Microsoft® “Roslyn”

How To: Write a C# Quick Fix

June 2012

# Prerequisites

* [Roslyn Project Overview](http://go.microsoft.com/fwlink/?LinkId=230702)

# Introduction

In previous releases of Visual Studio, it has always been difficult to create custom refactorings that target C# or Visual Basic. With the Project Roslyn Services APIs, 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 smart tags and showing rich previews is all done for you automatically.

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

# Writing the Code Issue

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:

int x = 0;

Console.WriteLine(x);

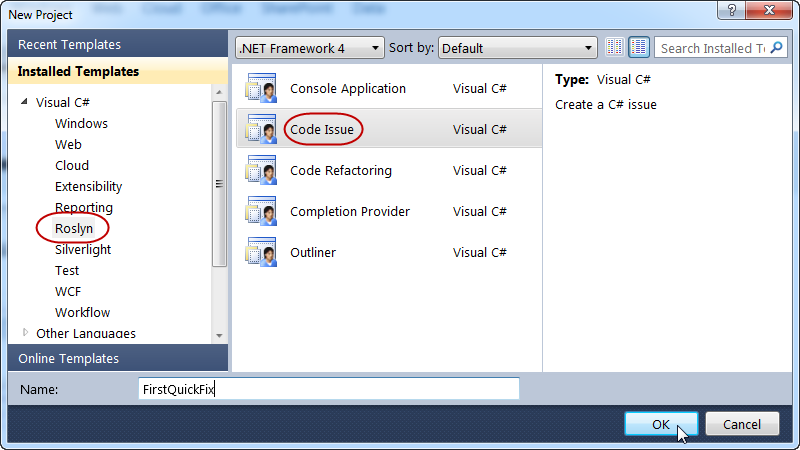
In the code above, x is assigned a constant value and is never written to. Thus, it can be declared using the const modifier:

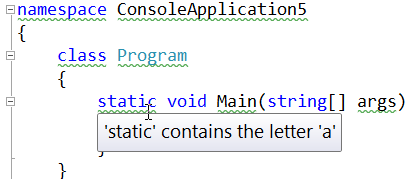
const int x = 0;

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 Roslyn Compiler APIs and exposing it as a Code Issue is pretty easy.

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



1. Press F5 to run the newly created Code Issue project in a new instance of Visual Studio with the Roslyn Language Service loaded. This new instance of Visual Studio is called the debuggee.
   * In the debuggee Visual Studio instance, create a new C# Console Application project. Hover over one of the tokens in the code with a wavy underline and the warning text provided by a code issue appears. These code issues are provided by the GetIssues method in the debugger project. So initially, the debugger project contains enough code to create a Code Issue for every token in a C# file that contains the letter ‘a’.  
       
     
   * Now that you’ve seen the initial Code Issue in action, close the debuggee Visual Studio instance and return to your Code Issue project.
2. Take a moment to familiarize yourself with the Code Issue Provider the CodeIssueProvider.cs file of your project. There are two important aspects to draw your attention to:
   * Every ICodeIssueProvider implementer must provide at least one Export attribute that describes important details about the provider. There are three possible attributes that can be used, depending on what your Code Issue Provider operates on: ExportSyntax**Node**CodeIssueProvider, ExportSyntax**Token**CodeIssueProvider and ExportSyntaxT**rivia**CodeIssueProvider. In this case, the Code Issue Provider operates on nodes so it provides an ExportSyntax**Node**CodeIssueProvider attribute.
   * Every Code Issue Provider must implement the ICodeIssueProvider interface. This interface contains three overloaded methods called GetIssues: one each for operating on syntax nodes, tokens or trivia. The Export attribute(s) determine which of these methods are called. Because this Code Issue Provider provides an ExportSyntaxNodeCodeIssueProvider attribute, only the method operating on nodes is implemented. The others throw a NotImplementedException.

**Note**: For a full treatment of syntax nodes, tokens and trivia please refer to the [Roslyn Project Overview](http://go.microsoft.com/fwlink/?LinkId=230702).

1. To restrict the Code Issue Provider to only operate on local variables, update the existing Code Issue Provider to operate on LocalDeclarationStatementSyntax nodes:
   * Add an argument, ’typeof(LocalDeclarationStatementSyntax)‘ to the end of the ExportSyntaxNodeCodeIssueProvider attribute’s argument list.
   * Remove the source code from inside the GetIssues method that operates on syntax nodes.
   * When you’re finished, the code in CodeIssueProvider.cs should look like the following code.

using System;

using System.Collections.Generic;

using System.ComponentModel.Composition;

using System.Linq;

using System.Threading;

using Roslyn.Compilers;

using Roslyn.Compilers.Common;

using Roslyn.Compilers.CSharp;

using Roslyn.Services;

using Roslyn.Services.Editor;

namespace FirstQuickFix

{

[ExportSyntaxNodeCodeIssueProvider("FirstQuickFix", LanguageNames.CSharp,

typeof(LocalDeclarationStatementSyntax))]

public class CodeIssueProvider : ICodeIssueProvider

{

public IEnumerable<CodeIssue> GetIssues(IDocument document,

CommonSyntaxNode node, CancellationToken cancellationToken)

{

}

#region Unimplemented ICodeIssueProvider members

public IEnumerable<CodeIssue> GetIssues(IDocument document,

CommonSyntaxToken token, CancellationToken cancellationToken)

{

throw new NotImplementedException();

}

public IEnumerable<CodeIssue> GetIssues(IDocument document,

CommonSyntaxTrivia trivia, CancellationToken cancellationToken)

{

throw new NotImplementedException();

}

#endregion

}

}

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

1. First, you’ll need to perform the necessary syntactic analysis.
   * In the GetIssues method that operates on syntax nodes, cast the node passed in to a LocalDeclarationStatementSyntax type. You can safely assume this cast will succeed because your ExportSyntaxNodeCodeIssueProvider declares that this is the syntax node type that your Code Issue Provider operates on.

var localDeclaration = (LocalDeclarationStatementSyntax)node;

* + Ensure that the local variable declaration doesn’t already have the const modifier.

// Only consider local variable declarations that aren't already const.

if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))

{

return null;

}

1. Next, you’ll perform some semantic analysis to determine whether the local variable declaration can be made const.
   * Retrieve an ISemanticModel for the IDocument that the Code Issue Provider is operating on.

**Note**: An IDocument is analogous to a file in Visual Studio, and an ISemanticModel is a representation of all semantic information in a single source file. Please see the [Roslyn Project Overview](http://go.microsoft.com/fwlink/?LinkId=230702) for a more detailed treatment of these concepts.

var semanticModel = document.GetSemanticModel(cancellationToken);

* + Next, ensure that every variable in the declaration has an initializer. This is necessary to match the C# specification which states that all const variables must be initialized. For example, int x = 0, y = 1; can be made const, but int x, y = 1; cannot. Additionally, use the ISemanticModel to ensure that each variable’s initializer is a compile-time constant. You’ll do this by calling ISemanticModel.GetConstantValue() for each variable’s initializer and checking that the returned Optional<object> contains a value.

// Ensure that all variables in the local declaration have initializers that

// are assigned with constant values.

foreach (var variable in localDeclaration.Declaration.Variables)

{

var initializer = variable.Initializer;

if (initializer == null)

{

return null;

}

var constantValue = semanticModel.GetConstantValue(initializer.Value);

if (!constantValue.HasValue)

{

return null;

}

}

* + Use the ISemanticModel 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 ISemanticModel.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.

var dataFlowAnalysis = semanticModel.AnalyzeStatementDataFlow(localDeclaration);

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

// and ensure that it is not written outside of the data flow analyis region.

foreach (var variable in localDeclaration.Declaration.Variables)

{

var variableSymbol = semanticModel.GetDeclaredSymbol(variable);

if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))

{

return null;

}

}

1. With all of the necessary analysis performed, you can return a new CodeIssue that reports a warning for the variable declaration.

return new[]

{

new CodeIssue(CodeIssue.Severity.Warning, localDeclaration.Span,

"Can be made constant")

};

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

public IEnumerable<CodeIssue> GetIssues(IDocument document,

CommonSyntaxNode node, CancellationToken cancellationToken)

{

var localDeclaration = (LocalDeclarationStatementSyntax)node;

// Only consider local variable declarations that aren't already const.

if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))

{

return null;

}

var semanticModel = document.GetSemanticModel(cancellationToken);

// Ensure that all variables in the local declaration have initializers that

// are assigned with constant values.

foreach (var variable in localDeclaration.Declaration.Variables)

{

var initializer = variable.Initializer;

if (initializer == null)

{

return null;

}

var constantValue = semanticModel.GetConstantValue(initializer.Value);

if (!constantValue.HasValue)

{

return null;

}

}

// Perform data flow analysis on the local declaration.

var dataFlowAnalysis = semanticModel.AnalyzeStatementDataFlow(localDeclaration);

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

// and ensure that it is not written outside of the data flow analyis region.

foreach (var variable in localDeclaration.Declaration.Variables)

{

var variableSymbol = semanticModel.GetDeclaredSymbol(variable);

if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))

{

return null;

}

}

return new[]

{

new CodeIssue(CodeIssue.Severity.Warning, localDeclaration.Span,

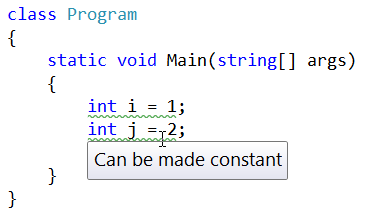
"Can be made constant")

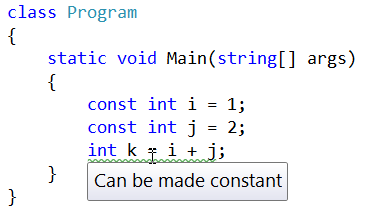
};

}

1. Press F5 to run the Code Issue project in a new instance of Visual Studio with the Roslyn Language Service loaded.
   * In the debuggee Visual Studio instance create a new C# Console Application project and add a few local variable declarations assigned to constant values in the Main method.

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



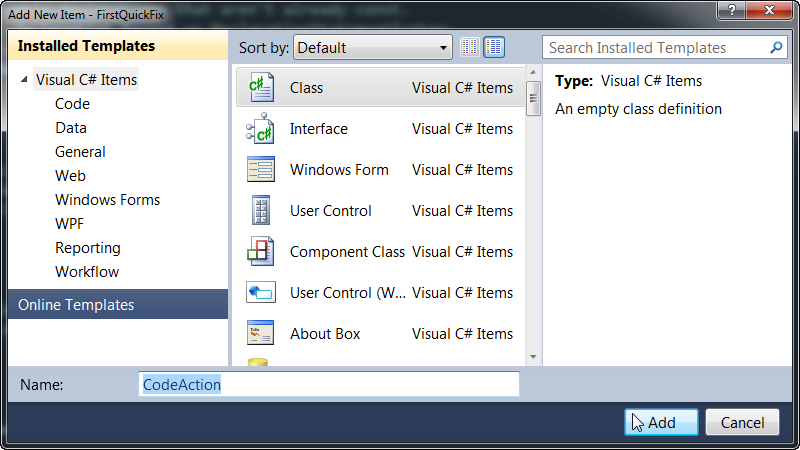
* + Notice that if you type const before each variable, the warnings are automatically removed. Additionally, changing a variable to const can affect the reporting of other variables.  
      
    

1. Congratulations! You’ve created your first code issue using the Roslyn APIs to perform non-trivial syntactic and semantic analysis.

# Writing the Code Action

Any Code Issue can provide one or more Code Actions which define an edit that can be performed to the source code to address the reported issue. For the Code Issue that you just created, you can provide a Code Action that inserts the const keyword when the user chooses it from a smart tag in the editor. To do so, follow the steps below.

1. First, add a new Class implementing ICodeAction to your Code Issue project.
   * In the Solution Explorer, right-click on your Code Issue project and choose Add -> Class… from the context menu to display the Add New Item dialog.
   * Name the new file “CodeAction” and click Add.



1. At the top of the file, add the following using directives after “using System.Text;”.

using System.Threading;

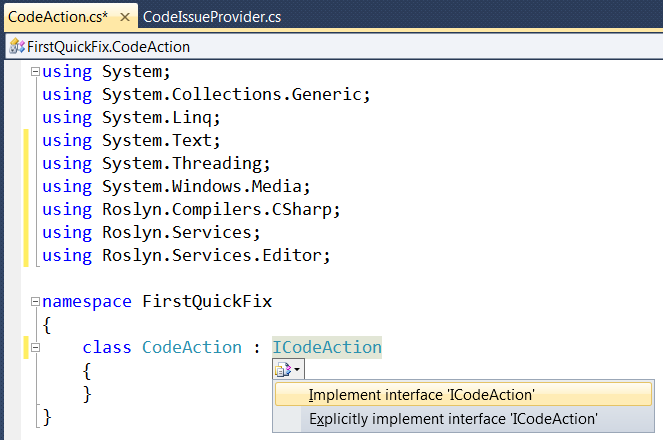
using System.Windows.Media;

using Roslyn.Compilers;

using Roslyn.Compilers.CSharp;

using Roslyn.Services;

using Roslyn.Services.Editor;

1. Implement the ICodeAction interface on your newly created class. After typing “ICodeAction” press CTRL+. to display the smart tag beneath. Press ENTER to generate a default implementation of ICodeAction.  
     
   
2. Remove the default value for the cancellationToken argument of GetEdit.
3. Change the Icon property to return null, and change the Description property to return "Make Constant".
4. At this point, your code should look like so:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Windows.Media;

using Roslyn.Compilers;

using Roslyn.Compilers.CSharp;

using Roslyn.Services;

using Roslyn.Services.Editor;

namespace FirstQuickFix

{

class CodeAction : ICodeAction

{

public string Description

{

get { return "Make Constant"; }

}

public CodeActionEdit GetEdit(CancellationToken cancellationToken)

{

throw new NotImplementedException();

}

public ImageSource Icon

{

get { return null; }

}

}

}

1. In order to implement the GetEdit method, you’ll need to provide a couple of objects to the Code Action.
   * Return to your Code Issue Provider in CodeIssueProvider.cs and update the code at the bottom of GetIssues that returns a new CodeIssue to also include a new instance of your CodeAction class, passing document and localDeclaration.

return new[]

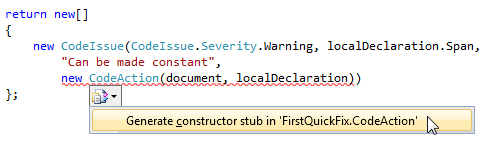
{

new CodeIssue(CodeIssue.Severity.Warning, localDeclaration.Span,

"Can be made constant",

new CodeAction(document, localDeclaration))

};

* + Invoke the smart tag using CTRL+. on CodeAction to generate a new constructor and fields in the CodeAction class.  
      
    

1. Now, return to the CodeAction.cs file and implement the GetEdit method.
   * First, create a new const keyword token that will be inserted at the front of the declaration statement. Be careful to first remove any leading trivia from the first token of the declaration statement and attach it to the const token.

// Remove the leading trivia from the local declaration.

var firstToken = localDeclaration.GetFirstToken();

var leadingTrivia = firstToken.LeadingTrivia;

var trimmedLocal = localDeclaration.ReplaceToken(

firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the original leading trivia.

var constToken = Syntax.Token(leadingTrivia, SyntaxKind.ConstKeyword);

* + Next, create a new SyntaxTokenList containing the const token and the existing modifiers of the declaration statement.

// Insert the const token into the modifiers list, creating a new modifiers list.

var newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);

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

// Produce the new local declaration.

var newLocal = trimmedLocal.WithModifiers(newModifiers);

* + Add a Formatting syntax annotation to the new declaration statement, which is an indicator to the code action engine to format any whitespace using the C# formatting rules.

// Add an annotation to format the new local declaration.

var formattedLocal = CodeAnnotations.Formatting.AddAnnotationTo(newLocal);

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

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

var oldRoot = document.GetSyntaxRoot(cancellationToken);

var newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

* + Finally, create a new CodeActionEdit passing a document updated with the tree transformation that you just performed.

// Create and return a new CodeActionEdit for the transformed tree.

return new CodeActionEdit(document.UpdateSyntaxRoot(newRoot));

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

public ICodeActionEdit GetEdit(CancellationToken cancellationToken)

{

// Remove the leading trivia from the local declaration.

var firstToken = localDeclaration.GetFirstToken();

var leadingTrivia = firstToken.LeadingTrivia;

var trimmedLocal = localDeclaration.ReplaceToken(

firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.

var constToken = Syntax.Token(leadingTrivia, SyntaxKind.ConstKeyword);

// Insert the const token into the modifiers list, creating a new modifiers list.

var newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);

// Produce the new local declaration.

var newLocal = trimmedLocal.WithModifiers(newModifiers);

// Add an annotation to format the new local declaration.

var formattedLocal = CodeAnnotations.Formatting.AddAnnotationTo(newLocal);

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

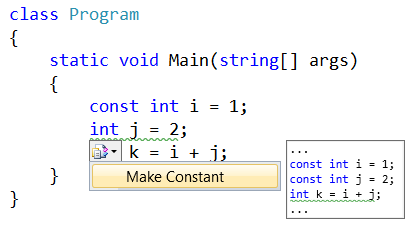
var oldRoot = document.GetSyntaxRoot(cancellationToken);

var newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Create and return a new CodeActionEdit for the transformed tree.

return new CodeActionEdit(document.UpdateSyntaxRoot(newRoot));

}

1. Press F5 to run the Code Issue project in a new instance of Visual Studio with the Roslyn Language Service loaded.
   * In the debuggee Visual Studio instance create a new C# Console Application project and add a few local variable declarations assigned to constant values in the Main method. You’ll see that they are reported as warnings and smart tags appear for them.
   * Move the editor caret to one of the squiggly underlines and press CTRL+. to display the smart tag. Notice that a preview window appears next to the smart tag menu showing what the code will look like after the Code Action is invoked. Your Code Issue is even run on the preview window to show that a different variable can be made const once the Code Action is applied.  
       
     

# Fixing Bugs

Sadly, there are a few bugs in the implementation.

1. The Code Issue Provider’s GetIssues method does not check to see if the constant value is actually convertible to the variable type. So, the current implementation will happily convert an incorrect declaration such as ‘int i = "abc"’ to a local constant.
2. Reference types are not handled properly. The only constant value allowed for a reference type is null, except in this case of System.String, which allows string literals. In other words, ‘const string s = "abc"’ is legal, but ‘const object s = "abc"’ is not.
3. If a variable is declared with the “var” keyword, the Code Action does the wrong thing and generates a “const var” declaration, which is not supported by the C# language. To fix this bug, the code action must replace the “var” keyword with the inferred type’s name.

Fortunately, both of these bugs can be addressed using the same techniques that you just learned.

1. To fix the first bug, first open CodeIssueProvider.cs and locate the foreach loop where each of the local declaration’s initializers are checked to ensure that they’re assigned with constant values.
   * Immediately before the foreach loop, call ISemanicModel.GetTypeInfo() to retrieve detailed information about the declared type of the local declaration:

var variableTypeName = localDeclaration.Declaration.Type;

var variableType = semanticModel.GetTypeInfo(variableTypeName).ConvertedType;

* + Next, add the following code before the closing brace of the foreach loop to call ISemanticModel.ClassifyConversion() and determine whether the initializer is convertible to the local declaration type. If there is no conversion, or the conversion is user-defined, the variable can’t be a local constant.

// Ensure that the initializer value can be converted to the type of the

// local declaration without a user-defined conversion.

var conversion = semanticModel.ClassifyConversion(initializer.Value, variableType);

if (!conversion.Exists || conversion.IsUserDefined)

{

return null;

}

1. The next bug fix builds upon the last one.
   * Before the closing brace of the same foreach loop, add the following code to check the type of local declaration when the constant is a string or null.

// Special cases:

// \* If the constant value is a string, the type of the local declaration

// must be System.String.

// \* If the constant value is null, the type of the local declaration must

// be a reference type.

if (constantValue.Value is string)

{

if (variableType.SpecialType != SpecialType.System\_String)

{

return null;

}

}

else if (variableType.IsReferenceType && constantValue.Value != null)

{

return null;

}

* + With this code in place, the GetIssues() method should look like so.

public IEnumerable<CodeIssue> GetIssues(IDocument document,

CommonSyntaxNode node, CancellationToken cancellationToken)

{

var localDeclaration = (LocalDeclarationStatementSyntax)node;

// Only consider local variable declarations that aren't already const.

if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))

{

return null;

}

var semanticModel = document.GetSemanticModel(cancellationToken);

var variableTypeName = localDeclaration.Declaration.Type;

var variableType = semanticModel.GetTypeInfo(variableTypeName).ConvertedType;

// Ensure that all variables in the local declaration have initializers that

// are assigned with constant values.

foreach (var variable in localDeclaration.Declaration.Variables)

{

var initializer = variable.Initializer;

if (initializer == null)

{

return null;

}

var constantValue = semanticModel.GetConstantValue(initializer.Value);

if (!constantValue.HasValue)

{

return null;

}

// Ensure that the initializer value can be converted to the type of the

// local declaration without a user-defined conversion.

var conversion = semanticModel.ClassifyConversion(

initializer.Value, variableType);

if (!conversion.Exists || conversion.IsUserDefined)

{

return null;

}

// Special cases:

// \* If the constant value is a string, the type of the local declaration

// must be System.String.

// \* If the constant value is null, the type of the local declaration must

// be a reference type.

if (constantValue.Value is string)

{

if (variableType.SpecialType != SpecialType.System\_String)

{

return null;

}

}

else if (variableType.IsReferenceType && constantValue.Value != null)

{

return null;

}

}

// Perform data flow analysis on the local declaration.

var dataFlowAnalysis = semanticModel.AnalyzeStatementDataFlow(localDeclaration);

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

// and ensure that it is not written outside of the data flow analyis region.

foreach (var variable in localDeclaration.Declaration.Variables)

{

var variableSymbol = semanticModel.GetDeclaredSymbol(variable);

if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))

{

return null;

}

}

return new[]

{

new CodeIssue(CodeIssue.Severity.Warning, localDeclaration.Span,

"Can be made constant",

new CodeAction(editFactory, document, localDeclaration))

};

}

1. Fixing the third issue requires a little more code to replace the ‘var’ keyword with the correct type name.
   * Return to CodeAction.cs and add replace the code at the comment which reads “Produce the local variable declaration” with the following code:

// If the type of the declaration is 'var', create a new type name

// for the inferred type.

var variableDeclaration = localDeclaration.Declaration;

var variableTypeName = variableDeclaration.Type;

if (variableTypeName.IsVar)

{

}

// Produce the new local declaration.

var newLocal = trimmedLocal.WithModifiers(newModifiers)

.WithDeclaration(variableDeclaration);

* + Next, add a check inside curly braces of the if-block you wrote above to ensure that the type of the variable declaration is not an alias. If it is an alias to some other type (e.g. “using var = System.String;”) then it is legal to declare a local “const var”.

var semanticModel = document.GetSemanticModel();

// Special case: Ensure that 'var' isn't actually an alias to another type

// (e.g. using var = System.String).

var aliasInfo = semanticModel.GetAliasInfo(variableTypeName);

if (aliasInfo == null)

{

}

* + Inside the curly braces that you wrote in the code above, add the following code to retrieve the type inferred for ‘var’ inside the curly braces of the if-block you wrote above.

// Retrieve the type inferred for var.

var type = semanticModel.GetTypeInfo(variableTypeName).ConvertedType;

// Special case: Ensure that 'var' isn't actually a type named 'var'.

if (type.Name != "var")

{

}

* + Now, add the code to create a new TypeSyntax for the inferred type inside the curly braces of the if-block you wrote above.

// Create a new TypeSyntax for the inferred type. Be careful

// to keep any leading and trailing trivia from the var keyword.

var typeName = Syntax.ParseTypeName(type.ToDisplayString())

.WithLeadingTrivia(variableTypeName.GetLeadingTrivia())

.WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

* + Add a NameSimplification syntax annotation to the type name to ensure that the code action engine reduces the type name to its minimally-qualified form.

// Add an annotation to simplify the type name.

var simplifiedTypeName = CodeAnnotations.NameSimplification

.AddAnnotationTo(typeName);

* + Finally, replace the variable declaration’s type with the one you just created.

// Replace the type in the variable declaration.

variableDeclaration = variableDeclaration.WithType(simplifiedTypeName);

* + With this bug fix in place, your GetEdit method should now look like the following:

public ICodeActionEdit GetEdit(CancellationToken cancellationToken)

{

// Remove the leading trivia from the local declaration.

var firstToken = localDeclaration.GetFirstToken();

var leadingTrivia = firstToken.LeadingTrivia;

var trimmedLocal = localDeclaration.ReplaceToken(

firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.

var constToken = Syntax.Token(leadingTrivia, SyntaxKind.ConstKeyword);

// Insert the const token into the modifiers list, creating a new modifiers list.

var newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);

// If the type of declaration is 'var', create a new type name for the

// inferred type.

var variableDeclaration = localDeclaration.Declaration;

var variableTypeName = variableDeclaration.Type;

if (variableTypeName.IsVar)

{

var semanticModel = document.GetSemanticModel();

// Special case: Ensure that 'var' isn't actually an alias to another type

// (e.g. using var = System.String).

var aliasInfo = semanticModel.GetAliasInfo(variableTypeName);

if (aliasInfo == null)

{

// Retrieve the type inferred for var.

var type = semanticModel.GetTypeInfo(variableTypeName).ConvertedType;

// Special case: Ensure that 'var' isn't actually a type named 'var'.

if (type.Name != "var")

{

// Create a new TypeSyntax for the inferred type. Be careful

// to keep any leading and trailing trivia from the var keyword.

var typeName = Syntax.ParseTypeName(type.ToDisplayString())

.WithLeadingTrivia(variableTypeName.GetLeadingTrivia())

.WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

// Add an annotation to simplify the type name.

var simplifiedTypeName = CodeAnnotations.NameSimplification

.AddAnnotationTo(typeName);

// Replace the type in the variable declaration.

variableDeclaration =

variableDeclaration.WithType(simplifiedTypeName);

}

}

}

// Produce the new local declaration.

var newLocal = trimmedLocal.WithModifiers(newModifiers)

.WithDeclaration(variableDeclaration);

// Add an annotation to format the new local declaration.

var formattedLocal = CodeAnnotations.Formatting.AddAnnotationTo(newLocal);

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

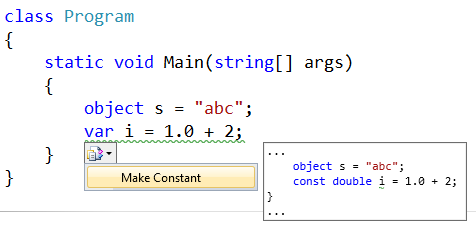
var oldRoot = document.GetSyntaxRoot(cancellationToken);

var newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Create and return a new CodeActionEdit for the transformed tree.

return new CodeActionEdit(document.UpdateSyntaxRoot(newRoot));

}

1. Once again, press F5 to run the Code Issue project in a new instance of Visual Studio with the Roslyn Language Service loaded.
   * In the debuggee Visual Studio instance create a new C# Console Application project and add ‘int x = "abc";’ to the Main method. Thanks to the first bug fix, no warning should be reported for this local variable declaration.
   * Next, add ‘object s = "abc";’ to the Main method. Because of the second bug fix, no warning should be reported.
   * Finally, add another local variable that uses the ‘var’ keyword. You’ll see that a warning is reported and a smart tag appears beneath.
   * Move the editor caret over the squiggly underline and press CTRL+. to display the smart tag. Upon selecting your code action, note that the ‘var’ keyword is now handled correctly.  
       
     
2. Congratulations! You’ve created your first Roslyn extension that performs on-the-fly code analysis to detect an issue and provides a quick fix to correct it.