

How To: Write a C# 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:

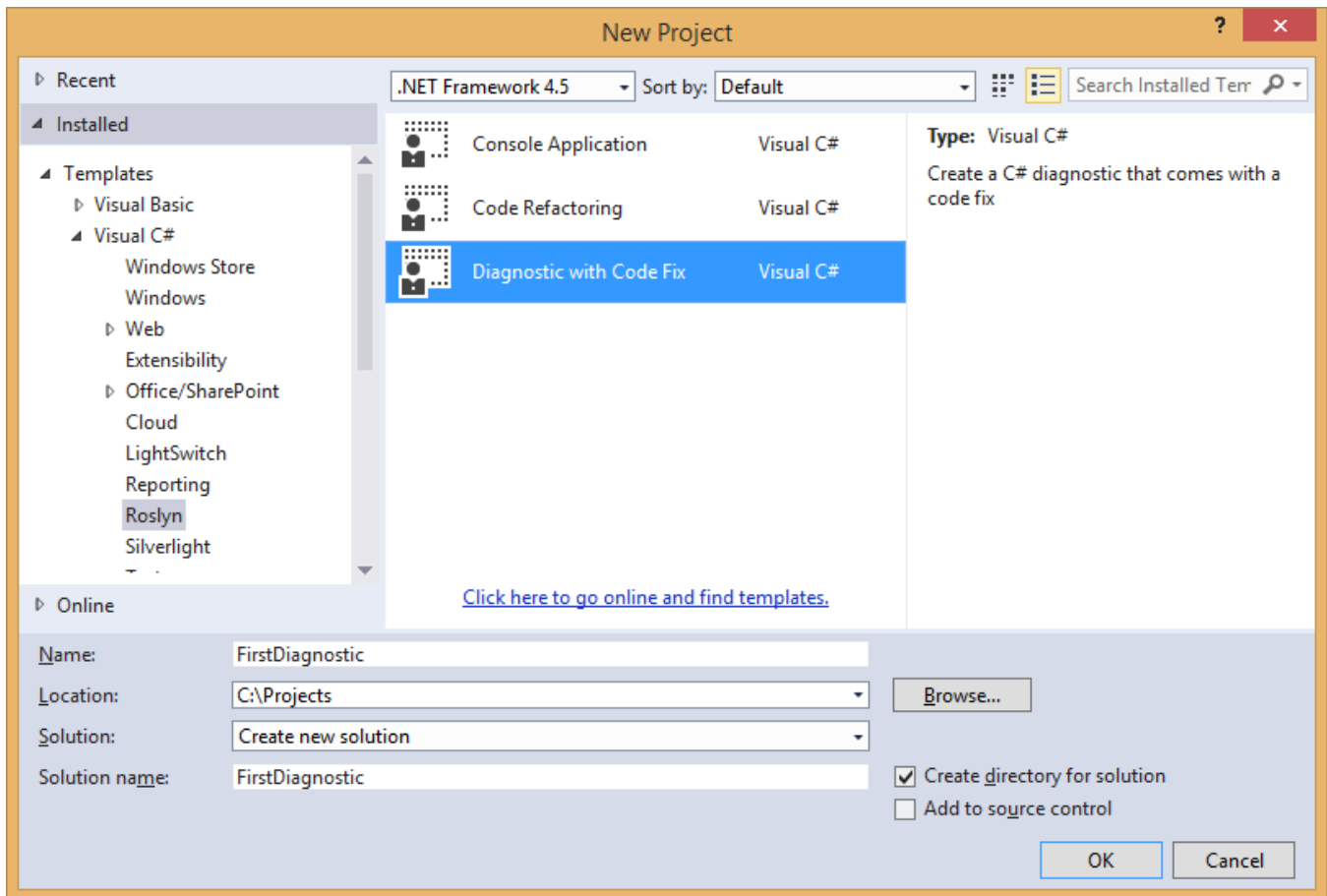
```
int x = 0;
Console.WriteLine(x);
```

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 .NET Compiler Platform and exposing it as a Diagnostic is pretty easy.

1. First, create a new C# Diagnostic project.
 - In Visual Studio, choose File -> New -> Project... to display the New Project dialog.
 - Under Visual C# -> Roslyn, choose "Diagnostic with Code Fix".
 - Name your project "FirstDiagnostic" and click OK.



2. 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 C# 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 C# file whose identifier contains lowercase letters.

```

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}

```

- Now that you've seen the initial Diagnostic in action, close the second Visual Studio instance and return to your Diagnostic project.
3. Take a moment to familiarize yourself with the Diagnostic Analyzer in the DiagnosticAnalyzer.cs 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.
 4. 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<TSyntaxKind> interface. To start:
 - Change the interface that the DiagnosticAnalyzer type implements from ISymbolAnalyzer to ISyntaxNodeAnalyzer<SyntaxKind>. Hit Ctrl+. on the squiggled SyntaxKind type to add a using statement for the Microsoft.CodeAnalysis.CSharp namespace. Delete the TODO comment above the type definition.
 - Delete the existing member implementations of SymbolKindsOfInterest and AnalyzeSymbol, which no longer apply.
 - Click on the red squiggle on DiagnosticAnalyzer that's complaining about the missing implementation of ISyntaxNodeAnalyzer<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 internal const strings near the top of the type to match the const rule.

```
internal const string DiagnosticId = "MakeConst";
internal const string Description = "Make Constant";
internal const string MessageFormat = "Can be made constant";
internal const string Category = "Usage";
```

- When you're finished, the code in DiagnosticAnalyzer.cs should look like the following code.

```
using System;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Linq;
using System.Threading;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.Diagnostics;

namespace FirstDiagnostic
{
    [DiagnosticAnalyzer]
    [ExportDiagnosticAnalyzer(DiagnosticId, LanguageNames.CSharp)]
    public class DiagnosticAnalyzer : ISyntaxNodeAnalyzer<SyntaxKind>
    {
        internal const string DiagnosticId = "MakeConst";
        internal const string Description = "Make Constant";
        internal const string MessageFormat = "Can be made constant";
        internal const string Category = "Usage";

        internal static DiagnosticDescriptor Rule = new
DiagnosticDescriptor(DiagnosticId, Description, MessageFormat, Category,
DiagnosticSeverity.Warning);

        public ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get {
return ImmutableArray.Create(Rule); } }

        public ImmutableArray<SyntaxKind> SyntaxKindsOfInterest
        {
            get
            {
                return ImmutableArray.Create(SyntaxKind.LocalDeclarationStatement);
            }
        }

        public void AnalyzeNode(SyntaxNode node, SemanticModel semanticModel,
Action<Diagnostic> addDiagnostic, CancellationToken cancellationTok
)
        {
            throw new NotImplementedException();
        }
    }
}
```

- Now you're ready to write the logic to determine whether a local variable can be declared as a const in the AnalyzeNode method.
5. 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. Use Ctrl+. to add a using statement for the Microsoft.CodeAnalysis.CSharp.Syntax namespace required here.

```
var localDeclaration = (LocalDeclarationStatementSyntax)node;
```

- Ensure that the local variable declaration doesn't already have the const 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 aren't already const.
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

6. 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](#) for a more detailed description of semantic models.

- 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 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<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;
    }

    var constantValue = semanticModel.GetConstantValue(initializer.Value);
    if (!constantValue.HasValue)
    {
        return;
    }
}
```

- 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.
var 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.
foreach (var variable in localDeclaration.Declaration.Variables)
{
    var variableSymbol = semanticModel.GetDeclaredSymbol(variable);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
```

7. 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 void AnalyzeNode(SyntaxNode node, SemanticModel semanticModel,
Action<Diagnostic> addDiagnostic, CancellationToken cancellationToken)
{
    var localDeclaration = (LocalDeclarationStatementSyntax)node;

    // Only consider local variable declarations that aren't already const.
    if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
    {
        return;
    }

    // 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;
        }

        var constantValue = semanticModel.GetConstantValue(initializer.Value);
        if (!constantValue.HasValue)
        {
            return;
        }
    }

    // Perform data flow analysis on the local declaration.
    var 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.
    foreach (var variable in localDeclaration.Declaration.Variables)
    {
        var variableSymbol = semanticModel.GetDeclaredSymbol(variable);
        if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
        {
            return;
        }
    }

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

```

8. 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 C# Console Application project and add a few local variable declarations initialized with constant values to the Main method.

```
static void Main(string[] args)
{
    int i = 1;
    int j = 2;
    int k = i + j;
}
```

- You'll see that they are reported as warnings as below.

```
class Program
{
    static void Main(string[] args)
    {
        int i = 1;
        int j = 2;
    }
}
```

Can be made constant

- 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.

```
class Program
{
    static void Main(string[] args)
    {
        const int i = 1;
        const int j = 2;
        int k = i + j;
    }
}
```

Can be made constant

9. 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 inserts 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.cs 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 local declaration identified by the diagnostic.  
var declaration =  
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>().First();
```

4. 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 new[] { CodeAction.Create("Make constant", c => MakeConstAsync(document,  
declaration, c)) };
```

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

```

using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CodeFixes;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Rename;
using Microsoft.CodeAnalysis.Text;

namespace FirstDiagnostic
{
    [ExportCodeFixProvider(DiagnosticAnalyzer.DiagnosticId, LanguageNames.CSharp)]
    internal class CodeFixProvider : ICodeFixProvider
    {
        public IEnumerable<string> GetFixableDiagnosticIds()
        {
            return new[] { DiagnosticAnalyzer.DiagnosticId };
        }

        public async Task<IEnumerable<CodeAction>> GetFixesAsync(Document document,
            TextSpan span, IEnumerable<Diagnostic> diagnostics, CancellationToken
            cancellationToken)
        {
            var root = await
            document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);

            var diagnosticSpan = diagnostics.First().Location.SourceSpan;

            // Find the local declaration identified by the diagnostic.
            var declaration =
            root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>().First();

            // Return a code action that will invoke the fix.
            return new[] { CodeAction.Create("Make constant", c =>
            MakeConstAsync(document, declaration, c)) };
        }
    }
}

```

6. 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 Task<Document> MakeConstAsync(Document document,
LocalDeclarationStatementSyntax localDeclaration, CancellationToken
cancellationToken)
```

- Then, 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.LeadingsTrivia;
var trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
var constToken = SyntaxFactory.Token(leadingTrivia, SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

- 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 Formatter syntax annotation to the new declaration statement, which is an indicator to the Code Fix engine to format any whitespace using the C# 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.
var formattedLocal = newLocal.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.
var oldRoot = await document.GetSyntaxRootAsync(cancellationToken);
var newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);
```

- 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 Task<Document> MakeConstAsync(Document document,  
LocalDeclarationStatementSyntax localDeclaration, CancellationToken  
cancellation_token)  
{  
    // Remove the leading trivia from the local declaration.  
    var firstToken = localDeclaration.GetFirstToken();  
    var leadingTrivia = firstToken.LeadingsTrivia;  
    var trimmedLocal = localDeclaration.ReplaceToken(  
        firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));  
  
    // Create a const token with the leading trivia.  
    var constToken = SyntaxFactory.Token(leadingTrivia, SyntaxKind.ConstKeyword,  
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));  
  
    // 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 = newLocal.WithAdditionalAnnotations(Formatter.Annotation);  
  
    // Replace the old local declaration with the new local declaration.  
    var oldRoot = await document.GetSyntaxRootAsync(cancellation_token);  
    var newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);  
  
    // Return document with transformed tree.  
    return document.WithSyntaxRoot(newRoot);  
}
```

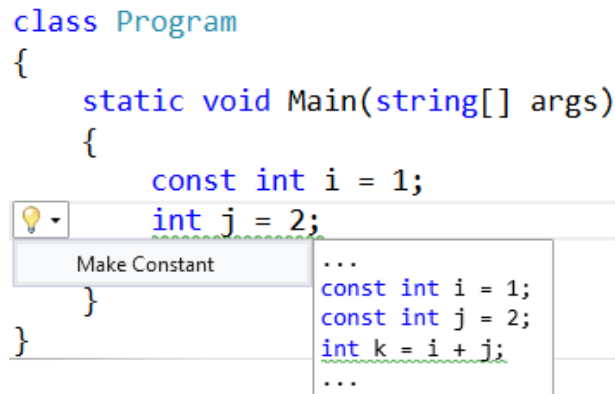
7. Press Ctrl+F5 to run the Diagnostic project in a second instance of Visual Studio with the Roslyn Preview extension loaded.

- In the second Visual Studio instance, create a new C# Console Application project and, like before, add a few local variable declarations initialized with to constant values in the Main method.

```
static void Main(string[] args)  
{  
    int i = 1;  
    int j = 2;  
    int k = i + j;  
}
```

- 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.



Fixing Bugs

Sadly, there are a few bugs in the implementation.

1. The Diagnostic Analyzer's `AnalyzeNode` 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 Fix does the wrong thing and generates a `"const var"` declaration, which is not supported by the C# language. To fix this bug, the code fix must replace the `"var"` keyword with the inferred type's name.

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

1. To fix the first bug, first open `DiagnosticAnalyzer.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 `SemanticModel.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 curly brace of the foreach loop to call `SemanticModel.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;
}
```

2. The next bug fix builds upon the last one.

- Before the closing curly brace of the same foreach loop, add the following code to check the type of the 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;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

- With this code in place, the AnalyzeNode method should look like so.

```

public void AnalyzeNode(SyntaxNode node, SemanticModel semanticModel,
Action<Diagnostic> addDiagnostic, CancellationToken cancellationToken)
{
    var localDeclaration = (LocalDeclarationStatementSyntax)node;

    // Only consider local variable declarations that aren't already const.
    if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
    {
        return;
    }

    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;
        }

        var constantValue = semanticModel.GetConstantValue(initializer.Value);
        if (!constantValue.HasValue)
        {
            return;
        }

        // 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;
        }

        // 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;
            }
        }
        else if (variableType.IsReferenceType && constantValue.Value != null)
        {
            return;
        }
    }
}

```

```

    }

    // Perform data flow analysis on the local declaration.
    var 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.
    foreach (var variable in localDeclaration.Declaration.Variables)
    {
        var variableSymbol = semanticModel.GetDeclaredSymbol(variable);
        if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
        {
            return;
        }
    }

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

```

3. Fixing the third issue requires a little more code to replace the 'var' keyword with the correct type name.

- Return to CodeFixProvider.cs and replace the code at the comment which reads "Produce the new local 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 = await document.GetSemanticModelAsync(cancellationToken);

// 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 = SyntaxFactory.ParseTypeName(type.ToString())
    .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
    .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());
```

- Add a Simplifier syntax annotation to the type name to ensure that the code fix engine reduces the type name to its minimally-qualified form. Use Ctrl+. on Simplifier to add the using statement for Microsoft.CodeAnalysis.Simplification.

```
// Add an annotation to simplify the type name.
var simplifiedTypeName = typeName.WithAdditionalAnnotations(Simplifier.Annotation);
```

- 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 MakeConstAsync method should now look like the following:

```

private async Task<Document> MakeConstAsync(Document document,
LocalDeclarationStatementSyntax localDeclaration, CancellationToken
cancellationToken)
{
    // Remove the leading trivia from the local declaration.
    var firstToken = localDeclaration.GetFirstToken();
    var leadingTrivia = firstToken.LeadingsTrivia;
    var trimmedLocal = localDeclaration.ReplaceToken(
        firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

    // Create a const token with the leading trivia.
    var constToken = SyntaxFactory.Token(leadingTrivia, SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));

    // Insert the const token into the modifiers list, creating a new modifiers list.
    var newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);

    // 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)
    {
        var semanticModel = await document.GetSemanticModelAsync(cancellationToken);

        // 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 = SyntaxFactory.ParseTypeName(type.ToDisplayString())
                    .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                    .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

                // Add an annotation to simplify the type name.
                var simplifiedTypeName =
typeName.WithAdditionalAnnotations(Simplifier.Annotation);

                // 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 = newLocal.WithAdditionalAnnotations(Formatter.Annotation);

// Replace the old local declaration with the new local declaration.
var root = await document.GetSyntaxRootAsync(cancellationToken);
var newRoot = root.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
}

```

4. Once again, 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 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 (though there's a compiler error as expected).
 - 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 suggestion appears beneath to the left.
 - Move the editor caret over the squiggly underline and press Ctrl+. to display the suggested code fix. Upon selecting your code fix, note that the 'var' keyword is now handled correctly.

```

class Program
{
    static void Main(string[] args)
    {
        object s = "abc";
        var i = 1.0 + 2;
    }
}

```

Make Constant

```

...
object s = "abc";
const double i = 1.0 + 2;
...

```

5. 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.