.NET Compiler Platform (“Roslyn”)

How To: Write a Visual Basic Diagnostic and Code Fix

April 2014

# Introduction

In previous releases of Visual Studio, it has been difficult to create custom warnings that target C# or Visual Basic. With the Diagnostics API in the .NET Compiler Platform (“Roslyn”), this once difficult task has become easy! All that is needed is to perform a bit of analysis to identify an issue, and optionally provide a tree transformation as a code fix. The heavy lifting of running your analysis on a background thread, showing squiggly underlines in the editor, populating the Visual Studio Error List, creating “light bulb” suggestions and showing rich previews is all done for you automatically.

In this walkthrough, we’ll explore the creation of a Diagnostic and an accompanying Code Fix using the Roslyn APIs. A Diagnostic is a way to perform source code analysis and report a problem to the user. Optionally, a Diagnostic can also provide a Code Fix which represents a modification to the user’s source code. For example, a Diagnostic could be created to detect and report any local variable names that begin with an uppercase letter, and provide a Code Fix that corrects them.

# Writing the Diagnostic

Suppose that you wanted to report to the user any local variable declarations that can be converted to local constants. For example, consider the following code:

Dim x As Integer = 0

Console.WriteLine(x)

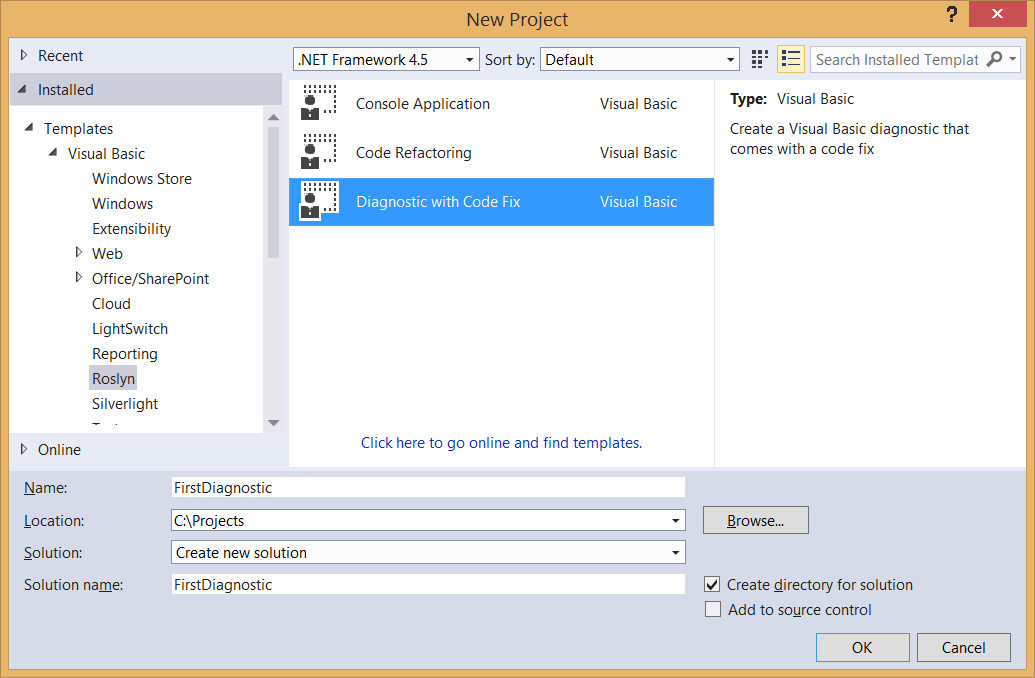
In the code above, x is assigned a constant value and is never written to. Thus, it can be declared using the Const modifier:

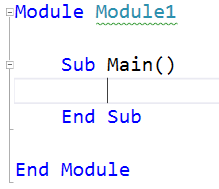
Const x As Integer = 0

Console.WriteLine(x)

The analysis to determine whether a variable can be made constant is actually fairly involved, requiring syntactic analysis, constant analysis of the initializer expression and dataflow analysis to ensure that the variable is never written to. However, performing this analysis with the .NET Compiler Platform and exposing it as a Diagnostic is pretty easy.

1. First you’ll create a new Visual Basic Diagnostic project.
   * In Visual Studio, choose File -> New -> Project… to display the New Project dialog.
   * Under Visual Basic -> Roslyn, choose “Diagnostic with Code Fix.”
   * Name your project “FirstDiagnostic” and click OK.



1. Press Ctrl+F5 to run the newly created Diagnostic project in a second instance of Visual Studio with the Roslyn Preview extension loaded.
   * In the second Visual Studio instance that you just started, create a new Visual Basic Console Application project. Hover over the token with a wavy underline, and the warning text provided by a Diagnostic appears.  
       
     If you don’t see a wavy underline, make sure that the Roslyn Preview extension is enabled under Tools -> Extensions and Updates. If the Roslyn Preview extension does not show up there, you may still need to run the ‘Install Roslyn Preview into Roslyn Experimental Hive.exe’ installer from the SDK Preview .zip file.  
       
     This Diagnostic is provided by the AnalyzeSymbol method in the debugger project. So initially, the debugger project contains enough code to create a Diagnostic for every type declaration in a Visual Basic file whose identifier contains lowercase letters.  
       
     
   * Now that you’ve seen the initial Diagnostic in action, close the second Visual Studio instance and return to your Diagnostic project.
2. Take a moment to familiarize yourself with the Diagnostic Analyzer in the DiagnosticAnalyzer.vb file of your project. There are two important aspects to draw your attention to:
   * Every Diagnostic Analyzer must provide an <ExportDiagnosticAnalyzer> attribute that describes important details, such as the Diagnostic ID and the language it operates on. For now, you must also provide the <DiagnosticAnalyzer> attribute – this is a known issue.
   * Every Diagnostic Analyzer must implement one or more interfaces that implement the IDiagnosticAnalyzer interface. In this case, the template implemented the ISymbolAnalyzer interface by default. This interface requires a method called AnalyzeSymbol that is called whenever a symbol declaration is changed or added within the target code.
3. There are various ways to implement our analyzer to find local variables that could be constant. One straightforward way is to visit the syntax nodes for local declarations one at a time, ensuring their initializers have constant values, using the ISyntaxNodeAnalyzer(Of TSyntaxKind) interface. To start:
   * Change the interface that the DiagnosticAnalyzer type implements from ISymbolAnalyzer to ISyntaxNodeAnalyzer(Of SyntaxKind). Delete the TODO comment above the Implements clause.
   * Delete the existing member implementations of SymbolKindsOfInterest and AnalyzeSymbol, which no longer apply.
   * Click on the red squiggle on the Implements clause that’s complaining about the missing implementation of ISyntaxNodeAnalyzer(Of SyntaxKind). Hit Ctrl+. and select Implement Interface to add stub implementations of SyntaxKindsOfInterest and AnalyzeNode.
   * Specify that LocalDeclarationStatement is the particular SyntaxKind of interest by implementing the SyntaxKindsOfInterest property.

Return ImmutableArray.Create(SyntaxKind.LocalDeclarationStatement)

* + Update the Diagnostic metadata in the Friend Const strings near the top of the type to match the const rule.

Friend Const DiagnosticId = "MakeConst"

Friend Const Description = "Make Constant"

Friend Const MessageFormat = "Can be made constant"

Friend Const Category = "Usage"

* + When you’re finished, the code in DiagnosticAnalyzer.vb should look like the following code.

Imports System.Collections.Immutable

Imports Microsoft.CodeAnalysis.Diagnostics

<DiagnosticAnalyzer>

<ExportDiagnosticAnalyzer(DiagnosticAnalyzer.DiagnosticId, LanguageNames.VisualBasic)>

Public Class DiagnosticAnalyzer

Implements ISyntaxNodeAnalyzer(Of SyntaxKind)

Friend Const DiagnosticId = "MakeConst"

Friend Const Description = "Make Constant"

Friend Const MessageFormat = "Can be made constant"

Friend Const Category = "Usage"

Friend Shared Rule As New DiagnosticDescriptor(DiagnosticId, Description, MessageFormat, Category, DiagnosticSeverity.Warning)

Public ReadOnly Property SupportedDiagnostics As ImmutableArray(Of DiagnosticDescriptor) Implements IDiagnosticAnalyzer.SupportedDiagnostics

Get

Return ImmutableArray.Create(Rule)

End Get

End Property

Public ReadOnly Property SyntaxKindsOfInterest As ImmutableArray(Of SyntaxKind) Implements ISyntaxNodeAnalyzer(Of SyntaxKind).SyntaxKindsOfInterest

Get

Return ImmutableArray.Create(SyntaxKind.LocalDeclarationStatement)

End Get

End Property

Public Sub AnalyzeNode(node As SyntaxNode, semanticModel As SemanticModel, addDiagnostic As Action(Of Diagnostic), cancellationToken As CancellationToken) Implements ISyntaxNodeAnalyzer(Of SyntaxKind).AnalyzeNode

Throw New NotImplementedException()

End Sub

End Class

* + Now you’re ready to write the logic to determine whether a local variable can be declared as a Const in the AnalyzeNode method.

1. First, you’ll need to perform the necessary syntactic analysis.
   * In the AnalyzeNode method, cast the node passed in to the LocalDeclarationStatementSyntax type. You can safely assume this cast will succeed because the SyntaxKindsOfInterest property now declares that your Diagnostic Analyzer only operates on syntax nodes of that type.

Dim localDeclaration = CType(node, LocalDeclarationStatementSyntax)

* + Ensure that the local variable declaration only has a Dim modifier. We’ll return early here without surfacing a diagnostic if the variable is already declared as a constant.

' Only consider local variable declarations that are Dim (no Static or Const).

If Not localDeclaration.Modifiers.All(Function(m) m.VisualBasicKind() = SyntaxKind.DimKeyword) Then

Return

End If

1. Next, you’ll perform some semantic analysis using the SemanticModel argument to determine whether the local variable declaration can be made Const. A SemanticModel is a representation of all semantic information in a single source file. Please see the [.NET Compiler Platform Project Overview](http://go.microsoft.com/fwlink/?LinkId=230702) for a more detailed description of semantic models.
   * Ensure that every variable in the declaration has an initializer. This is necessary to match the Visual Basic specification which states that all Const variables must be initialized. For example, Dim x As Integer = 0, y As Integer = 1 can be made Const, but Dim x As Integer, y As Integer = 1 cannot. Additionally, use the SemanticModel to ensure that each variable’s initializer is a compile-time constant. You’ll do this by calling SemanticModel.GetConstantValue() for each variable’s initializer and checking that the returned Optional(Of Object) contains a value.

' Ensure that all variable declarators in the local declaration have

' initializers and a single variable name. Additionally, ensure that

' each variable is assigned with a constant value.

For Each declarator In localDeclaration.Declarators

If declarator.Initializer Is Nothing OrElse declarator.Names.Count <> 1 Then

Return

End If

If Not semanticModel.GetConstantValue(declarator.Initializer.Value).HasValue Then

Return

End If

Next

* + Use the SemanticModel to perform data flow analysis on the local declaration statement. Then, use the results of this data flow analysis to ensure that none of the local variables are written with a new value anywhere else. You’ll do this by calling SemanticModel.GetDeclaredSymbol to retrieve the ILocalSymbol for each variable and checking that it isn’t contained with the WrittenOutside collection of the data flow analysis.

' Perform data flow analysis on the local declaration.

Dim dataFlowAnalysis = semanticModel.AnalyzeDataFlow(localDeclaration)

' Retrieve the local symbol for each variable in the local declaration

' and ensure that it is not written outside of the data flow analysis region.

For Each declarator In localDeclaration.Declarators

Dim variable = declarator.Names.Single()

Dim variableSymbol = semanticModel.GetDeclaredSymbol(variable)

If dataFlowAnalysis.WrittenOutside.Contains(variableSymbol) Then

Return

End If

Next

1. With all of the necessary analysis performed, you can create a new Diagnostic object that represents a warning for the non-Const variable declaration. This Diagnostic will get its metadata from the static Rule template defined above. You report this Diagnostic by passing it to the addDiagnostic delegate that was passed in as an argument.

addDiagnostic(Diagnostic.Create(Rule, node.GetLocation()))

At this point, your AnalyzeNode method should look like so:

Public Sub AnalyzeNode(node As SyntaxNode, semanticModel As SemanticModel, addDiagnostic As Action(Of Diagnostic), cancellationToken As CancellationToken) Implements ISyntaxNodeAnalyzer(Of SyntaxKind).AnalyzeNode

Dim localDeclaration = CType(node, LocalDeclarationStatementSyntax)

' Only consider local variable declarations that are Dim (no Static or Const).

If Not localDeclaration.Modifiers.All(Function(m) m.VisualBasicKind() = SyntaxKind.DimKeyword) Then

Return

End If

' Ensure that all variable declarators in the local declaration have

' initializers and a single variable name. Additionally, ensure that

' each variable is assigned with a constant value.

For Each declarator In localDeclaration.Declarators

If declarator.Initializer Is Nothing OrElse declarator.Names.Count <> 1 Then

Return

End If

If Not semanticModel.GetConstantValue(declarator.Initializer.Value).HasValue Then

Return

End If

Next

' Perform data flow analysis on the local declaration.

Dim dataFlowAnalysis = semanticModel.AnalyzeDataFlow(localDeclaration)

' Retrieve the local symbol for each variable in the local declaration

' and ensure that it is not written outside the data flow analysis region.

For Each declarator In localDeclaration.Declarators

Dim variable = declarator.Names.Single()

Dim variableSymbol = semanticModel.GetDeclaredSymbol(variable)

If dataFlowAnalysis.WrittenOutside.Contains(variableSymbol) Then

Return

End If

Next

addDiagnostic(Diagnostic.Create(Rule, node.GetLocation()))

End Sub

1. Press Ctrl+F5 to run the Diagnostic project in a new instance of Visual Studio with the Roslyn Language Service loaded.
   * In the second Visual Studio instance create a new Visual Basic Console Application project and add a few local variable declarations initialized with constant values to the Main method.

Sub Main()

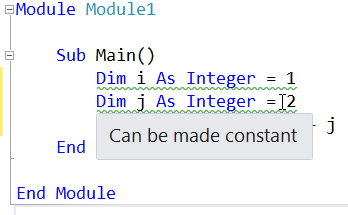
Dim i As Integer = 1

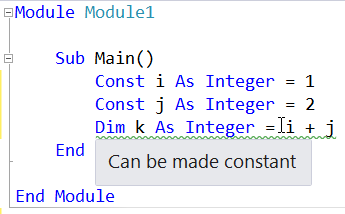
Dim j As Integer = 2

Dim k As Integer = i + j

End Sub

* + You’ll see that they are reported as warnings as below.



* + Notice that if you type Const before each variable, the warnings are automatically removed. Additionally, changing a variable to Const can affect the reporting of other variables.  
      
    

1. Congratulations! You’ve created your first Diagnostic using the .NET Compiler Platform APIs to perform non-trivial syntactic and semantic analysis.

# Writing the Code Fix

Any Diagnostic can provide one or more Code Fixes which define an edit that can be performed to the source code to address the reported issue. For the Diagnostic that you just created, you can provide a Code Fix that replaces Dim with the Const keyword when the user chooses it from the light bulb UI in the editor. To do so, follow the steps below.

1. First, open the CodeFixProvider.vb file that was already added by the Diagnostic with Code Fix template. This Code Fix is already wired up to the Diagnostic ID produced by your Diagnostic Analyzer, but it doesn’t yet implement the right code transform.
2. Delete the MakeUppercaseAsync method, which no longer applies.
3. In GetFixesAsync, change the ancestor node type you’re searching for to LocalDeclarationStatementSyntax to match the Diagnostic.

' Find the type statement identified by the diagnostic.

Dim declaration = root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType(Of LocalDeclarationStatementSyntax)().First()

1. Change the last line that creates the CodeAction object to call a MakeConstAsync method that you’ll be defining next and remove the TODO comment. Each CodeAction represents a fix that users can choose to apply in Visual Studio.

Return {CodeAction.Create("Make constant", Function(c) MakeConstAsync(document, declaration, c))}

1. At this point, your code should look like so:

Imports Microsoft.CodeAnalysis.Rename

<ExportCodeFixProvider(DiagnosticAnalyzer.DiagnosticId, LanguageNames.VisualBasic)>

Friend Class CodeFixProvider

Implements ICodeFixProvider

Public Function GetFixableDiagnosticIds() As IEnumerable(Of String) Implements ICodeFixProvider.GetFixableDiagnosticIds

Return {DiagnosticAnalyzer.DiagnosticId}

End Function

Public Async Function GetFixesAsync(document As Document, span As TextSpan, diagnostics As IEnumerable(Of Diagnostic), cancellationToken As CancellationToken) As Task(Of IEnumerable(Of CodeAction)) Implements ICodeFixProvider.GetFixesAsync

Dim root = Await document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(False)

Dim diagnosticSpan = diagnostics.First().Location.SourceSpan

' Find the type statement identified by the diagnostic.

Dim declaration = root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType(Of LocalDeclarationStatementSyntax)().First()

' Return a code action that will invoke the fix.

Return {CodeAction.Create("Make constant", Function(c) MakeConstAsync(document, declaration, c))}

End Function

End Class

1. Now it’s time to implement the MakeConstAsync method, which will transform the original Document into the fixed Document.
   * First, declare a MakeConstAsync method with the following signature. This method will transform the Document representing the user’s source file into a fixed Document that now contains a Const declaration.

Private Async Function MakeConstAsync(document As Document, localDeclaration As LocalDeclarationStatementSyntax, cancellationToken As CancellationToken) As Task(Of Document)

* + Then, create a new Const keyword token that will replace the first token of the local declaration. Be careful to first remove any trivia from the first token of the local declaration and attach it to the Const token.

' Create a const token with the leading trivia from the local declaration.

Dim firstToken = localDeclaration.GetFirstToken()

Dim constToken = SyntaxFactory.Token(

firstToken.LeadingTrivia, SyntaxKind.ConstKeyword, firstToken.TrailingTrivia)

* + Next, create a new SyntaxTokenList containing just the Const token.

' Create a new modifier list with the const token.

Dim newModifiers = SyntaxFactory.TokenList(constToken)

* + Create a new local declaration containing the new list of modifiers.

' Produce new local declaration.

Dim newLocalDeclaration = localDeclaration.WithModifiers(newModifiers)

* + Add a Formatter syntax annotation to the new declaration statement, which is an indicator to the Code Fix engine to format any whitespace using the Visual Basic formatting rules. You will need to hit Ctrl+. on the Formatter type to add a using statement for the Microsoft.CodeAnalysis.Formatting namespace.

' Add an annotation to format the new local declaration.

Dim formattedLocalDeclaration = newLocalDeclaration.WithAdditionalAnnotations(Formatter.Annotation)

* + Retrieve the root SyntaxNode from the Document and use it to replace the old declaration statement with the new one.

' Replace the old local declaration with the new local declaration.

Dim oldRoot = Await document.GetSyntaxRootAsync(cancellationToken)

Dim newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocalDeclaration)

* + Finally, return a new Document containing the updated syntax root, representing the result of the tree transformation that you just performed.

' Return document with transformed tree.

Return document.WithSyntaxRoot(newRoot)

* + At this point, your MakeConstAsync method should be like so:

Private Async Function MakeConstAsync(document As Document, localDeclaration As LocalDeclarationStatementSyntax, cancellationToken As CancellationToken) As Task(Of Document)

' Create a const token with the leading trivia from the local declaration.

Dim firstToken = localDeclaration.GetFirstToken()

Dim constToken = SyntaxFactory.Token(

firstToken.LeadingTrivia, SyntaxKind.ConstKeyword, firstToken.TrailingTrivia)

' Create a new modifier list with the const token.

Dim newModifiers = SyntaxFactory.TokenList(constToken)

' Produce new local declaration.

Dim newLocalDeclaration = localDeclaration.WithModifiers(newModifiers)

' Add an annotation to format the new local declaration.

Dim formattedLocalDeclaration = newLocalDeclaration.WithAdditionalAnnotations(Formatter.Annotation)

' Replace the old local declaration with the new local declaration.

Dim oldRoot = Await document.GetSyntaxRootAsync(cancellationToken)

Dim newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocalDeclaration)

' Return document with transformed tree.

Return document.WithSyntaxRoot(newRoot)

End Function

1. Press Ctrl+F5 to run the Diagnostic project in a second instance of Visual Studio with the Roslyn Preview extension loaded.
   * In the second Visual Studio instance, create a new Visual Basic Console Application project and, like before, add a few local variable declarations initialized with to constant values in the Main method.

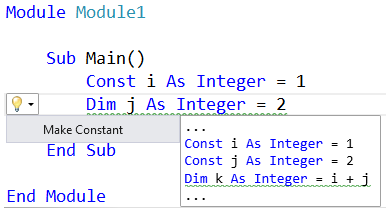
Sub Main()

Dim i As Integer = 1

Dim j As Integer = 2

Dim k As Integer = i + j

End Sub

* + You’ll see that they are reported as warnings and “light bulb” suggestions appear next to them when the editor caret is on the same line.
  + Move the editor caret to one of the squiggly underlines and press Ctrl+. to display the suggestion. Notice that a preview window appears next to the suggestion menu showing what the code will look like after the Code Fix is invoked.  
      
    

1. Congratulations! You’ve created your first .NET Compiler Platform extension that performs on-the-fly code analysis to detect an issue and provides a quick fix to correct it.