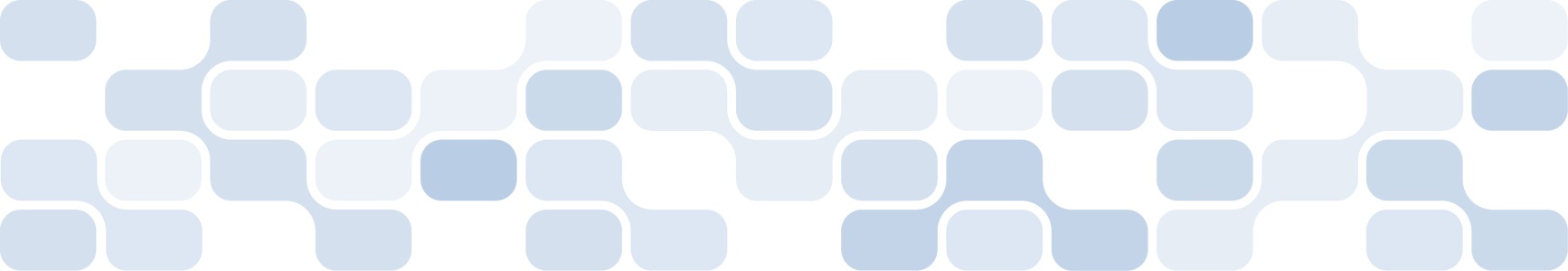
**The Roslyn Project**

**Exposing the C# and VB compiler’s**

**code analysis**

**September 2012**

MS_logo_KKaren Ng, Principal Lead Program Manager, Microsoft Corporation

Matt Warren, Principal Architect, Microsoft Corporation

Peter Golde, Partner Architect, Microsoft Corporation  
Anders Hejlsberg, Technical Fellow, Microsoft Corporation

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

 2011 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, Windows, Windows Server, Windows Vista, Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are property of their respective owners.

Contents

[1 Introduction 2](#_Toc306201201)

[2 Exposing the Core APIs 2](#_Toc306201202)

[2.1 Key Concepts 4](#_Toc306201203)

[2.2 API Layers 5](#_Toc306201204)

[2.2.1 Compiler APIs 5](#_Toc306201205)

[2.2.2 Scripting APIs 5](#_Toc306201206)

[2.2.3 Workspace APIs 5](#_Toc306201207)

[2.2.4 Services APIs 5](#_Toc306201208)

[3 Working with Syntax 6](#_Toc306201209)

[3.1 Key Concepts 6](#_Toc306201210)

[3.1.1 Syntax Trees 6](#_Toc306201211)

[3.1.2 Syntax Nodes 7](#_Toc306201212)

[3.1.3 Syntax Tokens 7](#_Toc306201213)

[3.1.4 Syntax Trivia 8](#_Toc306201214)

[3.1.5 Spans 8](#_Toc306201215)

[3.1.6 Kinds 9](#_Toc306201216)

[3.1.7 Errors 9](#_Toc306201217)

[3.2 Obtaining a Syntax Tree 9](#_Toc306201218)

[3.3 Manipulating Syntax Nodes 10](#_Toc306201219)

[3.3.1 Creating a node 10](#_Toc306201220)

[3.3.2 Modifying an existing node 11](#_Toc306201221)

[3.3.3 Replacing a node 12](#_Toc306201222)

[3.3.4 Creating a Syntax Tree 13](#_Toc306201223)

[4 Working with Semantics 13](#_Toc306201224)

[4.1 Key Concepts 13](#_Toc306201225)

[4.1.1 Compilation 14](#_Toc306201226)

[4.1.2 Symbols 14](#_Toc306201227)

[4.1.3 Semantic Model 15](#_Toc306201228)

[4.2 Obtaining a Compilation 15](#_Toc306201229)

[4.3 Obtaining Symbols 15](#_Toc306201230)

[4.4 Asking Semantic Questions 16](#_Toc306201231)

[4.5 Working with Fault Tolerance 18](#_Toc306201232)

[4.6 Using the Control and Data Flow Analysis APIs 18](#_Toc306201233)

[5 Working with a Workspace 19](#_Toc306201234)

[5.1 Key Concepts 20](#_Toc306201235)

[5.1.1 Workspace 20](#_Toc306201236)

[5.1.2 Solutions, Projects, Documents 20](#_Toc306201237)

[5.2 Obtaining a Workspace 21](#_Toc306201238)

[5.3 Working with a Solution 22](#_Toc306201239)

[5.4 Updating a Workspace 23](#_Toc306201240)

[6 Working with Services 23](#_Toc306201241)

[6.1 Key Concepts 24](#_Toc306201242)

[6.1.1 Managed Extensibility Framework 24](#_Toc306201243)

[6.2 Creating an IDE extension 24](#_Toc306201244)

[7 Summary 25](#_Toc306201245)

# Introduction

Traditionally, compilers are black boxes -- source code goes in one end, magic happens in the middle, and object files or assemblies come out the other end. As compilers perform their magic, they build up deep understanding of the code they are processing, but that knowledge is unavailable to anyone but the compiler implementation wizards. The information is promptly forgotten after the translated output is produced.

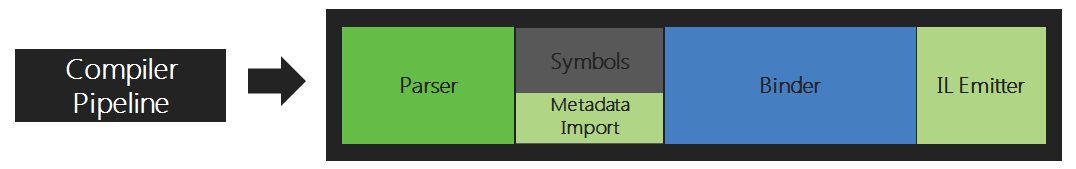
For decades, this world view has served us well, but it is no longer sufficient. Increasingly we rely on integrated development environment (IDE) features such as IntelliSense, refactoring, intelligent rename, “Find all references,” and “Go to definition” to increase our productivity. We rely on code analysis tools to improve our code quality and code generators to aid in application construction. As these tools get smarter, they need access to more and more of the deep code knowledge that only compilers possess. This is the core mission of the Roslyn project: opening up the black boxes and allowing tools and end users to share in the wealth of information compilers have about our code. Instead of being opaque source-code-in and object-code-out translators, through the Roslyn project, compilers become services—APIs that you can use for code related tasks in your tools and applications.

The transition to compilers as services dramatically lowers the barrier to entry for creating code focused tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and transformation, interactive use of the C# and VB languages, and embedding of C# and VB in domain specific languages.

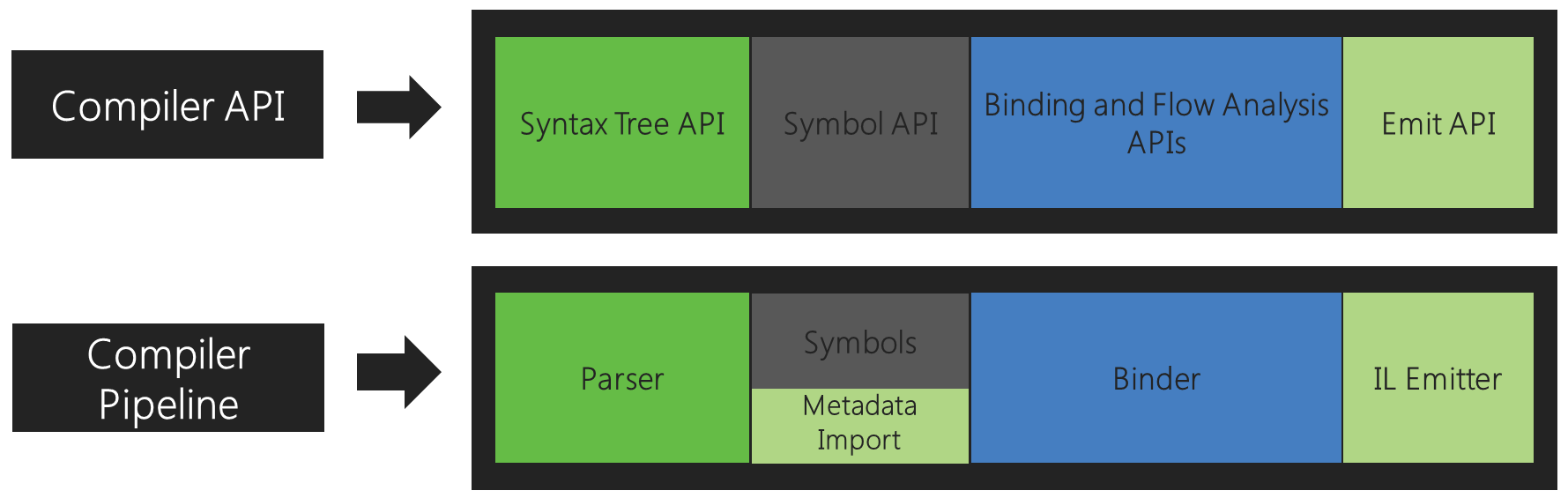
The Microsoft “Roslyn” CTP previews the new language object models for code generation, analysis, and refactoring, and the upcoming support for scripting and interactive use of C# and Visual Basic. This document provides a conceptual overview of the Roslyn project. Further details can be found in the walkthroughs and samples included in the Roslyn CTP.

# Exposing the Core APIs

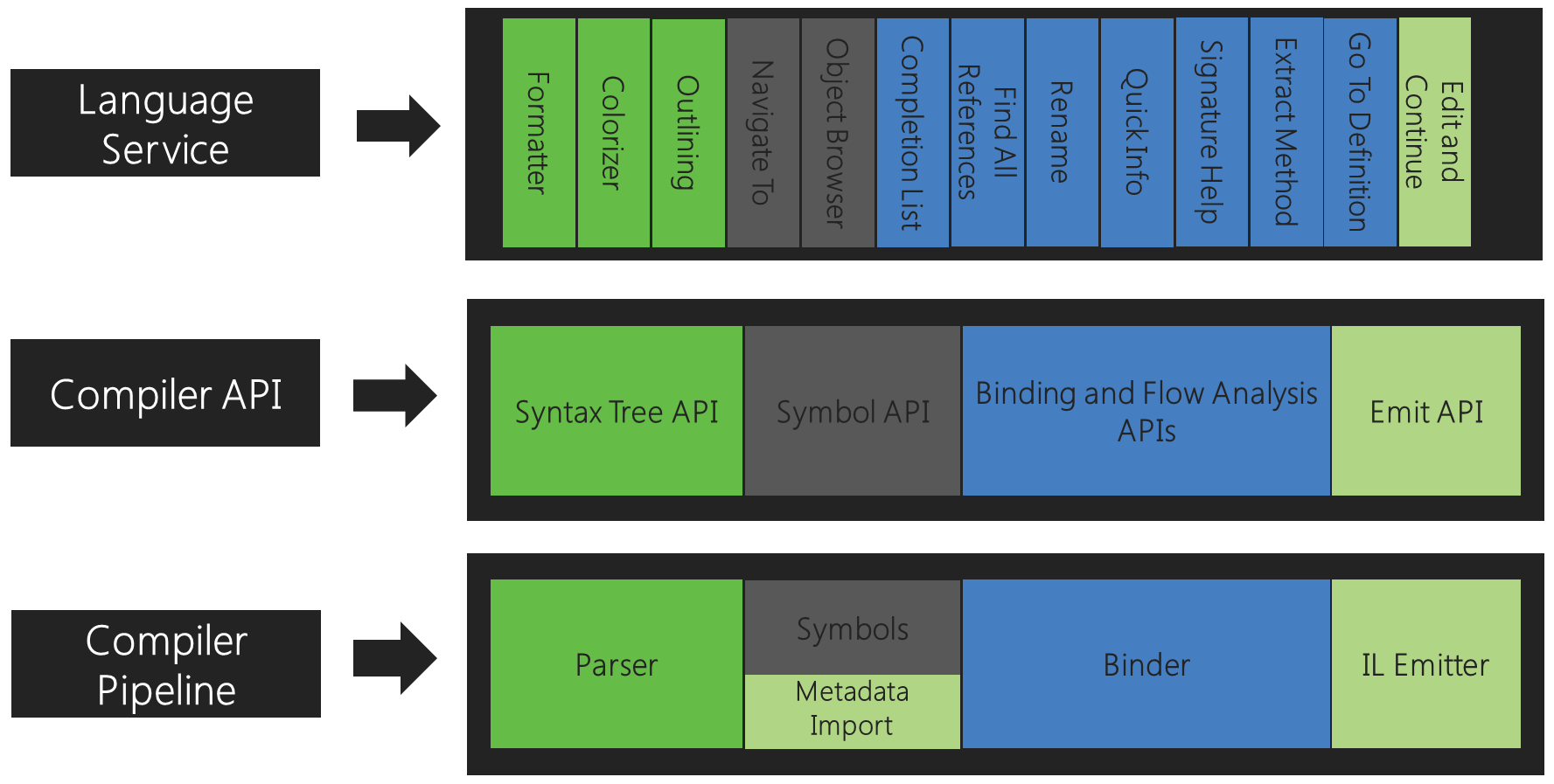
The Roslyn project exposes the C# and Visual Basic compiler’s code analysis to you as a consumer by providing an API layer that mirrors a traditional compiler pipeline.



Each phase of this pipeline is now a separate component. First the parse phase, where source is tokenized and parsed into syntax that follows the language grammar. Second the declaration phase, where declarations from source and imported metadata are analyzed to form named symbols. Next the bind phase, where identifiers in the code are matched to symbols. Finally, the emit phase, where all the information built up by the compiler is emitted as an assembly.



Corresponding to each of those phases, an object model is surfaced that allows access to the information at that phase. The parsing phase is exposed as a syntax tree, the declaration phase as a hierarchical symbol table, the binding phase as a model that exposes the result of the compiler’s semantic analysis and the emit phase as an API that produces IL byte codes.



Each compiler combines these components together as a single end-to-end whole.

To ensure that the public Compiler APIs are sufficient for building world-class IDE features, the Roslyn language services are built using them. For instance, the code outlining and formatting features use the syntax trees, the Object Browser and navigation features use the symbol table, refactorings and Go to Definition use the semantic model, and Edit and Continue uses all of these, including the Emit API.

## Key Concepts

This section discusses the key concepts regarding architectural layers in the Roslyn project.

## API Layers

The Roslyn project exposes three collections of APIs – the Compiler APIs, the Services APIs, and the Editor Services APIs.

### Compiler APIs

The compiler layer contains the object models that correspond with information exposed at each phase of the compiler pipeline, both syntactic and semantic. The compiler layer also contains a representation of a single invocation of a compiler, including assembly references, compiler options, and source code files. There are two distinct APIs that represent the C# language and the Visual Basic language. These two APIs are similar in shape. This layer has no dependencies on Visual Studio components.

#### Scripting APIs

As a part of the compiler layer, a set of he scripting APIs are exposed that represents a runtime execution context for C# or Visual Basic snippets of code. It contains a scripting engine that allows evaluation of expressions and statements as top-level constructs in programs. This layer has no dependencies on Visual Studio components.

*Note:* This document does not cover the Scripting APIs. To learn more, see the *Interactive - Scripting Introduction.docx* and *Interactive - Paint-like Application Scripting.docx* documents.

### Services APIs

The Services layer contains the Workspace API, which is the starting point for doing code analysis and refactoring over entire solutions. It assists you in organizing all the information about the projects in a solution into single object model, offering you direct access to the compiler layer object models without needing to parse files, configure options or manage project to project dependencies.

In addition, the services layer surfaces a set of commonly used APIs used when implementing code analysis and refactoring tools that function within a host environment like the Visual Studio IDE, such as the Find All References, Formatting, and Code Generation APIs.

This layer has no dependencies on Visual Studio components.

### Editor Services APIs

The editor services layer contains all the Visual Studio IDE features, such as IntelliSense, refactorings, and code formatting features. It also contains the Editor Services APIs, which allow a user to easily extend Visual Studio.

This layer has a dependency on the Visual Studio text editor and the Visual Studio software development kit (SDK).

# Working with Syntax

The most fundamental data structure exposed by the Compiler APIs is the syntax tree. These trees represent the lexical and syntactic structure of source code. They serve two important purposes:

1. To allow tools—such as an IDE, add-ins, code analysis tools, and refactorings—to see and process the syntactic structure of source code in a user’s project.
2. To enable tools—such as refactorings and an IDE—to create, modify, and rearrange source code in a natural manner without having use direct text edits. By creating and manipulating trees, tools can easily create and rearrange source code.

## Key Concepts

This section will discuss the key concepts regarding the Syntax API.

### Syntax Trees

Syntax trees are the primary structure used for compilation, code analysis, binding, refactoring, IDE features, and code generation. No part of the source code is understood without it first being identified and categorized into one of many well-known structural language elements.

Syntax trees have three key attributes. The first attribute is that syntax trees hold all the source information in *full fidelity*. This means that the syntax tree contains every piece of information found in the source text, every grammatical construct, every lexical token, and everything else in between including whitespace, comments, and preprocessor directives. For example, each literal mentioned in the source is represented exactly as it was typed. The syntax trees also represent errors in source code when the program is incomplete or malformed, by representing skipped or missing tokens in the syntax tree.

This enables the second attribute of syntax trees. A syntax tree obtained from the parser is completely *round-trippable* back to the text it was parsed from. From any syntax node, it is possible to get the text representation of the sub-tree rooted at that node. This means that syntax trees can be used as a way to construct and edit source text. By creating a tree you have by implication created the equivalent text, and by editing a syntax tree, making a new tree out of changes to an existing tree, you have effectively edited the text.

The third attribute of syntax trees is that they are *immutable* and thread-safe. This means that after a tree is obtained, it is a snapshot of the current state of the code, and never changes. This allows multiple users to interact with the same syntax tree at the same time in different threads without locking or duplication. Because the trees are immutable and no modifications can be made directly to a tree, factory methods help create and modify syntax trees by creating additional snapshots of the tree. The trees are efficient in the way they reuse underlying nodes, so the new version can be rebuilt fast and with little extra memory.

A syntax tree is literally a tree data structure, where non-terminal structural elements parent other elements. Each syntax tree is made up of *nodes, tokens,* and *trivia.*

### Syntax Nodes

Syntax nodes are one of the primary elements of syntax trees. These nodes represent syntactic constructs such as declarations, statements, clauses, and expressions. Each category of syntax nodes is represented by a separate class derived from SyntaxNode. The set of node classes is not extensible.

All syntax nodes are non-terminal nodes in the syntax tree, which means they always have other nodes and tokens as children. As a child of another node, each node has a *parent* node that can be accessed through the Parent property. Because nodes and trees are immutable, the parent of a node never changes. The root of the tree has a null parent.

Each node has a ChildNodes method, which returns a list of child nodes in sequential order based on its position in the source text. This list does not contain tokens. Each node also has a collection of Descendant**\*** methods**—**such as DescendantNodes, DescendantTokens, or DescendantTrivia**—**that represent a list of all the nodes, tokens, or trivia that exist in the sub-tree rooted by that node.

In addition, each syntax node subclass exposes all the same children through strongly typed properties. For example, a BinaryExpressionSyntax node class has three additional properties specific to binary operators: Left, OperatorToken, and Right. The type of Left and Right is ExpressionSyntax, and the type of OperatorToken is SyntaxToken.

Some syntax nodes have optional children. For example, an IfStatementSyntax has an optional ElseClauseSyntax. If the child is not present, the property returns null.

### Syntax Tokens

Syntax tokens are the terminals of the language grammar, representing the smallest syntactic fragments of the code. They are never parents of other nodes or tokens. Syntax tokens consist of keywords, identifiers, literals, and punctuation.

For efficiency purposes, the SyntaxToken type is a CLR value type. Therefore, unlike syntax nodes, there is only one structure for all kinds of tokens with a mix of properties that have meaning depending on the kind of token that is being represented.

For example, an integer literal token represents a numeric value. In addition to the raw source text the token spans, the literal token has a Value property that tells you the exact decoded integer value. This property is typed as Object because it may be one of many primitive types.

The ValueText property tells you the same information as the Value property; however this property is always typed as String. An identifier in C# source text may include Unicode escape characters, yet the syntax of the escape sequence itself is not considered part of the identifier name. So although the raw text spanned by the token does include the escape sequence, the ValueText property does not. Instead, it includes the Unicode characters identified by the escape.

### Syntax Trivia

Syntax trivia represent the parts of the source text that are largely insignificant for normal understanding of the code, such as whitespace, comments, and preprocessor directives.

Because trivia are not part of the normal language syntax and can appear anywhere between any two tokens, they are not included in the syntax tree as a child of a node. Yet, because they are important when implementing a feature like refactoring and to maintain full fidelity with the source text, they do exist as part of the syntax tree.

You can access trivia by inspecting a token’s LeadingTrivia or TrailingTrivia collections. When source text is parsed, sequences of trivia are associated with tokens. In general, a token owns any trivia after it on the same line up to the next token. Any trivia after that line is associated with the following token. The first token in the source file gets all the initial trivia, and the last sequence of trivia in the file is tacked onto the end-of-file token, which otherwise has zero width.

Unlike syntax nodes and tokens, syntax trivia do not have parents. Yet, because they are part of the tree and each is associated with a single token, you may access the token it is associated with using the Token property.

Like syntax tokens, trivia are value types. The single SyntaxTrivia type is used to describe all kinds of trivia.

### Spans

Each node, token, or trivia knows its position within the source text and the number of characters it consists of. A text position is represented as a 32-bit integer, which is a zero-based Unicode character index. A TextSpan object is the beginning position and a count of characters, both represented as integers. If TextSpan has a zero length, it refers to a location between two characters.

Each node has two TextSpan properties: Span and FullSpan.

The Span property is the text span from the start of the first token in the node’s sub-tree to the end of the last token. This span does not include any leading or trailing trivia.

The FullSpan property is the text span that includes the node’s normal span, plus the span of any leading or trailing trivia.

For example:

if (x > 3)  
{

// this is bad  
 throw new Exception("Not right."); // better exception?  
}

The statement node inside the block has a span indicated by the purple underline. It includes the characters throw new Exception(“Not right.”);. The full span is indicated by the orange underline. It includes the same characters as the span and the characters associated with the leading and trailing trivia.

### Kinds

Each node, token, or trivia has a Kind property, of type SyntaxKind, that identifies the exact syntax element represented. Each language, C# or VB, has a single SyntaxKind enumeration that lists all the possible nodes, tokens, and trivia elements in the grammar.

The Kind property allows for easy disambiguation of syntax node types that share the same node class. For tokens and trivia, this property is the only way to distinguish one type of element from another.

For example, a single BinaryExpressionSyntax class has Left, OperatorToken, and Right as children. The Kind property distinguishes whether it is an AddExpression, SubtractExpression, or MultiplyExpression kind of syntax node.

### Errors

Even when the source text contains syntax errors, a full syntax tree that is round-trippable to the source is exposed. When the parser encounters code that does not conform to the defined syntax of the language, it uses one of two techniques to create a syntax tree.

First, if the parser expects a particular kind of token, but does not find it, it may insert a *missing token* into the syntax tree in the location that the token was expected. A missing token represents the actual token that was expected, but it has an empty span, and its IsMissing property returns true.

Second, the parser may skip tokens until it finds one where it can continue parsing. In this case, the skipped tokens that were skipped are attached as a trivia node with the kind SkippedTokens.

## Obtaining a Syntax Tree

The common Syntax APIs are found in the Roslyn.Compilers and the Roslyn.Compilers.Common namespace, while the language specific Syntax APIs are found in Roslyn.Compilers.CSharp and Roslyn.Compilers.VisualBasic. They are available if you include the following using directives:

using Roslyn.Compilers;

using Roslyn.Compilers.Common;

using Roslyn.Compilers.CSharp;

using Roslyn.Compilers.VisualBasic;

The easiest way to obtain a syntax tree corresponding to a complete program is to use the SyntaxTree.ParseText function, which produces a syntax tree by parsing the source code. To see how to obtain a syntax tree from source text, consider this example:

SyntaxTree tree = SyntaxTree.ParseText(

                @"using System;

                namespace HelloWorld

                {

                    class Program

                    {

                        static void Main(string[] args)

                        {

                            Console.WriteLine(""Hello, World!"");

                        }

                    }

                }");

You can also obtain a syntax node by using the parsing functions on the Syntax class. These functions allow you to parse a subset of a source file, such as an expression, statement, or comment.

ExpressionSyntax myExpression = Syntax.ParseExpression("1+1");

StatementSyntax myStatement = Syntax.ParseStatement("for (int i = 0; i <  length; i++) { }");

The type of myExpression is BinaryExpressionSyntax, and the type of myStatement is ForStatementSyntax.

## Manipulating Syntax Nodes

To allow refactoring and modification of code, you can easily modify syntax trees by constructing new syntax trees from pieces of existing trees and other nodes you create.

### Creating a node

To create a node, you use the factory methods in the Syntaxclass. Each kind of node, token, or trivia has a corresponding factory method that you can use to create an instance of that type.

To make a namespace declaration node you can use the Syntax.NamespaceDeclaration method.

To create the qualified name Roslyn.Compilers you can create it from its parts.

QualifiedNameSyntax namespaceName = Syntax.QualifiedName(

left: Syntax.IdentifierName("Roslyn"),

right: Syntax.IdentifierName("Compilers"));

Or you can get the parser to do it for you.

NameSyntax namespaceName = Syntax.ParseName("Roslyn.Compilers");

Finally, you can create the namespace declaration.

NamespaceDeclarationSyntax myNamespace = Syntax.NamespaceDeclaration(

namespaceName);

If myNamespace were converted to text, this would be the result.

namespaceRoslyn.Compilers{}

Unfortunately, this text has no whitespace and will produce errors if you try to parse it. This is probably not what you want. When creating syntax nodes and tokens you have to specify the trivia explicitly.

Rather than use the default namespace token created for you, you can add a space after the namespace token namespace by specifying it as an argument when you construct the token. The Syntax class has a Space property that is a trivia that represents a whitespace with a single space character.

SyntaxToken namespaceToken = Syntax.Token(

SyntaxKind.NamespaceKeyword,

Syntax.Space);

This will result in the following text.

namespace Roslyn.Compilers{}

It is also possible to get the declaration node to format itself using simple built-in layout rules. You can create a new formatted node using the NormalizeWhitespace function.

NamespaceDeclarationSyntax formattedNamespace = myNamespace. NormalizeWhitespace();

This will result in the following text.

namespace Roslyn.Compilers

{

}

### Modifying an existing node

If you intend to change the source code based on your analysis, you will often want to modify an existing node rather than creating an entirely new one. You can create an updated node from an existing one by specifying the elements you want changed.

For instance, you can update the namespace HelloWorld from the syntax tree in Section 3.2 to the new namespace Roslyn.Compilers.

To get the old namespace.

NamespaceDeclarationSyntax oldNamespace = tree.GetRoot().DescendantNodes()

.OfType<NamespaceDeclarationSyntax>()

.First();

To create an updated node with the new namespace name.

NamespaceDeclarationSyntax newNamespace =

oldNamespace.WithName(

                                   namespaceName

                                       );

If newNamespace were converted to text, this would be the result.

namespace Roslyn.Compilers {

class Program

       {

        static void Main(string[] args)

             {

              Console.WriteLine(""Hello, World!"");

             }

       }

}

### Replacing a node

You can replace a node entirely in the current syntax tree by using the ReplaceNode function to create a tree with the old node replaced with the new node.

CompilationUnitSyntax newRoot = tree.GetRoot().ReplaceNode(oldNamespace, newNamespace);

If newRoot were converted to text, you would see this result.

using System;

namespace Roslyn.Compilers

{

class Program

    {

    static void Main(string[] args)

           {

           Console.WriteLine(""Hello, World!"");

}

}

}

### Creating a Syntax Tree

When you modify a syntax tree you don’t really change it. Syntax trees are immutable. Instead, you create new trees from old trees with new nodes you have constructed. However, the functions that make these trees do not make new SyntaxTree instances. They make new syntax nodes.

If you need to get back to an actual SyntaxTree instance, you can construct one using the SyntaxTree.Create function.

SyntaxTree newTree = SyntaxTree.Create(newRoot);

# Working with Semantics

Syntax trees represent the lexical and syntactic structure of source code. Although this information alone is enough to describe all the declarations and logic in the source, it is not enough information to identify what is being referenced.

For example, many types, fields, methods, and local variables with the same name may be spread throughout the source. Although each of these is uniquely different, determining which one an identifier actually refers to often requires a deep understanding of the language rules.

In addition to the program elements represented in source code, programs can also refer to previously compiled libraries, packaged in assembly files. Although no source code is available for assemblies and therefore no syntax nodes or trees, programs can still refer to elements inside them.

In addition to a syntactic model of the source code, a semantic model encapsulates the language rules, giving you an easy way to make these distinctions.

## Key Concepts

This section will discuss the key concepts regarding the Semantic API and how to work with semantic information.

### Compilation

A *compilation* is a representation of everything needed to compile a C# or Visual Basic program, which includes all the assembly references, compiler options, and source files.

Because all this information is in one place, the elements contained in the source code can be described in more detail. The compilation represents each declared type, member, or variable as a *symbol*. The compilation contains a variety of methods that help you find and relate the symbols that have either been declared in the source code or imported as metadata from an assembly.

Similar to syntax trees, compilations are immutable. After you create a compilation, it cannot be changed by you or anyone else you might be sharing it with. However, you can create a new compilation from an existing compilation, specifying a change as you do so. For example, you might create a compilation that is the same in every way as an existing compilation, except it may include an additional source file or assembly reference.

### Symbols

A *symbol* represents a distinct element declared by the source code or imported from an assembly as metadata. Every namespace, type, method, property, field, event, parameter, or local variable is represented by a symbol.

A variety of methods and properties on the Compilation type help you find symbols. For example, you can find a symbol for a declared type by its common metadata name. You can also access the entire symbol table as a tree of symbols rooted by the global namespace.

Symbols also contain additional information that the compiler determined from the source or metadata, such as other referenced symbols. Each kind of symbol is represented by a separate subclass of the class Symbol, each with its own methods and properties detailing the information the compiler has gathered. Many of these properties directly reference other symbols. For example, the ReturnType property of the MethodSymbol class tells you the actual type symbol that the method declaration referenced.

Symbols present a common representation of namespaces, types, and members, between source code and metadata. For example, a method that was declared in source code and a method that was imported from metadata are both represented by a MethodSymbol with the same properties.

Symbols are similar in concept to the CLR type system as represented by the System.Reflection API, yet they are richer in that they model more than just types. Namespaces, local variables, and labels are all symbols. In addition, symbols are a representation of language concepts, not CLR concepts. There is a lot of overlap, but there are many meaningful distinctions as well. For instance, an iterator method in C# or Visual Basic is a single symbol. However, when the iterator method is translated to CLR metadata, it is a type and multiple methods.

### Semantic Model

A *semantic model* represents all the semantic information for a single source file. You can use it to discover the following:

* The symbols referenced at a specific location in source.
* The resultant type of any expression.
* All diagnostics, which are errors and warnings.
* How variables flow in and out of regions of source.
* The answers to more speculative questions.

## Obtaining a Compilation

You can create a compilation by combining syntax trees, assembly references, and compilation options. Creating a compilation does not produce an assembly on disk. A compilation is merely an in-memory representation of the analysis the compiler does of the source files and imported metadata.

You can create a compilation using the Create factory method. You can supply all the syntax trees, assembly references, and options in one call or you can spread them out over multiple calls.

SyntaxTree tree = SyntaxTree.ParseText(sourceText);

MetadataReference mscorlib = MetadataReference.CreateAssemblyReference(

"mscorlib");

Compilation compilation = Compilation.Create(

outputName: "HelloWorld",

syntaxTrees: new [] { tree },

references: new [] { mscorlib });

Compilation compilation = Compilation.Create("HelloWorld")

.AddReferences(mscorlib)

.AddSyntaxTrees(tree);

Creating multiple intermediate compilations is not expensive. All analysis work by the compiler is deferred until you need it.

## Obtaining Symbols

To navigate the hierarchy of symbols inside a compilation, you can call properties and methods on the compilation and other symbols.

You can obtain a symbol representing the global namespace.

NamespaceSymbol globalNamespace = compilation.GlobalNamespace;

From a namespace symbol, you can get all the members in that namespace, which includes both namespaces and types.

foreach (Symbol member in globalNamespace.GetMembers())

{

// Process member

}

You can also ask for members with a specific name. This method is more efficient than first obtaining all members and filtering that list based on your interest. For example, to get the symbol for “System.String”.

NamespaceSymbol systemNamespace = globalNamespace.GetMembers("System")

.First() as NamespaceSymbol;

NamedTypeSymbol stringType = systemNamespace.GetMembers("String")

.First() as NamedTypeSymbol;

You can also obtain a type directly if you know the metadata name. The metadata name is the name that is encoded and referenced in CLR metadata.

NamedTypeSymbol stringType = compilation.GetTypeByMetadataName(

"System.String");

From a type symbol, you can obtain all the fields, methods, properties, events, and nested types within that type:

foreach (Symbol member in stringType.GetMembers())

{

// Process member

}

You can use other properties and methods on each symbol to navigate the symbol model. For example, you can obtain the base class or the interfaces of a type:

NamedTypeSymbol baseType = stringType.BaseType;

foreach (NamedTypeSymbol i in stringType.Interfaces) { /\* ... \*/ }

## Asking Semantic Questions

When you want to know the resultant type of an expression or the method symbol that was used in an invocation, you will need to obtain a SemanticModel for the source file.

You can obtain a SemanticModel by requesting one from the compilation.

SemanticModel semanticModel = compilation.GetSemanticModel(tree);

The syntax tree instance for the related source file must be one of the instances specified as part of the compilation. Alternatively, you can obtain a semantic model from the IDocument class. See Section 5, Working with a Workspace.

When you have a semantic model, you can discover the symbol associated with any declaration in the tree.

MethodDeclarationSyntax methodDecl = tree.GetRoot()

.DescendantNodes()

.OfType<MethodDeclarationSyntax>()

.First();

Symbol methodSymbol = semanticModel.GetDeclaredSymbol(methodDecl);

You can find the resultant type of an expression:

ExpressionSyntax expression = methodDecl.DescendantNodes()

.OfType<ExpressionSyntax>()

.First();

TypeInfo info = semanticModel.GetTypeInfo(expression);

TypeSymbol resultantType = info.Type;

You can also find the method symbol referenced in an invocation:

InvocationExpressionSyntax invocation = expression.DescendantNodes()

.OfType<InvocationExpressionSyntax>()

.First();

SymbolInfo info = semanticModel.GetSymbolInfo(invocation);

Symbol invokedSymbol = info.Symbol;

Note that the TypeSymbol and Symbol properties are different. The TypeSymbol property refers to the expression’s resultant type, and the Symbol property refers to a single symbol referenced by the expression. Some expressions have a type but no symbols. Other expressions have both. An invocation expression has both a symbol and a resultant type. The symbol is the method symbol being invoked, and the resultant type is the return type of the method.

To determine the resultant type of an expression that does not appear in the source or does but at a different location, use the speculative GetSpeculativeTypeInfo method with the BindAsExpression option:

ExpressionSyntax otherExpression = Syntax.ParseExpression("X.Y");

int location = methodDecl.Body.Span.Start;

TypeInfo info = semanticModel.GetSpeculativeTypeInfo(

location, otherExpression,

SpeculativeBindingOption.BindAsExpression);

TypeSymbol resultantType = info.Type;

Likewise, to determine the type of a type name expression that does not appear in the source or does but in a different location, use the GetSpeculativeSymbolInfo method with the BindAsTypeOrNamespace option:

ExpressionSyntax typeExpression = Syntax.ParseTypeName("System.String");

int location = methodDecl.Body.Span.Start;

SymbolInfo info = semanticModel.GetSpeculativeSymbolInfo(

location, typeExpression,

SpeculativeBindingOption.BindAsTypeOrNamespace);

Symbol resultantType = info.Symbol;

## Working with Fault Tolerance

Rarely is source code free of errors. If you are building a tool to analyze source code as a user types in an editor, you will have to deal with partially finished expressions and other unintended errors.

The semantic model helps you understand the code even in the face of errors. The analysis engine underneath the semantic model provides a best guess of resultant types for expressions. For example, a call to a method might be ambiguous due to failure to pick a single overload, yet the candidate methods all have the same return type. The resultant type of the expression is deemed to be that common type, even as the expression overall is in error.

If no single correct symbol can be determined for an invocation, it is also possible to discover the candidate(s) that the compiler was attempting to pick from.

InvocationExpressionSyntax invocation = expression.DescendantNodes()

.OfType<InvocationExpressionSyntax>()

.First();

SymbolInfo info = semanticModel.GetSymbolInfo(invocation);

Symbol invokedSymbol = info.Symbol;

if (invokedSymbol == null)

{

ReadOnlyArray<Symbol> candidates = info.CandidateSymbols;

CandidateReason reason = info.CandidateReason;

...

}

The CandidateSymbols property is a list of all candidates, and the CandidateReason is an enumeration value detailing why a single symbol could not be determined.

## Using the Control and Data Flow Analysis APIs

You can use the Control and Data Flow Analysis APIs to determine which local variables are used in a region of code or which statements transfer control into or out of a region.

From a semantic model, you can determine how control flows into and out of a region of source using the AnalyzeControlFlow method.

BlockSyntax statement = methodDecl.Body;

ControlFlowAnalysis controlFlow =

semanticModel.AnalyzeControlFlow(statement);

To determine the return statements in the region.

var returns = controlFlow.ReturnStatements;

Or any statement that may jump out of the region.

var jumpsOut = controlFlow.ExitPoints;

Or any statement that may jump into the region.

var jumpsIn = controlFlow.EntryPoints;

To determine which variables are assigned or referenced in a region of source code, use the AnalyzeDataFlow method.

DataFlowAnalysis dataFlow =

semanticModel.AnalyzeDataFlow(statement);

To determine the local variables declared in a region.

var declared = dataFlow.VariablesDeclared;

To determine the symbols inside a region that are assigned a value outside the region.

var usedInside = dataFlow.DataFlowsIn;

To determine the symbols that are assigned a value inside the region but that are consequently used outside the region.

var usedOutside = dataFlow.DataFlowsOut;

In addition, you can determine any local variable that is read or written inside or outside the region.

# Working with a Workspace

The Services layer is the starting point for doing code analysis and refactoring over entire solutions. Within this layer, the Workspace API assists you in organizing all the information about the projects in a solution into single object model, offering you direct access to compiler layer object models like source text, syntax trees, semantic models and compilations without needing to parse files, configure options or manage inter-project dependencies.

Host environments, like an IDE, provide a workspace for you corresponding to the open solution. It is also possible to use this model outside of an IDE by simply loading a solution file.

## Key Concepts

This section will discuss the key concepts regarding the Workspace API and how to easily work with solutions, projects, and documents.

### Workspace

A *workspace* is an active representation of your solution as a collection of projects, each with a collection of documents. A workspace is typically tied to a host environment that is constantly changing as a user types or manipulates properties.

The workspace provides access to the current model of the solution. When a change in the host environment occurs, the workspace fires corresponding events, and the CurrentSolution property is updated. For example, when the user types in a text editor corresponding to one of the source documents, the workspace uses an event to signal that the overall model of the solution has changed and which document was modified. You can then react to those changes by analyzing the new model for correctness, highlighting areas of significance, or by making a suggestion for a code change.

You can also create stand-alone workspaces that are disconnected from the host environment or used in an application that has no host environment.

### Solutions, Projects, Documents

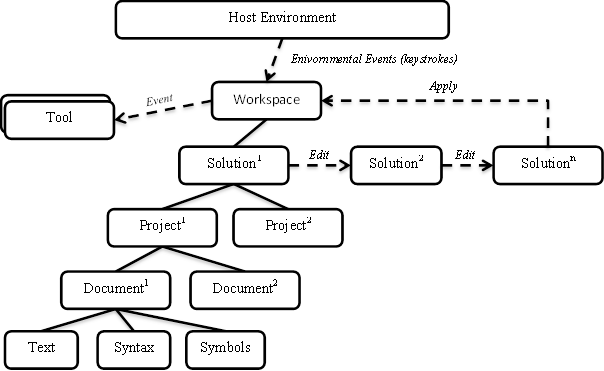
Although a workspace may change every time a key is pressed, you can work with the model of the solution in isolation.

A *solution* is an immutable model of the projects and documents. This means that the model can be shared without locking or duplication. After you obtain a solution instance from the Workspace’s CurrentSolution property, that instance will never change. However, like with syntax trees and compilations, you can modify solutions by constructing new instances based on existing solutions and specific changes. To get the workspace to reflect your changes, you must explicitly apply the changed solution back to the workspace.

A *project* is a part of the overall immutable solution model. It represents all the source code documents, parse and compilation options, and both assembly and project-to-project references. From a project, you can access the corresponding compilation without needing to determine project dependencies or parse any source files.

A *document* is also a part of the overall immutable solution model. A document represents a single source file from which you can access the text of the file, the syntax tree, and the semantic model.

The following diagram is a representation of how the Workspace relates to the host environment, tools, and how edits are made.



## Obtaining a Workspace

The Workspace APIs is found in the Roslyn.Services namespace, and they are available if you include the following using directive:

using Roslyn.Services;

The workspace you use will typically be provided directly by the host environment (such as the Visual Studio IDE). However, you can work with a workspace outside of a host environment by constructing your own IWorkspace instance.

You can construct a workspace by loading a solution file.

IWorkspace workspace = Workspace.LoadSolution(@"HelloWorld.sln");

ISolution solution = workspace.CurrentSolution;

Unlike a workspace provided by the host environment, this manually loaded workspace will not update automatically based on external changes in the host environment. Unless you update the workspace yourself, the current solution will stay the same for the duration of its use.

If you do not expect to make edits and save changes back to the workspace, you can construct an ISolution instance separately.

ISolution solution = Solution.Load(@"HelloWorld.sln");

Alternatively, you can create an empty ISolution instance and build up your projects and documents manually.

ProjectId pid1, pid2;

DocumentId did1, did2;

ISolution solution =   
 Solution.Create(SolutionId.CreateNewId("Solution"))

    .AddCSharpProject("Project1.dll", "Project1", out pid1)

    .AddDocument(pid1, "A.cs", "public class A { }", out did1)

    .AddCSharpProject("Project2.dll", "Project2", out pid2)

    .AddDocument(pid2, "B.cs", "class B : A { }", out did2)

    .AddProjectReference(pid2, pid1);

## Working with a Solution

The solution is the immutable part of the workspace, representing a snapshot in time of the state of your projects and documents. Each solution contains a collection of IProject instances.

You can access a project by its ProjectId, if you already know it.

IProject project = solution.GetProject(projectId);

Or you can access projects by assembly name.

IEnumerable<IProject> project = solution.GetProjectsByAssemblyName("Project1.dll");

Or you can enumerate all projects in the solution.

foreach (IProject project in solution.Projects) { ... }

Each project contains a collection of IDocument instances. You can access a document by its DocumentId, if you know it.

IDocument document = project.GetDocument(docId);

Or you can enumerate all documents in the project.

foreach (IDocument document in project.Documents) { ... }

You can access the text from a document.

IText text = document.GetText();

Or the syntax tree from a document.

CommonSyntaxTree syntax = document.GetSyntaxTree();

Or the semantic model from a document.

ISemanticModel semanticModel = document.GetSemanticModel();

## Updating a Workspace

If you intend to change the source code based on your analysis, you will eventually need to update the Workspace when you save your changes. Before you can update the workspace, you need to create an ISolution instance that includes your changes.

You can create a new solution by updating an existing one with the text of a document that you obtained from a transformed syntax tree.

IText newText = newRoot.GetText();

ISolution newSolution = solution.UpdateDocument(docId, newText);

When you have a new solution instance, you can either continue on with your analysis or commit the changes back to the workspace. Creating a solution from changes to an old solution does not automatically change the workspace. You can commit your changes by applying them to the workspace:

workspace.ApplyChanges(solution, newSolution);

To apply the changes, you must specify both the original solution you started from and the new solution.

# Working with Editor Services

The Compiler and Services APIs are sufficient for code analysis and code manipulation; however, in many scenarios, you may want to create an extension to Visual Studio or augment an existing feature in the IDE. The Editor Services APIs allow you to easily connect your code analysis logic with Visual Studio features such as IntelliSense, smart tags, and wavy underlines for errors.

The Editor Services APIs have a dependency on the Visual Studio text editor.

## Key Concepts

This section will discuss the key concepts in working with the Editor Services APIs to gain access to IDE services such as refactoring, find all references, formatting, and IntelliSense extensions.

### Managed Extensibility Framework

The Editor Services APIs have a dependency on the *Managed Extensibility Framework*, otherwise known as MEF, to load and discover extensions to Visual Studio. MEF is a library for creating lightweight, extensible applications. It allows application developers to discover and use extensions with no configuration required. It also lets extension developers easily encapsulate code and avoid fragile hard dependencies. MEF not only allows extensions to be reused within applications, but across applications as well.

## Creating an IDE extension

The Editor Services APIs are found in the Roslyn.Services.Editor namespace and they are available if you include the following using directive:

using Roslyn.Services.Editor;

To extend the Visual Studio IDE, the Editor Services API exposes a set of extension points to existing language service features. A few examples of these include the completion list, syntax classifiers, and code refactorings. For each of these extensions, you need to create your own MEF provider and ensure that there is at least one Export\* attribute that describes important details about the provider.

To write your own code refactoring, create a provider for a *code issue*. The code issue provider makes it easy to surface an error or suggestion to the user as a wavy underline in the editor or appear in the Error List window.

[ExportCodeIssueProvider("FirstQuickFixCS", LanguageNames.CSharp)]

class CodeIssueProvider : ICodeIssueProvider {}

In this case, the code issue provider will only appear if the source file contains C# code. To create a code issue for VB, use LanguageNames.VisualBasic.

Every code issue provider must implement the ICodeIssueProvider interface. This interface contains two methods for operating on syntax nodes or tokens, and two properties to restrict the kinds of syntax nodes and tokens for which the methods will be called.

public interface ICodeIssueProvider

{

IEnumerable<CodeIssue> GetIssues(

IDocument document,

CommonSyntaxNode node,

CancellationToken cancellationToken);

IEnumerable<CodeIssue> GetIssues(

IDocument document,

CommonSyntaxToken token,

CancellationToken cancellationToken);

IEnumerable<CodeIssue> SyntaxNodeTypes { get; }

IEnumerable<int> SyntaxTokenKinds { get; }

}

In addition to highlighting a code issue as a wavy underline, it is possible to create an automatic edit that will fix or change your source code. A *code action* represents a single action that can be applied to source code. It is created by implementing the ICodeAction interface.

public interface ICodeAction

{

string Description { get; }

ICodeActionEdit GetEdit(CancellationToken cancellationToken);

}

To understand more about how to use the Editor Services APIs, see the How to Write a Quick Fix walkthrough and the samples included in the Roslyn CTP.

# Summary

Project Roslyn exposes a set of Compiler APIs, Services APIs, and Editor Services APIs that provides rich information about your source code and that has full fidelity with the C# and Visual Basic languages. The transition to compilers as a service dramatically lowers the barrier to entry for creating code focused tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and transformation, interactive use of the C# and VB languages, and embedding of C# and VB in domain specific languages.