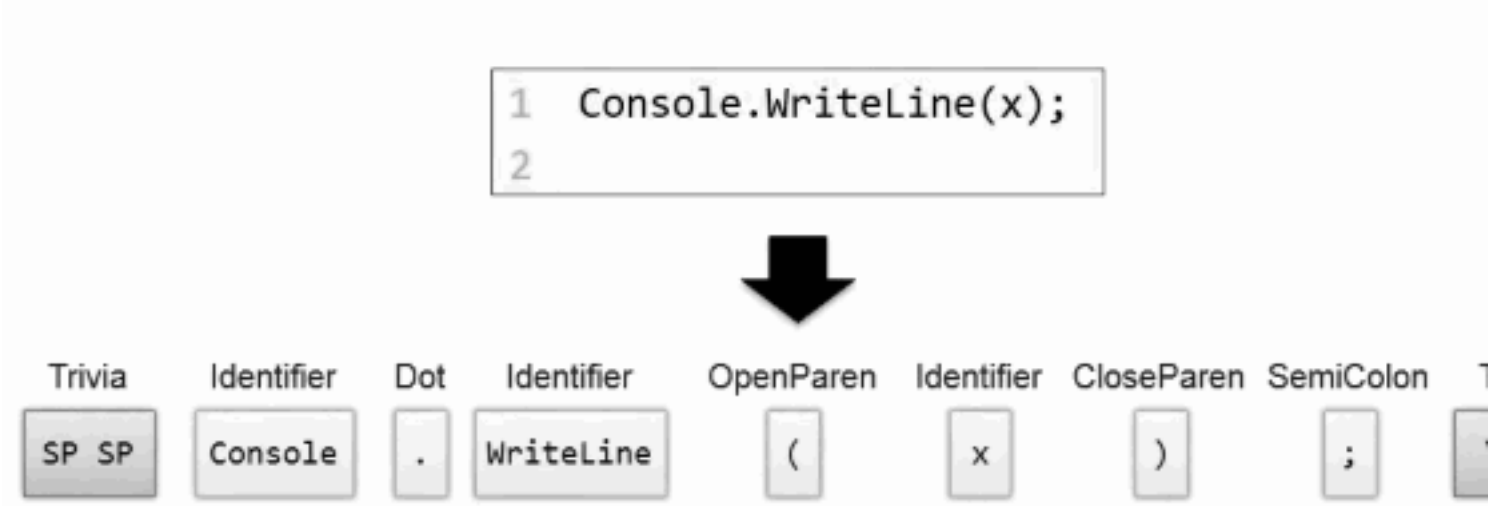
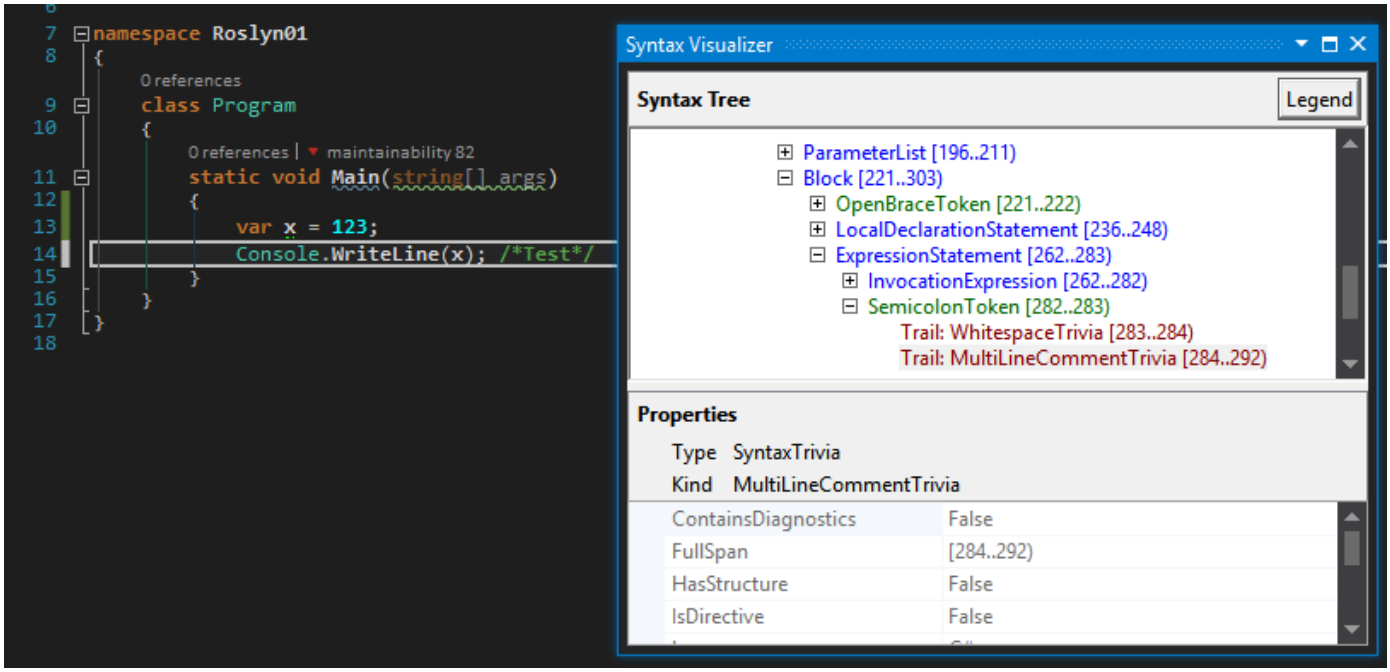
عنوان: **Roslyn #3**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۲۸ ۱۶:۲۰
آدرس: www.dotnettips.info
گروه‌ها: Roslyn

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.



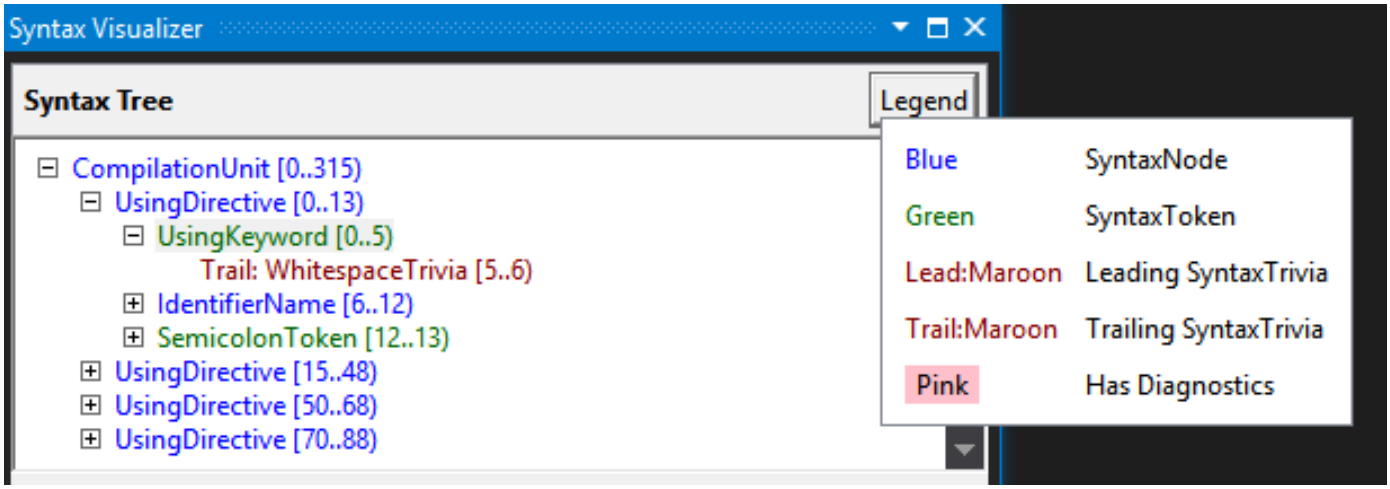
در تصویر فوق، مراحل تبدیل یک قطعه کد C# را به مجموعه‌ای از tokenهای معادل آن مشاهده می‌کنید. علاوه بر این‌ها، Roslyn syntax tree شامل موارد ویژه‌ای به نام Trivia نیز هست. برای مثال در حین نوشتن کدها، در ابتدای سطرها تعدادی space یا tab وجود دارند و یا در این بین ممکن است کامنتی نوشته شود. هرچند این موارد 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.



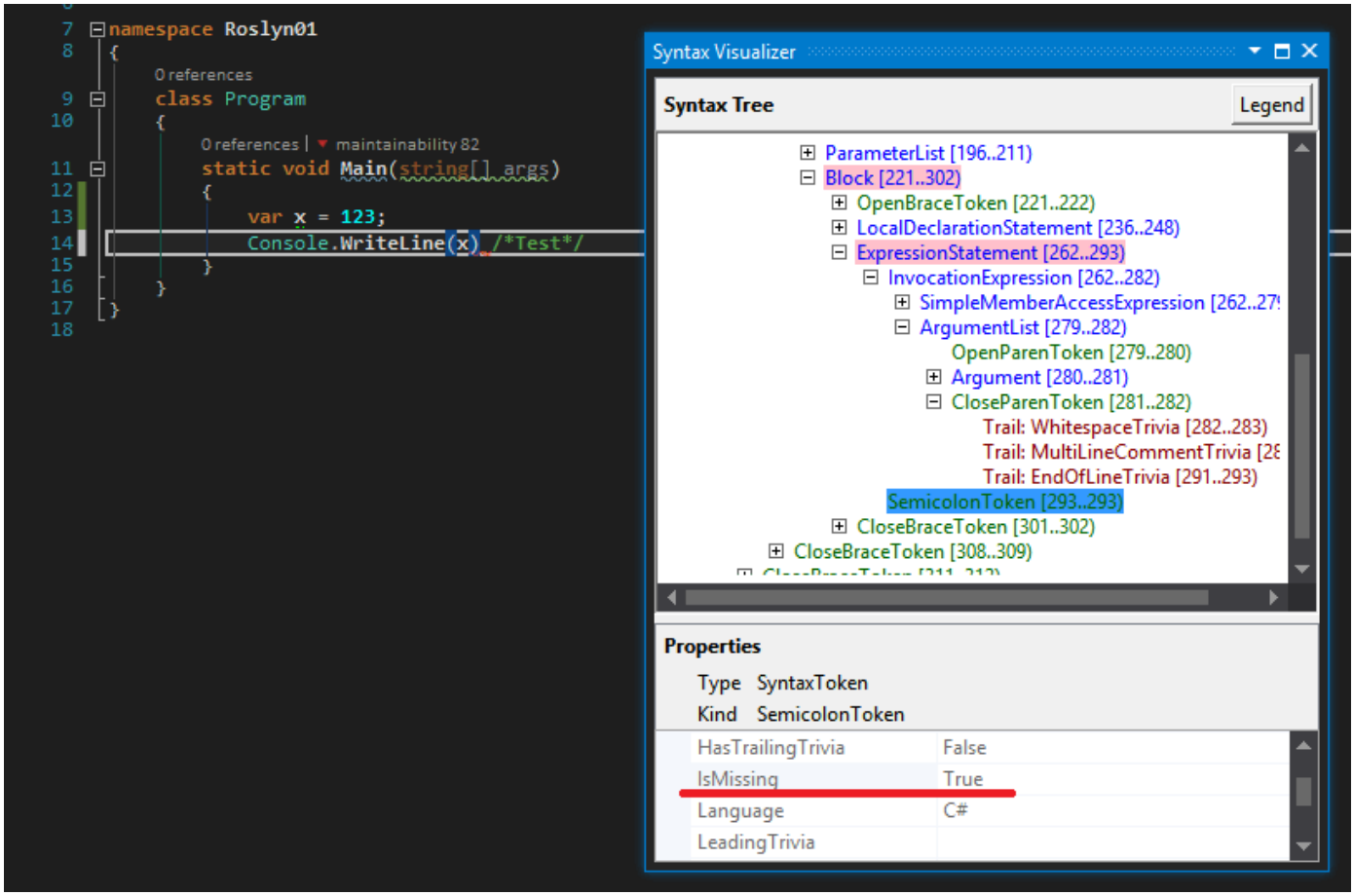
مفاهیم این رنگ‌ها را با کلیک بر روی دکمه‌ی Legend هم می‌توان مشاهده کرد.

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.