5 در مسیر نصب دات نت فریم ورک یعنی C:\Windows\Microsoft.NET\Framework\v4.0.30319 قرار گرفته‌است.



همانطور که مشاهده می‌کنید، نگارش کامپایلر مبتنی بر Roslyn، یک است و شماره build آن، بیانگر تاریخ کامپایل آن است: 2015/06/18=50618

اکنون شاید این سؤال مطرح شود که کامپایلر پیش فرض کدام است؟ برای یافتن آن، به منوی استارت ویندوز مراجعه کرده و گزینه‌ی developer command prompt for vs 2015 را اجرا کنید. در اینجا اگر دستور csc را اجرا کنید، خروجی آن همان کامپایلر مبتنی بر Roslyn است:



در همینجا اگر سوئیچ ? را برای مشاهده‌ی راهنمای کامپایلر خط فرمان Roslyn صادر کنید، یکی از گزینه‌های جدید آن، سوئیچ analyzer است:

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0>csc /?
```

```
Microsoft (R) Visual C# Compiler version 1.0.0.50618
Copyright (C) Microsoft Corporation. All rights reserved.


Visual C# Compiler Options

 /analyzer:<file list> Run the analyzers from this assembly
(Short form: /a)
```

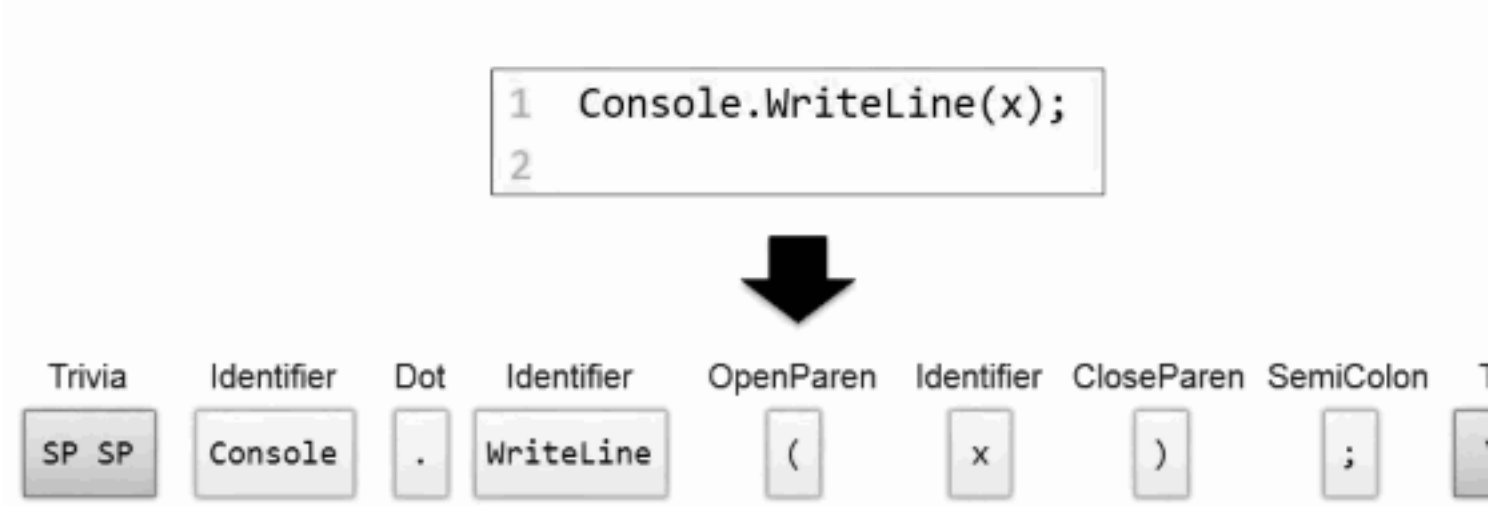
آنالایزرها اسمبلی‌ها و افزونه‌هایی هستند که قابلیت وارد شدن به pipeline کامپایلر را داشته و امکان دخالت در صدور خطاها و اخطارهای صادر شده‌ی توسط آن‌را دارند. به عبارتی این کامپایلر خط فرمان جدید، افزونه پذیر است.

همچنین این کامپایلر نسبت به نگارش قبلی آن، دارای سوئیچ و گزینه‌ی parallel نیز می‌باشد که به کمک ساختارهای داده‌ی immutable جدید دات نت مسیر شده‌است.

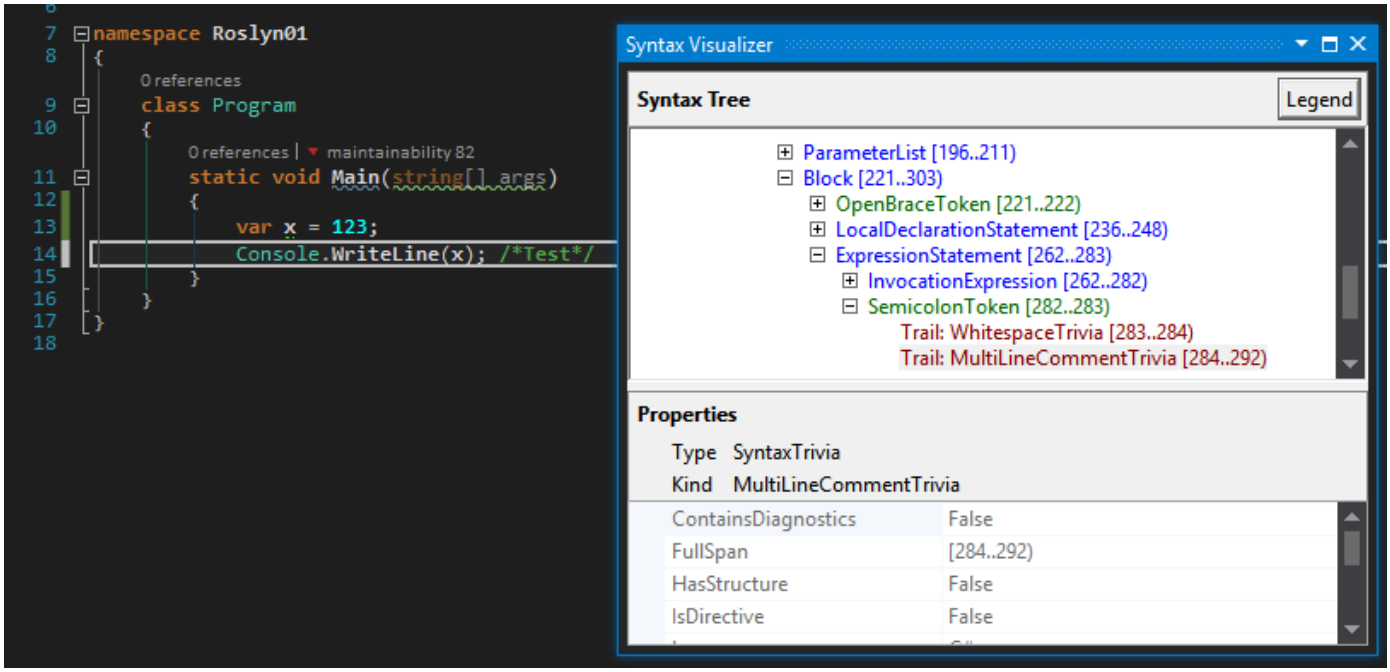
عنوان: **Roslyn #3**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۲۸ ۱۶:۲۰
آدرس: www.dotnettips.info
گروه‌ها: Roslyn

بررسی Syntax tree

زمانیکه صحبت از Syntax می‌شود، منظور نمایش متنی سورس کدها است. برای بررسی و آنالیز آن، نیاز است این نمایش متنی، به ساختار داده‌ای ویژه‌ای به نام Syntax tree تبدیل شود و این Syntax tree مجموعه‌ای است از tokenها. Tokenها بیانگر المان‌های مختلف یک زبان، شامل کلمات کلیدی، عملگرها و غیره هستند.



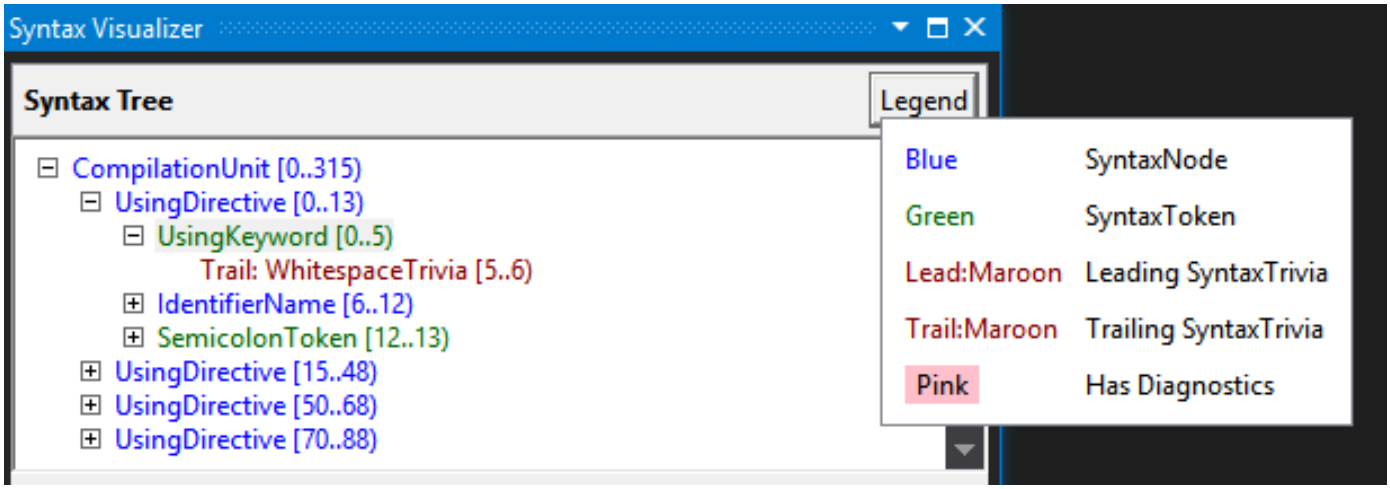
در تصویر فوق، مراحل تبدیل یک قطعه کد C# را به مجموعه‌ای از tokenهای معادل آن مشاهده می‌کنید. علاوه بر این‌ها، Roslyn syntax tree شامل موارد ویژه‌ای به نام Trivia نیز هست. برای مثال در حین نوشتن کدها، در ابتدای سطرها تعدادی space یا tab وجود دارند و یا در این بین ممکن است کامنتی نوشته شود. هرچند این موارد از دیدگاه یک کامپایلر بی‌معنا هستند، اما ابزارهای Refactoring ایی که به Trivia دقت نداشته باشند، خروجی کد به هم ریخته‌ای را تولید خواهند کرد و سبب سردرگمی استفاده کنندگان می‌شوند.



در تصویر فوق، اشاره‌گر ادیتور پس از تایپ semicolon قرار گرفته‌است. در این حالت می‌توانید دو نوع trivia مخصوص فضای خالی و کامنت‌ها را در syntax visualizer، مشاهده کنید.

به علاوه پس از هر token بازه‌ای از اعداد را مشاهده می‌کنید که بیانگر محل قرارگیری آن‌ها در سورس کد هستند. این محل‌ها جهت ارائه‌ی خطاهای دقیق مرتبط با آن نقاط، بسیار مفید هستند.

یک Syntax tree از مجموعه‌ای از syntax nodes تشکیل می‌شود و هر node شامل مواردی مانند تعاریف، عبارات و امثال آن است. در افزونه‌ی Syntax visualizer نودهایی که رنگ قرمز متمایل به قهوه‌ای دارند، بیانگر نودهای Trivia، نودهای آبی، Syntax nodes و نودهای سبز، Syntax token هستند.



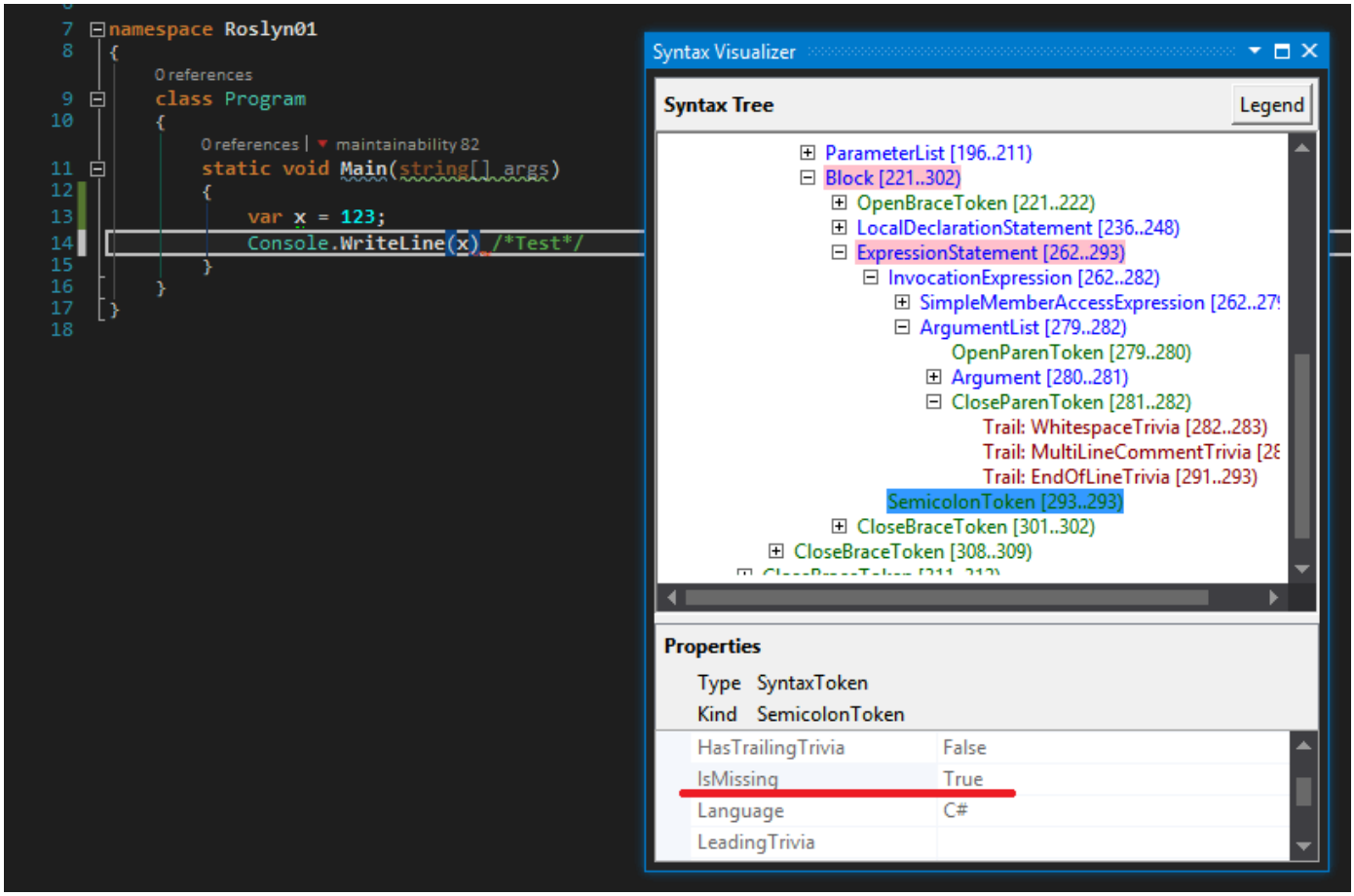
مفاهیم این رنگ‌ها را با کلیک بر روی دکمه‌ی Legend هم می‌توان مشاهده کرد.

تفاوت Syntax با Semantics

در Roslyn امکان کار با Syntax و Semantics کدها وجود دارد.
یک Syntax، از گرامر زبان خاصی پیروی می‌کند. در Syntax اطلاعات بسیار زیادی وجود دارند که معنای برنامه را تغییر نمی‌دهند؛ مانند کامنت‌ها، فضاهای خالی و فرمت ویژه‌ی کدها. البته فضاهای خالی در زبان‌هایی مانند پایتون دارای معنا هستند؛ اما در سی‌شارپ خیر. همچنین در Syntax، توافق نامه‌ای وجود دارد که بیانگر تعدادی واژه‌ی از پیش رزرو شده، مانند کلمات کلیدی هستند.
اما Semantics در نقطه‌ی مقابل Syntax قرار می‌گیرد و بیانگر معنای سورس کد است. برای مثال در اینجا تقدم و تاخر عملگرها مفهوم پیدا می‌کنند و یا اینکه Type system چیست و چه نوع‌هایی را می‌توان به دیگری نسبت داد و تبدیل کرد. عملیات Binding در این مرحله رخ می‌دهد و مفهوم identifierها را مشخص می‌کند. برای مثال x در این قسمت از سورس کد، به چه معنایی است و به کجا اشاره می‌کند؟

خواص ویژه‌ی Syntax tree در Roslyn

- تمام اجزای کد را شامل عناصر سازنده‌ی زبان و همچنین Trivia، به همراه دارد.
- API آن توسط کتابخانه‌های ثالث قابل دسترسی است.
- Immutable طراحی شده‌است. به این معنا که زمانیکه syntax tree توسط Roslyn ایجاد شد، دیگر تغییر نمی‌کند. به این ترتیب امکان دسترسی همزمان و موازی به آن بدون نیاز به انواع قفل‌های مسایل همزمانی وجود دارد. اگر کتابخانه‌ی ثالثی به Syntax tree ارائه شده دسترسی پیدا می‌کند، می‌تواند کاملا مطمئن باشد که این اطلاعات دیگر تغییری نمی‌کنند و نیازی به قفل کردن آن‌ها نیست. همچنین این مساله امکان استفاده‌ی مجدد از sub treeها را در حین ویرایش کدها میسر می‌کند. به آن‌ها mutating trees نیز گفته می‌شود.
- مقاوم است در برابر خطاها. اگر از قسمت اول به خاطر داشته باشید، Roslyn می‌بایستی جایگزین کامپایلر دومی به نام کامپایلر پس زمینه‌ی ویژوال استودیو که خطوط قرمزی را ذیل سطرهای مشکل دار ترسیم می‌کند، نیز می‌شد. فلسفه‌ی طراحی این کامپایلر، مقاوم بودن در برابر خطاهای تایپی و هماهنگی آن با تایپ کدها توسط برنامه نویس بود. Syntax tree در Roslyn نیز چنین خاصیتی را دارد و اگر مشغول به تایپ شوید، باز هم کار کرده و اینبار خطاهای موجود را نمایش می‌دهد که می‌تواند توسط ابزارهای نمایش دهنده‌ی ویژوال استودیو یا سایر ابزارهای ثالث استفاده شود.



برای نمونه در تصویر فوق، تایپ semicolon فراموش شده‌است؛ اما همچنان Syntax tree در دسترس است و به علاوه گزارش می‌دهد که semicolon مفقود است و تایپ نشده‌است.

Parse سورس کد توسط Roslyn

ابتدا یک پروژه‌ی کنسول ساده‌ی دات نت 4.6 را در VS 2015 آغاز کنید. سپس از طریق خط فرمان نیوگت، دستور ذیل را صادر نمائید:

```
PM> Install-Package Microsoft.CodeAnalysis
```

به این ترتیب API لازم جهت کار با Roslyn به پروژه اضافه خواهند شد.
سپس کدهای ذیل را به آن اضافه کنید:

```
using System;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace Roslyn01
{
    class Program
    {
        static void Main(string[] args)
        {
            parseText();
        }

        static void parseText()
        {
```

```
            var tree = CSharpSyntaxTree.ParseText("class Foo { void Bar(int x) {} }");
            Console.WriteLine(tree.ToString());
            Console.WriteLine(tree.GetRoot().NormalizeWhitespace().ToString());

            var res = SyntaxFactory.ClassDeclaration("Foo")
                .WithMembers(SyntaxFactory.List<MemberDeclarationSyntax>(new[] {
                    SyntaxFactory.MethodDeclaration(
                        SyntaxFactory.PredefinedType(
                            SyntaxFactory.Token(SyntaxKind.VoidKeyword)
                        ),
                        "Bar"
                    )
                    .WithBody(SyntaxFactory.Block())
                }))
                .NormalizeWhitespace();

            Console.WriteLine(res);
        }
    }
}
```

توضیحات:

کار Parse سورس کد دریافتی، بر اساس سرویس‌های زبان متناظر با آن‌ها آغاز می‌شود. برای مثال سرویس‌هایی مانند VisualBasicSyntaxTree و یا CSharpSyntaxTree مثال فوق که سورس کد مورد آنالیز آن، از نوع سی‌شارپ است.

این کلاس‌های Factory، دارای دو متد Create و ParseText هستند. کار متد ParseText آن مشخص است؛ یک قطعه‌ی متنی از کد را آنالیز کرده و معادل Syntax Tree آن‌را تولید می‌کند. متد Create آن، اشیایی مانند نودهای Syntax visualizer Syntax را دریافت کرده و بر اساس آن‌ها یک Syntax tree را تولید می‌کند.

کار با متد Create آنچنان ساده نیست. به همین جهت یکی از اعضای تیم Roslyn برنامه‌ای را به نام Roslyn Quoter ایجاد کرده‌است که نسخه‌ی آنلاین آن‌را در اینجا و سورس کد آن‌را در اینجا می‌توانید بررسی کنید.

جهت آزمایش، همان قطعه‌ی متنی سورس کد مثال فوق را در نسخه‌ی آنلاین آن جهت آنالیز و تولید ورودی متد Create، وارد کنید. خروجی آن‌را می‌توان مستقیما در متد Create بکار برد.

فرمت کردن خودکار کدها به کمک Roslyn

اگر بر روی tree حاصل، متد ToString را فراخوانی کنیم، خروجی آن مجددا سورس کد مورد آنالیز است. اگر علاقمند بودید که Roslyn به صورت خودکار کدهای ورودی را فرمت کند و تمام آن‌ها را در یک سطر نمایش ندهد، متد NormalizeWhitespace را بر روی ریشه‌ی Syntax tree فراخوانی کنید:

```
tree.GetRoot().NormalizeWhitespace().ToString()
```

اینبار خروجی فراخوانی فوق به صورت ذیل است:

```
class Foo
{
    void Bar(int x)
    {
    }
}
```

کوئری گرفتن از سورس کد توسط Roslyn

در ادامه قصد داریم با سه روش مختلف کوئری گرفتن از Syntax tree، آشنا شویم. برای این منظور متد ذیل را به پروژه‌ای که در ابتدای برنامه آغاز کردیم، اضافه کنید:

```
static void querySyntaxTree()
{
    var tree = CSharpSyntaxTree.ParseText("class Foo { void Bar() {} }");
    var node = (CompilationUnitSyntax)tree.GetRoot();
```

```
    // Using the object model
    foreach (var member in node.Members)
    {
        if (member.Kind() == SyntaxKind.ClassDeclaration)
        {
            var @class = (ClassDeclarationSyntax)member;

            foreach (var member2 in @class.Members)
            {
                if (member2.Kind() == SyntaxKind.MethodDeclaration)
                {
                    var method = (MethodDeclarationSyntax)member2;
                    // do stuff
                }
            }
        }
    }


    // Using LINQ query methods
    var bars = from member in node.Members.OfType<ClassDeclarationSyntax>()
               from member2 in member.Members.OfType<MethodDeclarationSyntax>()
               where member2.Identifier.Text == "Bar"
               select member2;
    var res = bars.ToList();


    // Using visitors
    new MyVisitor().Visit(node);
}
```

توضیحات:

روش اول کوئری گرفتن از Syntax tree، استفاده از object model آن است. در اینجا هربار، نوع و Kind هر نود را بررسی کرده و در نهایت به اجزای مدنظر خواهیم رسید. شروع کار هم با دریافت ریشه‌ی syntax tree توسط متد GetRoot و تبدیل نوع آن نود به CompilationUnitSyntax می‌باشد.

روش دوم استفاده از روش LINQ است؛ با توجه به اینکه ساختار یک Syntax tree بسیار شبیه است به LINQ to XML. در اینجا یک سری نود، ریشه و فرزندان آن‌ها را داریم که با روش LINQ بسیار سازگار هستند. برای نمونه در مثال فوق، در ریشه‌ی Parse شده، در تمام کلاس‌های آن، به دنبال متد یا متدهایی هستیم که نام آن‌ها Bar است.

و در نهایت روش مرسوم و متداول کار با Syntax trees، استفاده از الگوی Visitors است. همانطور که در کدهای دو روش قبل مشاهده می‌کنید، باید تعداد زیادی حلقه و if و else نوشت تا به جزء و المان مدنظر رسید. راه ساده‌تری نیز برای مدیریت این پیچیدگی وجود دارد و آن استفاده از الگوی Visitor است. کار این الگو ارائه‌ی متدهایی قابل override شدن است و فراخوانی آن‌ها، در طی حلقه‌هایی پشت صحنه که این Visitor را اجرا می‌کنند، صورت می‌گیرد. بنابراین در اینجا دیگر برای رسیدن به یک متد، حلقه نخواهید نوشت. تنها کاری که باید صورت گیرد، override کردن متد Visit المانی خاص در Syntax tree است.

هر نود در syntax tree دارای متدی است به نام Accept که یک Visitor را دریافت می‌کند. همچنین Visitorهای نوشته شده نیز دارای متد Visit یک نود هستند.

نمونه‌ای از این Visitors را در کلاس ذیل مشاهده می‌کنید:

```
class MyVisitor : CSharpSyntaxWalker
{
    public override void VisitMethodDeclaration(MethodDeclarationSyntax node)
    {
        if (node.Identifier.Text == "Bar")
        {
            // do stuff
        }

        base.VisitMethodDeclaration(node);
    }
}
```

در اینجا برای رسیدن به تعاریف متدها دیگر نیازی نیست تا حلقه نوشت. بازنویسی متد VisitMethodDeclaration، دقیقا همین کار را انجام می‌دهد و در طی پروسه‌ی Visit یک Syntax tree، اگر متدی در آن تعریف شده باشد، متد VisitMethodDeclaration حداقل یکبار فراخوانی خواهد شد.

کلاس پایه‌ی CSharpSyntaxWalker از کلاس CSharpSyntaxVisitor مشتق شده‌است و به تمام امکانات آن دسترسی دارد. علاوه

بر آن‌ها، کلاس CSharpSyntaxWalker به Tokens و Trivia نیز دسترسی دارد.
نحوه‌ی استفاده از Visitor سفارشی نوشته شده نیز به صورت ذیل است:

```
new MyVisitor().Visit(node);
```

در اینجا متد Visit این Visitor را بر روی نود ریشه‌ی Syntax tree اجرا کرده‌ایم.

عنوان: **Roslyn #4**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۲۹ ۱۴:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Roslyn

بررسی API کامپایل Roslyn

Compilation API، یک abstraction سطح بالا از فعالیت‌های کامپایل Roslyn است. برای مثال در اینجا می‌توان یک اسمبلی را از Syntax tree موجود، تولید کرد و یا جایگزین‌هایی را برای APIهای قدیمی CodeDOM و Reflection Emit ارائه داد. به علاوه این API امکان دسترسی به گزارشات خطاهای کامپایل را میسر می‌کند؛ به همراه دسترسی به اطلاعات Semantic analysis. در مورد تفاوت Syntax tree و Semantics در قسمت قبل بیشتر بحث شد.
با ارائه‌ی Roslyn، اینبار کامپایلرهای خط فرمان تولید شده مانند csc.exe، صرفا یک پوسته بر فراز Compilation API آن هستند. بنابراین دیگر نیازی به فراخوانی Process.Start بر روی فایل اجرایی csc.exe مانند یک سری کتابخانه‌های قدیمی نیست. در اینجا با کدنویسی، به تمام اجزاء و تنظیمات کامپایلر، دسترسی وجود دارد.

کامپایل پویای کد توسط Roslyn

برای کار با API کامپایل، سورس کد، به صورت یک رشته در اختیار کامپایلر قرار می‌گیرد؛ به همراه تنظیمات ارجاعاتی به اسمبلی‌هایی که نیاز دارد. سپس کار کامپایلر شروع خواهد شد و شامل مواردی است مانند تبدیل متن دریافتی به Syntax tree و همچنین تبدیل مواردی که اصطلاحا به آن‌ها Syntax sugars گفته می‌شود مانند خواص get و set دار به معادل‌های اصلی آن‌ها. در اینجا کار Semantic analysis هم انجام می‌شود و شامل تشخیص حوزه‌ی دید متغیرها، تشخیص overloadها و بررسی نوع‌های بکار رفته‌است. در نهایت کار تولید فایل باینری اسمبلی، از اطلاعات آنالیز شده صورت می‌گیرد. البته خروجی کامپایلر می‌تواند اسمبلی‌های exe یا dll، فایل XML مستندات اسمبلی و یا فایل‌های .netmudule و .winmdobj مخصوص WinRT هم باشد.
در ادامه، اولین مثال کار با Compilation API را مشاهده می‌کنید. پیشنیاز اجرای آن همان مواردی هستند که در قسمت قبل بحث شدند. یک برنامه‌ی کنسول ساده‌ی .NET 4.6 را آغاز کرده و سپس بسته‌ی نیوگت Microsoft.CodeAnalysis را در آن نصب کنید. در ادامه کدهای ذیل را به پروژه‌ی آماده شده اضافه کنید:

```
static void firstCompilation()
{
    var tree = CSharpSyntaxTree.ParseText("class Foo { void Bar(int x) {} }");

    var mscorlibReference = MetadataReference.CreateFromFile(typeof (object).Assembly.Location);

    var compilationOptions = new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary);

    var comp = CSharpCompilation.Create("Demo")
                                .AddSyntaxTrees(tree)
                                .AddReferences(mscorlibReference)
                                .WithOptions(compilationOptions);

    var res = comp.Emit("Demo.dll");

    if (!res.Success)
    {
        foreach (var diagnostic in res.Diagnostics)
        {
            Console.WriteLine(diagnostic.GetMessage());
        }
    }
}
```

در اینجا نحوه‌ی کامپایل پویای یک قطعه کد متنی سی‌شارپ را به DLL معادل آن مشاهده می‌کنید. مرحله‌ی اول اینکار، تولید Syntax tree از رشته‌ی متنی دریافتی است. سپس متد CSharpCompilation.Create یک وهله از Compilation API مخصوص C# را آغاز می‌کند. این API به صورت Fluent طراحی شده‌است و می‌توان سایر قسمت‌های آن‌را به همراه یک دات پس از ذکر متد، به طول زنجیره‌ی فراخوانی، اضافه کرد. برای نمونه در این مثال، نحوه‌ی افزودن ارجاعی را به اسمبلی mscorlib که System.Object در آن قرار دارد و همچنین ذکر نوع خروجی DLL یا DynamicallyLinkedLibrary را ملاحظه می‌کنید. اگر این تنظیم

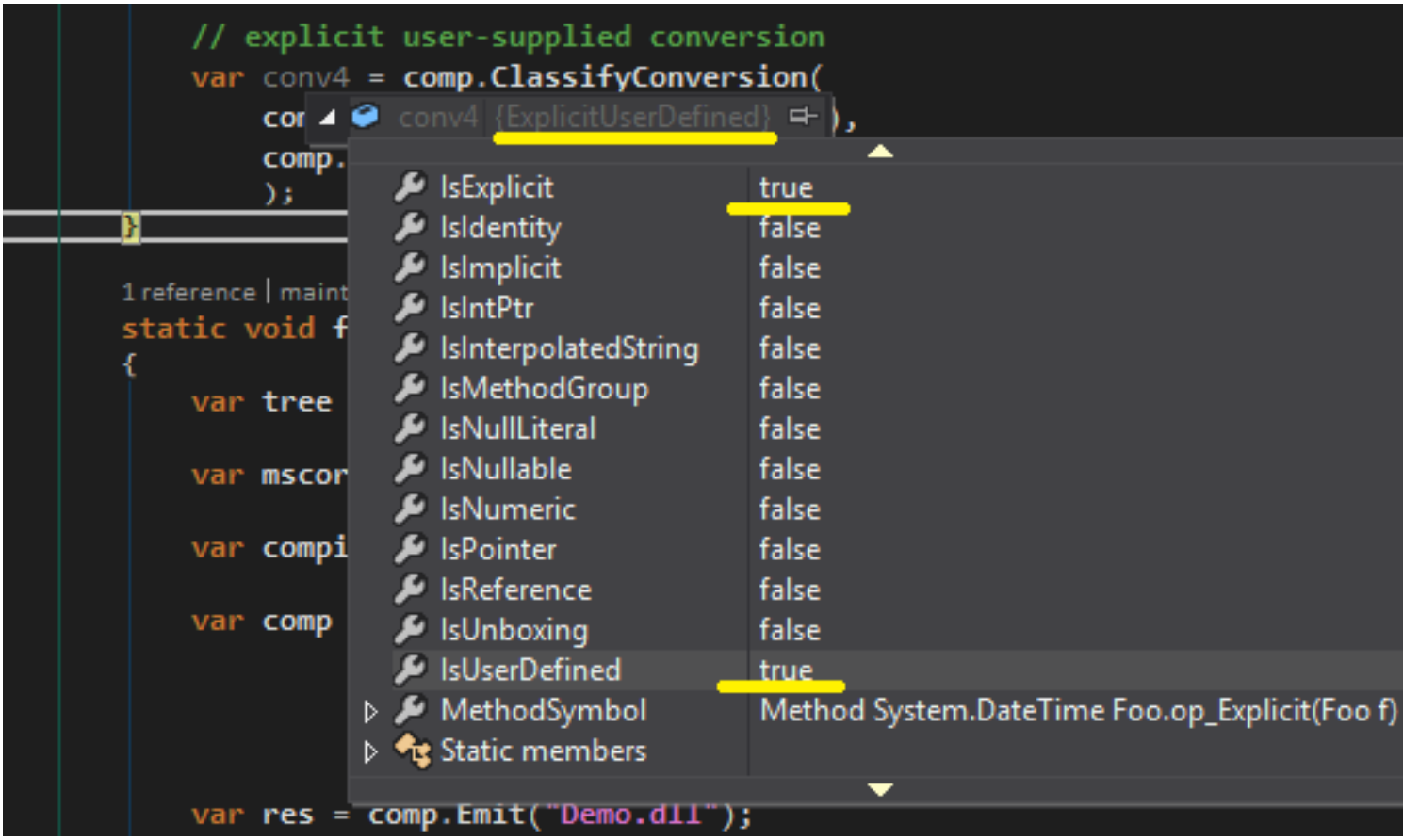
ذکر نشود، خروجی پیش فرض از نوع .exe خواهد بود و اگر mscorlib را اضافه نکنیم، نوع int سورس کد ورودی، شناسایی نشده و برنامه کامپایل نمی‌شود.
متدهای تعریف شده توسط Compilation API به یک s جمع، ختم می‌شوند؛ به این معنا که در اینجا در صورت نیاز، چندین Syntax tree یا ارجاع را می‌توان تعریف کرد.
پس از وهله سازی Compilation API و تنظیم آن، اکنون با فراخوانی متد Emit، کار تولید فایل اسمبلی نهایی صورت می‌گیرد. در اینجا اگر خطایی وجود داشته باشد، استثنایی را دریافت نخواهید کرد. بلکه باید خاصیت Success نتیجه‌ی آن‌را بررسی کرده و درصورت موفقیت آمیز نبودن عملیات، خطاهای دریافتی را از مجموعه‌ی Diagnostics آن دریافت کرد. کلاس Diagnostic، شامل اطلاعاتی مانند محل سطر و ستون وقوع مشکل و یا پیام متناظر با آن است.

معرفی مقدمات Semantic analysis

Compilation API به اطلاعات Semantics نیز دسترسی دارد. برای مثال آیا Type A قابل تبدیل به Type B هست یا اصلا نیازی به تبدیل ندارد و به صورت مستقیم قابل انتساب هستند؟ برای درک بهتر این مفهوم نیاز است یک مثال را بررسی کنیم:

```
static void semanticQuestions()
        {
            var tree = CSharpSyntaxTree.ParseText(@"
using System;

class Foo
{
    public static explicit operator DateTime(Foo f)
    {
        throw new NotImplementedException();
    }

    void Bar(int x)
    {
    }
}");

            var mscorlib = MetadataReference.CreateFromFile(typeof (object).Assembly.Location);
            var options = new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary);
            var comp =
CSharpCompilation.Create("Demo").AddSyntaxTrees(tree).AddReferences(mscorlib).WithOptions(options);
            // var res = comp.Emit("Demo.dll");

            // boxing
            var conv1 = comp.ClassifyConversion(
                comp.GetSpecialType(SpecialType.System_Int32),
                comp.GetSpecialType(SpecialType.System_Object)
                );

            // unboxing
            var conv2 = comp.ClassifyConversion(
                comp.GetSpecialType(SpecialType.System_Object),
                comp.GetSpecialType(SpecialType.System_Int32)
                );

            // explicit reference conversion
            var conv3 = comp.ClassifyConversion(
                comp.GetSpecialType(SpecialType.System_Object),
                comp.GetTypeByMetadataName("Foo")
                );

            // explicit user-supplied conversion
            var conv4 = comp.ClassifyConversion(
                comp.GetTypeByMetadataName("Foo"),
                comp.GetSpecialType(SpecialType.System_DateTime)
                );
        }
```

تا سطر CSharpCompilation.Create این مثال، مانند قبل است و تا اینجا به Compilation API دسترسی پیدا کرده‌ایم. پس از آن می‌خواهیم یک Semantic analysis مقدماتی را انجام دهیم. برای این منظور می‌توان از متد ClassifyConversion استفاده کرد. این متد یک نوع مبداء و یک نوع مقصد را دریافت می‌کند و بر اساس اطلاعاتی که از Compilation API بدست می‌آورد، می‌تواند مشخص کند که برای مثال آیا نوع کلاس Foo قابل تبدیل به DateTime هست یا خیر و اگر هست چه نوع تبدیلی را نیاز دارد؟



برای مثال نتیجه‌ی بررسی آخرین تبدیل انجام شده در تصویر فوق مشخص است. با توجه به تعریف `public static explicit operator DateTime` در سورس کد مورد آنالیز، این تبدیل `explicit` بوده و همچنین `user defined`. به علاوه متدی هم که این تبدیل را انجام می‌دهد، مشخص کرده‌است.

عنوان: **Roslyn #5**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۳۰ ۱۵:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Roslyn

بررسی Semantic Models

همانطور که از قسمت قبل به‌خاطر دارید، برای دسترسی به اطلاعات semantics، نیاز به یک context مناسب که همان Compilation API است، می‌باشد. این context دارای اطلاعاتی مانند دسترسی به تمام نوع‌های تعریف شده‌ی توسط کاربر و متادیتاهای ارجاعی، مانند کلاس‌های پایه‌ی دات نت فریم‌ورک است. بنابراین پس از ایجاد وهله‌ای از Compilation API، کار با فراخوانی متد GetSemanticModel آن ادامه می‌یابد. در ادامه با مثال‌هایی، کاربرد این متد را بررسی خواهیم کرد.

ساختار جدید Optional

خروجی‌های تعدادی از متدهای Roslyn با ساختار جدیدی به نام Optional ارائه می‌شوند:

```
public struct Optional<T>
    {
        public bool HasValue { get; }
        public T Value { get; }
    }
```

این ساختار که بسیار شبیه است به ساختار قدیمی Nullable<T>، منحصر به Value types نیست و Reference types را نیز شامل می‌شود و بیانگر این است که آیا یک Reference type، واقعا مقدار دهی شده‌است یا خیر؟

دریافت مقادیر ثابت Literals

فرض کنید می‌خواهیم مقدار ثابت int x = 42 ; را دریافت کنیم. برای اینکار ابتدا باید syntax tree آن تشکیل شود و سپس نیاز به یک سری حلقه و if و else و همچنین بررسی نال بودن بسیاری از موارد است تا به نود مقدار ثابت 42 برسیم. سپس متد GetConstantValue مربوط به GetSemanticModel را بر روی آن فراخوانی می‌کنیم تا به مقدار واقعی آن که ممکن است در اثر محاسبات جاری تغییر کرده باشد، برسیم.
اما روش بهتر و توصیه شده، استفاده از CSharpSyntaxWalker است که در انتهای قسمت سوم معرفی شد:

```
class ConsoleWriteLineWalker : CSharpSyntaxWalker
{
    public ConsoleWriteLineWalker()
    {
        Arguments = new List<ExpressionSyntax>();
    }

    public List<ExpressionSyntax> Arguments { get; }

    public override void VisitInvocationExpression(InvocationExpressionSyntax node)
    {
        var member = node.Expression as MemberAccessExpressionSyntax;
        var type = member?.Expression as IdentifierNameSyntax;
        if (type != null && type.Identifier.Text == "Console" && member.Name.Identifier.Text ==
"WriteLine")
        {
            if (node.ArgumentList.Arguments.Count == 1)
            {
                var arg = node.ArgumentList.Arguments.Single().Expression;
                Arguments.Add(arg);
                return;
            }
        }

        base.VisitInvocationExpression(node);
    }
}
```

اگر به کدهای ادامه‌ی بحث دقت کنید، قصد داریم مقادیر ثابت آرگومان‌های Console.WriteLine را استخراج کنیم. به همین جهت در این SyntaxWalker، نوع Console و متد WriteLine آن مورد بررسی قرار گرفته‌اند. اگر این نود دارای یک تک آرگومان بود، آین آرگومان استخراج شده و به لیست آرگومان‌های خروجی این کلاس اضافه می‌شود.
در ادامه نحوه‌ی استفاده‌ی از این SyntaxWalker را ملاحظه می‌کنید. در اینجا ابتدا سورس کدی حاوی یک سری Console.WriteLine که دارای تک آرگومان‌های ثابتی هستند، تبدیل به syntax tree می‌شود. سپس از روی آن CSharpCompilation تولید می‌گردد تا بتوان به اطلاعات semantics دسترسی یافت:

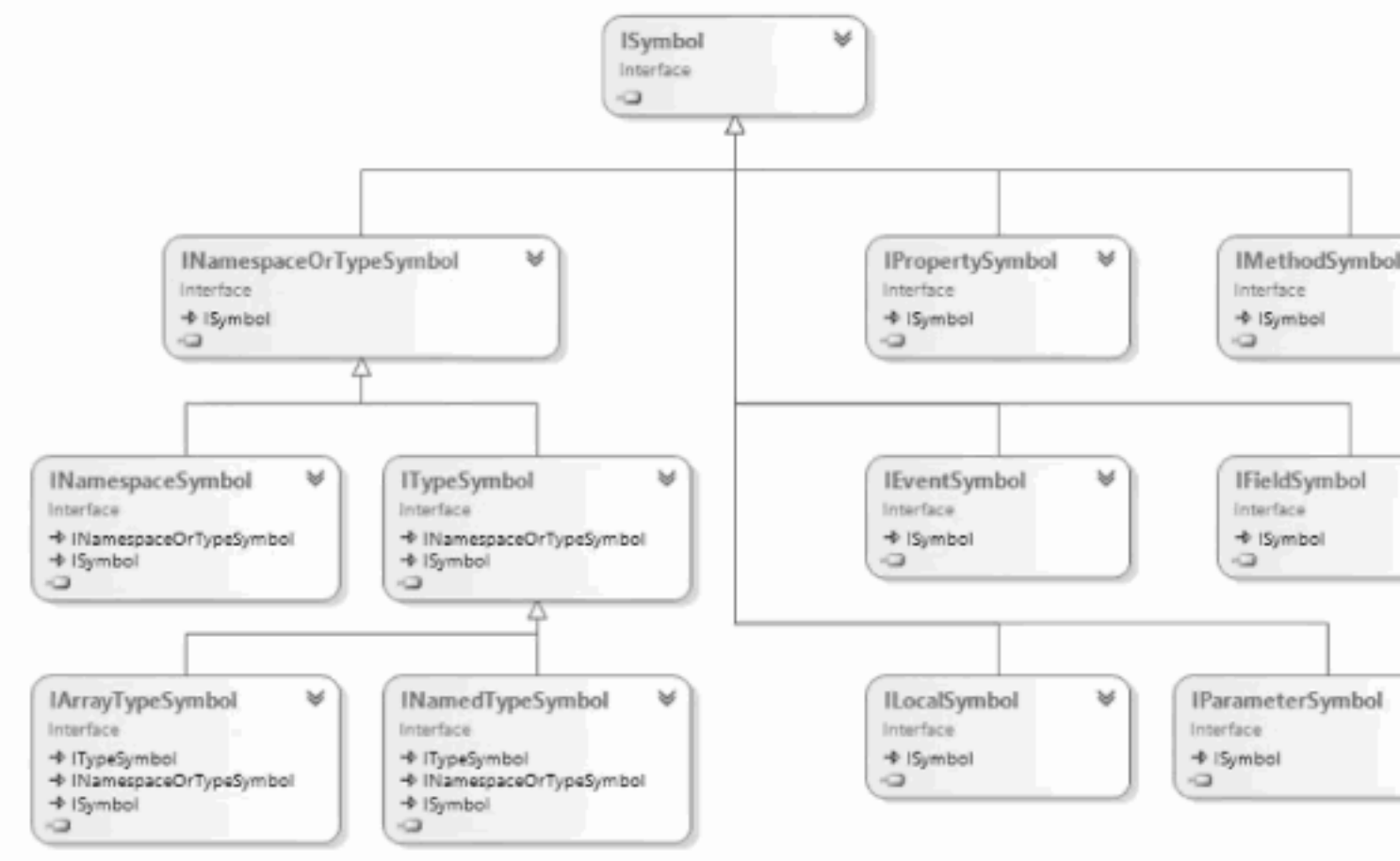
```
static void getConstantValue()
{
    // Get the syntax tree.
    var code = @"
                using System;

                class Foo
                {
                    void Bar(int x)
                    {
                        Console.WriteLine(3.14);
                        Console.WriteLine(""qux"");
                        Console.WriteLine('c');
                        Console.WriteLine(null);
                        Console.WriteLine(x * 2 + 1);
                    }
                }
                ";

    var tree = CSharpSyntaxTree.ParseText(code);
    var root = tree.GetRoot();

    // Get the semantic model from the compilation.
    var mscorlib = MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
    var comp = CSharpCompilation.Create("Demo").AddSyntaxTrees(tree).AddReferences(mscorlib);
    var model = comp.GetSemanticModel(tree);

    // Traverse the tree.
    var walker = new ConsoleWriteLineWalker();
    walker.Visit(root);


    // Analyze the constant argument (if any).
    foreach (var arg in walker.Arguments)
    {
        var val = model.GetConstantValue(arg);
        if (val.HasValue)
        {
            Console.WriteLine(arg + " has constant value " + (val.Value ?? "null") + " of type " +
(val.Value?.GetType() ?? typeof(object)));
        }
        else
        {
            Console.WriteLine(arg + " has no constant value");
        }
    }
}
```

در ادامه با استفاده از CSharpCompilation و متد GetSemanticModel آن به SemanticModel جاری دسترسی خواهیم یافت.
اکنون SyntaxWalker را وارد به حرکت بر روی ریشه‌ی syntax tree سورس کد آنالیز شده می‌کنیم. به این ترتیب لیست آرگومان‌های متدهای Console.WriteLine بدست می‌آیند. سپس با فراخوانی متد model.GetConstantValue بر روی هر آرگومان دریافتی، مقادیر آن‌ها با فرمت Optional<T> استخراج می‌شوند.
خروجی نمایش داده شده‌ی توسط برنامه به صورت ذیل است:

```
3.14 has constant value 3.14 of type System.Double
"qux" has constant value qux of type System.String
'c' has constant value c of type System.Char
null has constant value null of type System.Object
x * 2 + 1 has no constant value
```

درک مفهوم Symbols

اینترفیس ISymbol در Roslyn، ریشه‌ی تمام Symbolهای مختلف مدل سازی شده‌ی در آن است که تعدادی از آن‌ها را در تصویر ذیل مشاهده می‌کنید:



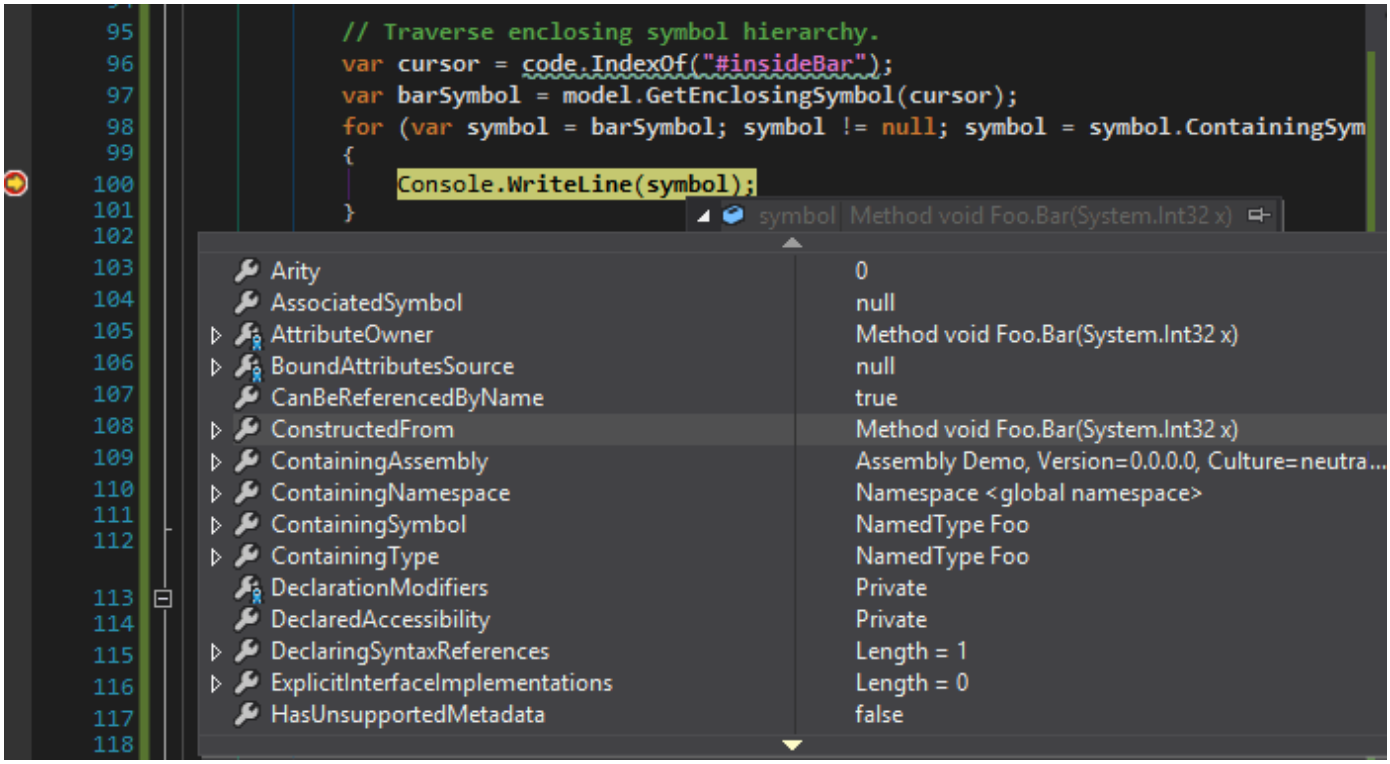
API کار با Symbols بسیار شبیه به API کار با Reflection است با این تفاوت که در زمان آنالیز کدها رخ می‌دهد و نه در زمان اجرای برنامه. همچنین در API Symbols امکان دسترسی به اطلاعاتی مانند labels, locals و امثال آن نیز وجود دارد که با استفاده از Reflection زمان اجرای برنامه قابل دسترسی نیستند. برای مثال فضاهای نام در Reflection صرفا به صورت رشته‌ای، با دات جدا شده از نوع‌های آنالیز شده‌ی توسط آن است؛ اما در اینجا مطابق تصویر فوق، یک اینترفیس مجزای خاص خود را دارد. جهت سهولت کار کردن با Symbols، الگوی Visitor با معرفی کلاس پایه‌ی SymbolVisitor نیز پیش بینی شده‌است.

```
static void workingWithSymbols()
{
    // Get the syntax tree.
    var code = @"
                using System;

                class Foo
                {
                    void Bar(int x)
                    {
                        // #insideBar
                    }
                }

                class Qux
                {
```

```
                    protected int Baz { get; set; }
                }
                ";

    var tree = CSharpSyntaxTree.ParseText(code);
    var root = tree.GetRoot();

    // Get the semantic model from the compilation.
    var mscorlib = MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
    var comp = CSharpCompilation.Create("Demo").AddSyntaxTrees(tree).AddReferences(mscorlib);
    var model = comp.GetSemanticModel(tree);

    // Traverse enclosing symbol hierarchy.
    var cursor = code.IndexOf("#insideBar");
    var barSymbol = model.GetEnclosingSymbol(cursor);
    for (var symbol = barSymbol; symbol != null; symbol = symbol.ContainingSymbol)
    {
        Console.WriteLine(symbol);
    }

    // Analyze accessibility of Baz inside Bar.
    var bazProp = ((CompilationUnitSyntax)root)
        .Members.OfType<ClassDeclarationSyntax>()
        .Single(m => m.Identifier.Text == "Qux")
        .Members.OfType<PropertyDeclarationSyntax>()
        .Single();
    var bazSymbol = model.GetDeclaredSymbol(bazProp);
    var canAccess = model.IsAccessible(cursor, bazSymbol);
}
```

یکی از کاربردهای مهم Symbols API دریافت اطلاعات Symbols نقطه‌ای خاص از کدها می‌باشد. برای مثال در محل اشاره‌گر ادیتور، چه Symbols ایی تعریف شده‌اند و از آن‌ها در مباحث ساخت افزونه‌های آنالیز کدها زیاد استفاده می‌شود. نمونه‌ای از آن‌را در قطعه کد فوق ملاحظه می‌کنید. در اینجا با استفاده از متد GetEnclosingSymbol، سعی در یافتن Symbols قرار گرفته‌ی در ناحیه‌ی #insideBar کدهای فوق داریم؛ با خروجی ذیل که نام demo.exe آن از نام CSharpCompilation آن گرفته شده‌است:

```
Foo.Bar(int)
Foo
<global namespace>
Demo.exe
Demo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
```



همچنین در ادامه‌ی کد، توسط متد IsAccessible قصد داریم بررسی کنیم آیا Symbol قرار گرفته در محل کرسر، دسترسی به خاصیت protected کلاس Qux را دارد یا خیر؟ که پاسخ آن خیر است.

آشنایی با Binding symbols

یکی از مراحل کامپایل کد، binding نام دارد و در این مرحله است که اطلاعات Symbolic هر نود از Syntax tree دریافت می‌شود. برای مثال در اینجا مشخص می‌شود که این x، آیا یک متغیر محلی است، یا یک فیلد و یا یک خاصیت؟
مثال ذیل بسیار شبیه است به مثال getConstantValue ابتدای بحث، با این تفاوت که در حلقه‌ی آخر کار از متد GetSymbolInfo استفاده شده‌است:

```
static void bindingSymbols()
{
    // Get the syntax tree.
    var code = @"
                using System;

                class Foo
                {
                    private int y;

                    void Bar(int x)
                    {
                        Console.WriteLine(x);
                        Console.WriteLine(y);

                        int z = 42;
                        Console.WriteLine(z);

                        Console.WriteLine(a);
                    }
                }";

    var tree = CSharpSyntaxTree.ParseText(code);
    var root = tree.GetRoot();

    // Get the semantic model from the compilation.
    var mscorlib = MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
    var comp = CSharpCompilation.Create("Demo").AddSyntaxTrees(tree).AddReferences(mscorlib);
```

```
    var model = comp.GetSemanticModel(tree);

    // Traverse the tree.
    var walker = new ConsoleWriteLineWalker();
    walker.Visit(root);

    // Bind the arguments.
    foreach (var arg in walker.Arguments)
    {
        var symbol = model.GetSymbolInfo(arg);
        if (symbol.Symbol != null)
        {
            Console.WriteLine(arg + " is bound to " + symbol.Symbol + " of type " +
symbol.Symbol.Kind);
        }
        else
        {
            Console.WriteLine(arg + " could not be bound");
        }
    }
}
```

با این خروجی:

```
x is bound to int of type Parameter
y is bound to Foo.y of type Field
z is bound to z of type Local
a could not be bound
```

در مثال فوق، با استفاده از Syntax Walker طراحی شده در ابتدای بحث که کار استخراج آرگومان‌های متدهای Console.WriteLine را انجام می‌دهد، قصد داریم بررسی کنیم، هر آرگومان به چه Symbol ایی بایند شده‌است و نوعش چیست؟ برای مثال Console.WriteLine اول که از پارامتر x استفاده می‌کند، نوع x مورد استفاده‌اش چیست؟ آیا فیلد است، متغیر محلی است یا یک پارامتر؟ این اطلاعات را با استفاده از متد model.GetSymbolInfo می‌توان استخراج کرد.

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.

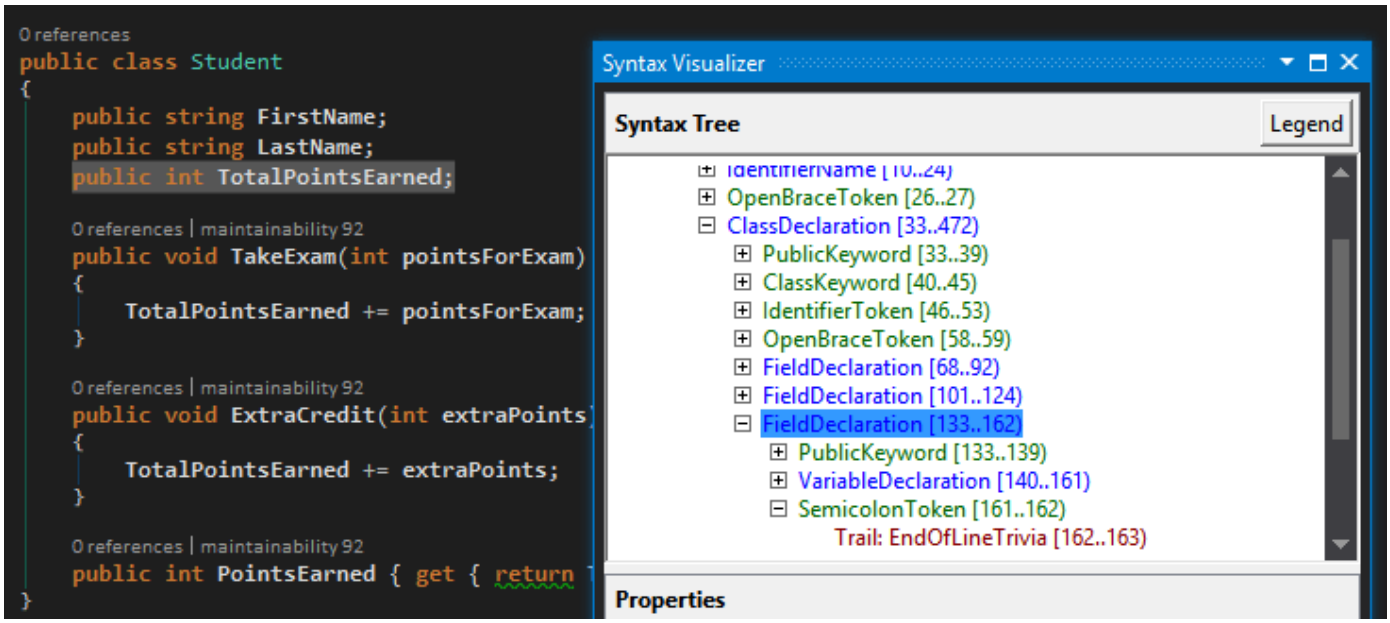
بررسی مثال معتبری که می‌تواند بهتر باشد

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.

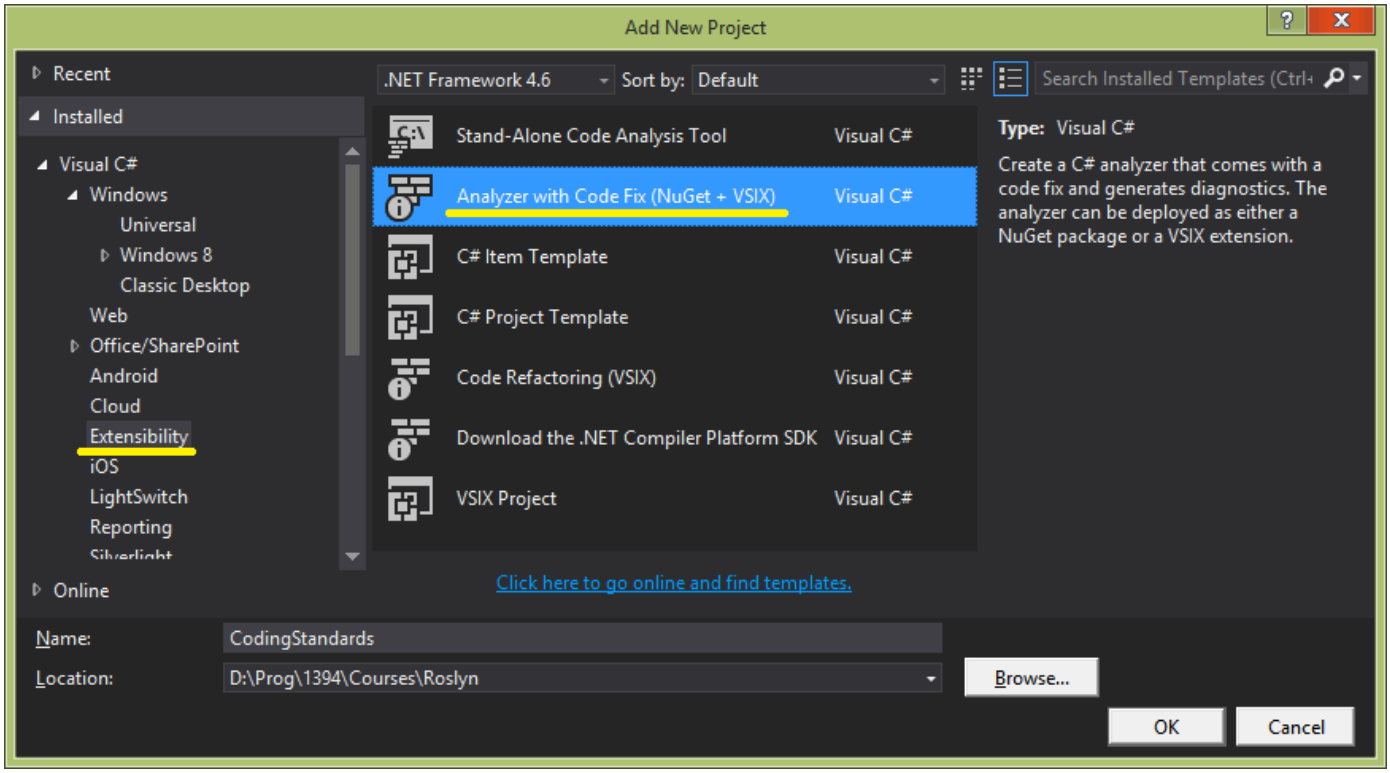
بنابراین در ادامه هدف ما این است که یک Roslyn Analyzer جدید را طراحی کنیم تا از طریق آن هشدارهایی را جهت تبدیل فیلدهای عمومی به خصوصی، به برنامه نویس نمایش دهیم.



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.

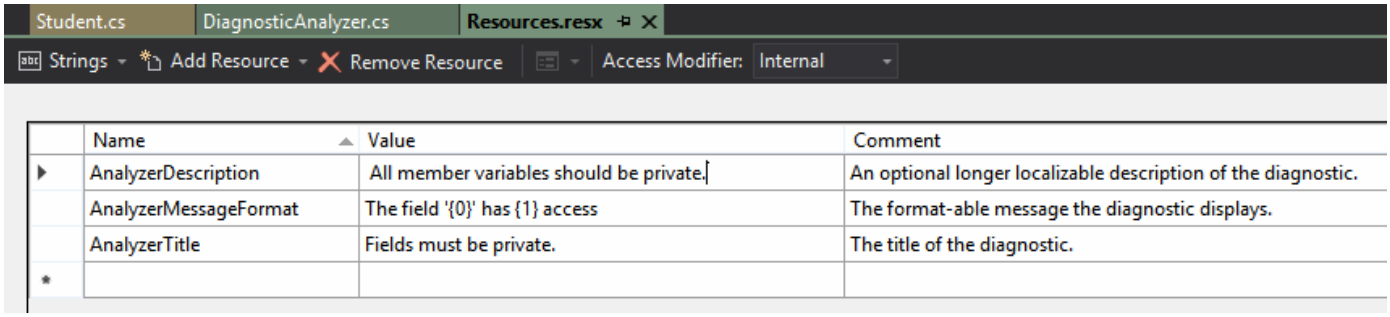
در ادامه کدهای فایل DiagnosticAnalyzer.cs را به صورت ذیل تغییر دهید:

```
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.

```
namespace StudentLibrary
{
    0 references
    public class Student
    {
        public string FirstName;
        public strin  stName;
        public int T
                                    Make Private
        0 references
        public void TakeExam(int pointsForE
        {
            TotalPointsEarned += pointsForE
        }

        0 references
        public void ExtraCredit(int extraPo
        {
            TotalPointsEarned += extraPoints;
        }

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

CodingStandards The field 'FirstName' has public access

```
...
{
    public string FirstName;
    private string FirstName;
    public string LastName;
...
```

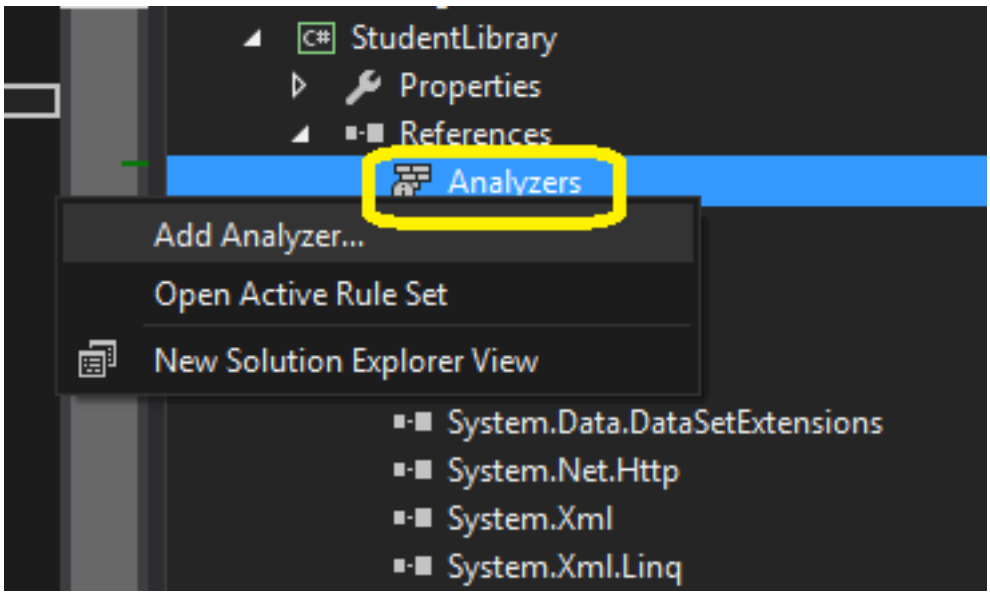
Preview changes

Fix all occurrences in: Document | Project | Solution

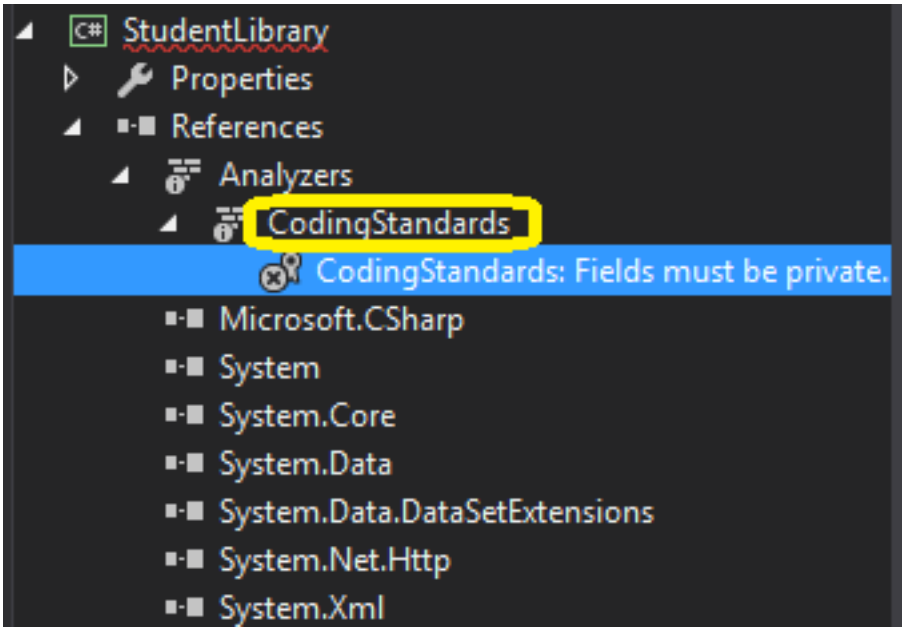
نتیجه‌ی اعمال این افزونه‌ی جدید را در تصویر فوق ملاحظه می‌کنید. زیر سطرهای دارای فیلد عمومی، خط قرمز کشیده شده‌است (به علت تعریف DiagnosticSeverity.Error). همچنین حالت فعلی و حالت برطرف شده را نیز با رنگ‌های قرمز و سبز می‌توان مشاهده کرد. کلیک بر روی گزینه‌ی make private، سبب اصلاح خودکار آن سطر می‌گردد.

روش دوم آزمایش یک Roslyn Analyzer

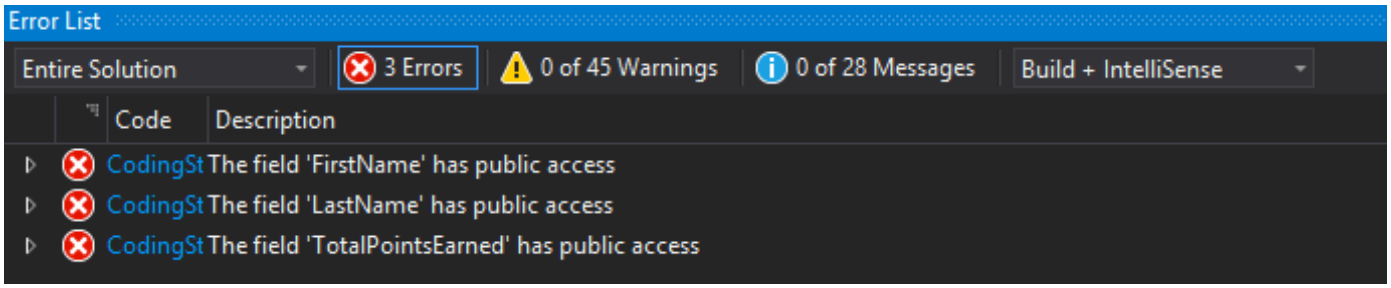
همانطور که از انتهای بحث قسمت دوم به‌خاطر دارید، این آنالایزرها را می‌توان به کامپایلر نیز معرفی کرد. روش انجام اینکار در ویژوال استودیوی 2015 در تصویر ذیل نمایش داده شده‌است.



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).

عنوان: **Roslyn #7**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۷/۰۱ ۱۵:۲۵
آدرس: www.dotnettips.info
گروه‌ها: Roslyn

معرفی Workspace API

Workspace، در حقیقت نمایش اجزای یک Solution در ویژوال استودیو است و یک Solution متشکل است از تعدادی پروژه به همراه وابستگی‌های بین آن‌ها. هدف از وجود Workspace API در Roslyn، دسترسی به اطلاعات لازم جهت انجام امور Refactoring در سطح یک Solution است. برای مثال اگر قرار است نام خاصیتی تغییر کند و این خاصیت در چندین پروژه‌ی دیگر در حال استفاده است، این نام باید در سراسر Solution جاری یافت شده و تغییر یابد. همچنین برفراز Workspace API تعدادی سرویس زبان مانند فرمت کننده‌های کدها، تغییرنام دهنده‌های سیمبل‌ها و توصیه کننده‌ها نیز تهیه شده‌اند.
همچنین این سرویس‌ها و API تهیه شده، منحصر به ویژوال استودیو نیستند و VS 2015 تنها از آن‌ها استفاده می‌کند. برای مثال نگارش‌های جدیدتر mono-develop لینوکسی نیز شروع به استفاده‌ی از Roslyn کرده‌اند.

نمایش اجزای یک Solution

در ادامه مثالی را مشاهده می‌کنید که توسط آن نام Solution و سپس تمام پروژه‌های موجود در آن‌ها به همراه نام فایل‌های مرتبط و همچنین ارجاعات آن‌ها در صفحه نمایش داده می‌شوند:

```
var ws = MSBuildWorkspace.Create();
var sln = ws.OpenSolutionAsync(@"..\..\..\Roslyn.sln").Result;


// Print the root of the solution.
Console.WriteLine(Path.GetFileName(sln.FilePath));


// Get dependency graph to perform a sort.
var g = sln.GetProjectDependencyGraph();
var ps = g.GetTopologicallySortedProjects();


// Print all projects, their documents, and references.
foreach (var p in ps)
{
    var proj = sln.GetProject(p);

    Console.WriteLine("> " + proj.Name);

    Console.WriteLine("  > References");
    foreach (var r in proj.ProjectReferences)
    {
        Console.WriteLine("    - " + sln.GetProject(r.ProjectId).Name);
    }

    foreach (var d in proj.Documents)
    {
        Console.WriteLine("  - " + d.Name);
    }
}
```

در ابتدا نیاز است یک وهله از MSBuildWorkspace را ایجاد کرد. اکنون با استفاده از این Workspace می‌توان solution خاصی را گشود و آنالیز کرد. قسمتی از خروجی آن چنین شکلی را دارد:

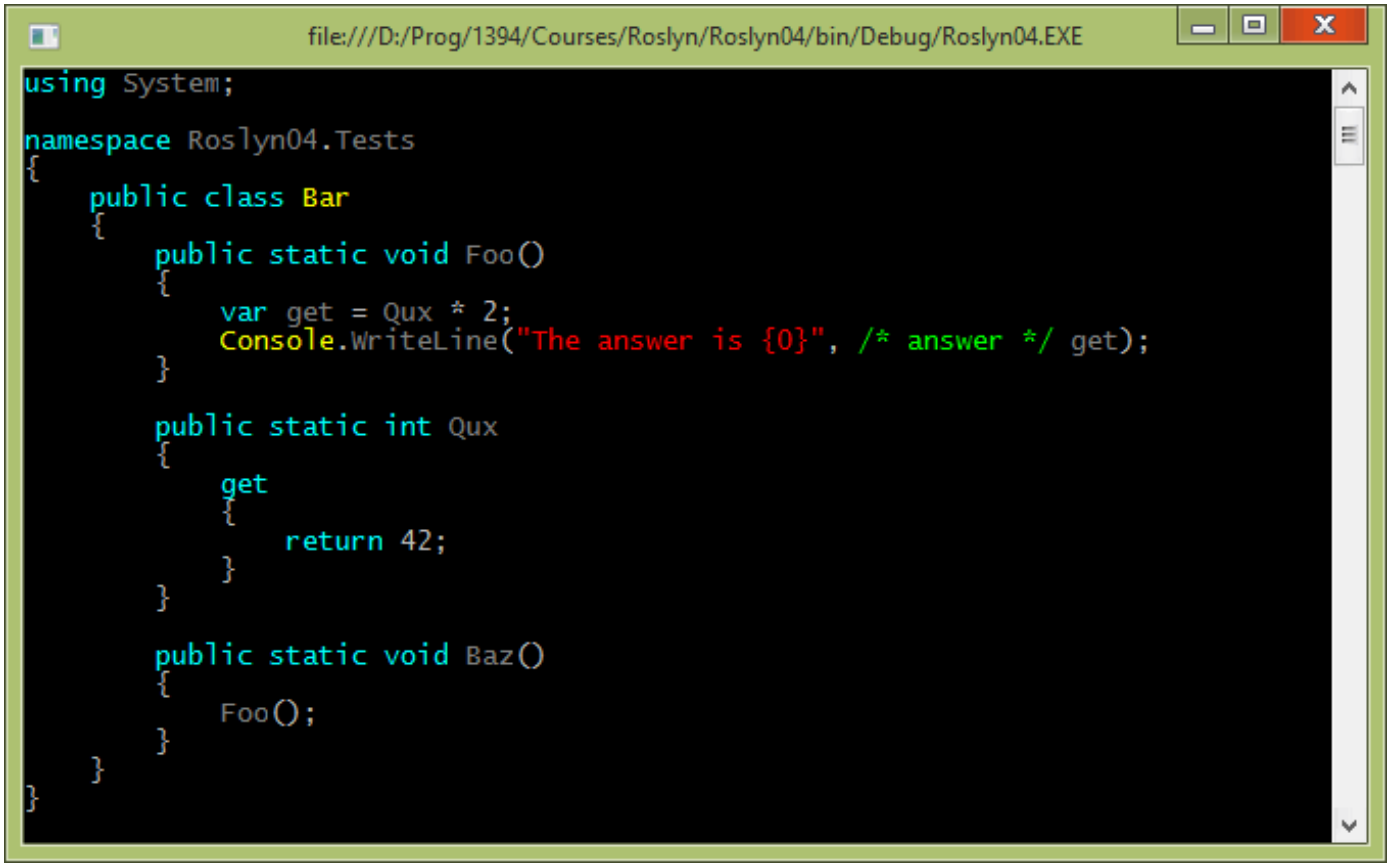
```
Roslyn.sln
> Roslyn01
  > References
  - Program.cs
  - AssemblyInfo.cs
  - .NETFramework,Version=v4.6.AssemblyAttributes.cs
```

ایجاد یک Syntax highlighter با استفاده از Classification service

هدف از Classification service، رندر کردن فایل‌ها در ادیتور جاری است. برای این منظور نیاز است بتوان واژه‌های کلیدی، کامنت‌ها، نام‌های نوع‌ها و امثال آن‌ها را به صورت کلاسه شده در اختیار داشت و سپس برای مثال هرکدام را با رنگی مجزا نمایش داد و رندر کرد.
در ادامه مثالی از آن‌را ملاحظه می‌کنید:

```
var ws = MSBuildWorkspace.Create();
var sln = ws.OpenSolutionAsync(@"..\..\..\Roslyn.sln").Result;

// Get the Tests\Bar.cs document.
var proj = sln.Projects.Single(p => p.Name == "Roslyn04.Tests");
var test = proj.Documents.Single(d => d.Name == "Bar.cs");

var tree = test.GetSyntaxTreeAsync().Result;
var root = tree.GetRootAsync().Result;

// Get all the spans in the document that are classified as language elements.
var spans = Classifier.GetClassifiedSpansAsync(test, root.FullSpan).Result.ToDictionary(c =>
c.TextSpan.Start, c => c);

// Print the source text with appropriate colorization.
var txt = tree.GetText().ToString();

var i = 0;
foreach (var c in txt)
{
    var span = default(ClassifiedSpan);
    if (spans.TryGetValue(i, out span))
    {
        var color = ConsoleColor.Gray;

        switch (span.ClassificationType)
        {
            case ClassificationTypeNames.Keyword:
                color = ConsoleColor.Cyan;
                break;
            case ClassificationTypeNames.StringLiteral:
            case ClassificationTypeNames.VerbatimStringLiteral:
                color = ConsoleColor.Red;
                break;
            case ClassificationTypeNames.Comment:
                color = ConsoleColor.Green;
                break;
            case ClassificationTypeNames.ClassName:
            case ClassificationTypeNames.InterfaceName:
            case ClassificationTypeNames.StructName:
            case ClassificationTypeNames.EnumName:
            case ClassificationTypeNames.TypeParameterName:
            case ClassificationTypeNames.DelegateName:
                color = ConsoleColor.Yellow;
                break;
            case ClassificationTypeNames.Identifier:
                color = ConsoleColor.DarkGray;
                break;
        }

        Console.ForegroundColor = color;
    }

    Console.Write(c);

    i++;
}
```

با این خروجی:



توضیحات:

در اینجا نیز کار با ایجاد یک Workspace و سپس گشودن Solution ایی مشخص در آن آغاز می‌شود. سپس در آن به دنبال پروژه‌ای به نام Roslyn04.Tests می‌گردیم. این پروژه حاوی تعدادی کلاس، جهت بررسی و آزمایش هستند. برای مثال در اینجا فایل Bar.cs آن قرار است آنالیز شود. پس از یافتن آن، ابتدا syntax tree آن دریافت می‌گردد و سپس به سرویس Classifier.GetClassifiedSpansAsync ارسال خواهد شد. خروجی آن شامل لیستی از Classified Spans است؛ مانند کلمات کلیدی، رشته‌ها، کامنت‌ها و غیره. در ادامه این لیست تبدیل به یک دیکشنری می‌شود که کلید آن محل آغاز این span و مقدار آن، مقدار span است. سپس متن syntax tree دریافت شده و حرف به حرف آن در طی یک حلقه بررسی می‌شود. در این حلقه، مقدار i به محل حروف جاری مورد آنالیز اشاره می‌کند. اگر این محل در دیکشنری Classified Spans وجود داشت، یعنی یک span جدید شروع شده‌است و بر این اساس، نوع آن span را می‌توان استخراج کرد و سپس بر اساس این نوع، رنگ متفاوتی را در صفحه نمایش داد.

سرویس فرمت کردن کدها

این سرویس کار فرمت خودکار کدهای بهم ریخته را انجام می‌دهد؛ مانند تنظیم فاصله‌های خالی و یا ایجاد indentation و امثال آن. در حقیقت Ctlr K+D در ویژوال استودیو، دقیقا از همین سرویس زبان استفاده می‌کند.

کار کردن با این سرویس از طریق برنامه نویسی به نحو ذیل است:

```
var ws = MSBuildWorkspace.Create();
var sln = ws.OpenSolutionAsync(@"..\..\..\Roslyn.sln").Result;


// Get the Tests\Qux.cs document.
var proj = sln.Projects.Single(p => p.Name == "Roslyn04.Tests");
var qux = proj.Documents.Single(d => d.Name == "Qux.cs");

Console.WriteLine("Before:");
Console.WriteLine();
Console.WriteLine(qux.GetSyntaxTreeAsync().Result.GetText());
```

```
Console.WriteLine();
Console.WriteLine();


// Apply formatting and print the result.
var res = Formatter.FormatAsync(qux).Result;

Console.WriteLine("After:");
Console.WriteLine();
Console.WriteLine(res.GetSyntaxTreeAsync().Result.GetText());
Console.WriteLine();
```

با این خروجی:

```
Before:

using System;

namespace Roslyn04.Tests
{
    class Qux {
        public void Baz()
        { Console.WriteLine(42);
            return;  }
    }
}


After:

using System;

namespace Roslyn04.Tests
{
    class Qux
    {
        public void Baz()
        {
            Console.WriteLine(42);
            return;
        }
    }
}
```

همانطور که ملاحظه می‌کنید، فایل Qux.cs که فرمت مناسبی ندارد. بنابراین باز شده و syntax tree آن به سرویس Formatter.FormatAsync جهت فرمت شدن ارسال می‌شود.

سرویس یافتن سیمبل‌ها

یکی دیگر از قابلیت‌هایی که در ویژوال استودیو وجود دارد، امکان یافتن سیمبل‌ها است. برای مثال این نوع یا کلاس خاص، در کجاها استفاده شده‌است و به آن ارجاعاتی وجود دارد. مواردی مانند Find all references، Go to definition و نمایش Call hierarchy از این سرویس استفاده می‌کنند.

```
var ws = MSBuildWorkspace.Create();
var sln = ws.OpenSolutionAsync(@"..\..\..\Roslyn.sln").Result;


// Get the Tests project.
var proj = sln.Projects.Single(p => p.Name == "Roslyn04.Tests");

// Locate the symbol for the Bar.Foo method and the Bar.Qux property.
var comp = proj.GetCompilationAsync().Result;

var barType = comp.GetTypeByMetadataName("Roslyn04.Tests.Bar");

var fooMethod = barType.GetMembers().Single(m => m.Name == "Foo");
var quxProp = barType.GetMembers().Single(m => m.Name == "Qux");
```

```
// Find callers across the solution.
Console.WriteLine("Find callers of Foo");
Console.WriteLine();

var callers = SymbolFinder.FindCallersAsync(fooMethod, sln).Result;
foreach (var caller in callers)
{
    Console.WriteLine(caller.CallingSymbol);
    foreach (var location in caller.Locations)
    {
        Console.WriteLine("    " + location);
    }
}

Console.WriteLine();
Console.WriteLine();

// Find all references across the solution.
Console.WriteLine("Find all references to Qux");
Console.WriteLine();

var references = SymbolFinder.FindReferencesAsync(quxProp, sln).Result;
foreach (var reference in references)
{
    Console.WriteLine(reference.Definition);
    foreach (var location in reference.Locations)
    {
        Console.WriteLine("    " + location.Location);
    }
}
```

در این مثال، پروژه‌ی Roslyn04.Tests که حاوی کلاس‌های Foo و Qux است، جهت آنالیز باز شده‌است. در اینجا برای رسیدن به Symbols نیاز است ابتدا به Compilation API دسترسی یافت و سپس متادیتاها را بر اساس آن استخراج کرد. سپس متدهای Foo و خاصیت Qux آن یافت شده‌اند.

اکنون با استفاده از سرویس SymbolFinder.FindCallersAsync تمام فراخوان‌های متد Foo را در سراسر Solution جاری می‌یابیم.

سپس با استفاده از سرویس SymbolFinder.FindReferencesAsync تمام ارجاعات به خاصیت Qux را در Solution جاری نمایش می‌دهیم.

سرویس توصیه کننده

Intellisense در ویژوال استودیو از سرویس توصیه کننده‌ی Roslyn استفاده می‌کند.

```
var ws = MSBuildWorkspace.Create();
var sln = ws.OpenSolutionAsync(@"..\..\..\Roslyn.sln").Result;

// Get the Tests\Foo.cs document.
var proj = sln.Projects.Single(p => p.Name == "Roslyn04.Tests");
var foo = proj.Documents.Single(d => d.Name == "Foo.cs");


// Find the 'dot' token in the first Console.WriteLine member access expression.
var tree = foo.GetSyntaxTreeAsync().Result;
var model = proj.GetCompilationAsync().Result.GetSemanticModel(tree);
var consoleDot =
tree.GetRoot().DescendantNodes().OfType<MemberAccessExpressionSyntax>().First().OperatorToken;


// Get recommendations at the indicated cursor position.
//
//   Console.WriteLine
//           ^
var res = Recommender.GetRecommendedSymbolsAtPosition(

                    model, consoleDot.GetLocation().SourceSpan.Start + 1, ws).ToList();

foreach (var rec in res)
{
    Console.WriteLine(rec);
}
```

در این مثال سعی شده‌است لیست توصیه‌های ارائه شده در حین تایپ دات، توسط سرویس Recommender.GetRecommendedSymbolsAtPosition دریافت و نمایش داده شوند. در ابتدای کار، کلاس Foo گشوده شده و سپس Syntax tree و Semantic model آن استخراج می‌شود. این model پارامتر اول متد سرویس توصیه کننده‌است. سپس نیاز است محل مکانی را به آن معرفی کنیم تا کار توصیه کردن را بر اساس آن شروع کند. برای نمونه در اینجا OperatorToken در حقیقت همان دات مربوط به Console.WriteLine است. پس از یافتن این توکن، امکان دسترسی به مکان آن وجود دارد.
تعدادی از خروجی‌های مثال فوق به صورت زیر هستند:

```
System.Console.Beep()
System.Console.Beep(int, int)
System.Console.Clear()
```

سرویس تغییر نام دادن

هدف از سرویس Renamer.RenameSymbolAsync، تغییر نام یک identifier در کل Solution است. نمونه‌ای از نحوه‌ی کاربرد آن‌را در مثال ذیل مشاهده می‌کنید:

```
var ws = MSBuildWorkspace.Create();
var sln = ws.OpenSolutionAsync(@"..\..\..\Roslyn.sln").Result;


// Get Tests\Bar.cs before making changes.
var oldProj = sln.Projects.Single(p => p.Name == "Roslyn04.Tests");
var oldDoc = oldProj.Documents.Single(d => d.Name == "Bar.cs");

Console.WriteLine("Before:");
Console.WriteLine();

var oldTxt = oldDoc.GetTextAsync().Result;
Console.WriteLine(oldTxt);

Console.WriteLine();
Console.WriteLine();


// Get the symbol for the Bar.Foo method.
var comp = oldProj.GetCompilationAsync().Result;

var barType = comp.GetTypeByMetadataName("Roslyn04.Tests.Bar");
var fooMethod = barType.GetMembers().Single(m => m.Name == "Foo");


// Perform the rename.
var newSln = Renamer.RenameSymbolAsync(sln, fooMethod, "Foo2", ws.Options).Result;


// Get Tests\Bar.cs after making changes.
var newProj = newSln.Projects.Single(p => p.Name == "Roslyn04.Tests");
var newDoc = newProj.Documents.Single(d => d.Name == "Bar.cs");

Console.WriteLine("After:");
Console.WriteLine();

var newTxt = newDoc.GetTextAsync().Result;
Console.WriteLine(newTxt);
```

در این مثال، متد Foo کلاس Bar، قرار است به Foo2 تغییرنام یابد. به همین منظور ابتدا پروژه‌ی حاوی فایل Bar.cs باز شده و اطلاعات این کلاس استخراج می‌گردد. سپس اصل این کلاس تغییر نیافته نمایش داده می‌شود. در ادامه با استفاده از API کامپایل، به متادیتای متد Foo یا به عبارتی Symbol آن دسترسی پیدا می‌کنیم. سپس این Symbol به متد یا سرویس Renamer.RenameSymbolAsync ارسال می‌شود تا کار تغییر نام صورت گیرد. پس از اینکار مجددا متن کلاس تغییر یافته نمایش داده خواهد شد.

سرویس ساده کننده

هدف از سرویس ساده کننده، ساده‌کردن و کاهش کدهای ارائه شده، از دید Semantics است. برای مثال اگر فضای نامی در قسمت using ذکر شده‌است، دیگر نیازی نیست تا این فضای نام به ابتدای فراخوانی یک متد آن اضافه شود و می‌توان این قطعه از کد را ساده‌تر کرد و کاهش داد.

```
var ws = MSBuildWorkspace.Create();
var sln = ws.OpenSolutionAsync(@"..\..\..\Roslyn.sln").Result;


// Get the Tests\Baz.cs document.
var proj = sln.Projects.Single(p => p.Name == "Roslyn04.Tests");
var baz = proj.Documents.Single(d => d.Name == "Baz.cs");

Console.WriteLine("Before:");
Console.WriteLine();
Console.WriteLine(baz.GetSyntaxTreeAsync().Result.GetText());

Console.WriteLine();
Console.WriteLine();

var oldRoot = baz.GetSyntaxRootAsync().Result;


var memberAccesses = oldRoot.DescendantNodes().OfType<CastExpressionSyntax>();
var newRoot = oldRoot.ReplaceNodes(memberAccesses, (_, m) =>
m.WithAdditionalAnnotations(Simplifier.Annotation));

var newDoc = baz.WithSyntaxRoot(newRoot);


// Invoke the simplifier and print the result.
var res = Simplifier.ReduceAsync(newDoc).Result;

Console.WriteLine("After:");
Console.WriteLine();
Console.WriteLine(res.GetSyntaxTreeAsync().Result.GetText());
Console.WriteLine();
```

در این مثال نحوه‌ی ساده سازی castهای اضافی را ملاحظه می‌کنید. برای مثال اگر نوع متغیری int است، دیگر نیازی نیست در سراسر کد در کنار این متغیر، cast به int را هم ذکر کرد و می‌توان این کد را ساده‌تر نمود.

کدهای کامل این سری را از اینجا می‌توانید دریافت کنید:

Roslyn-Samples.zip