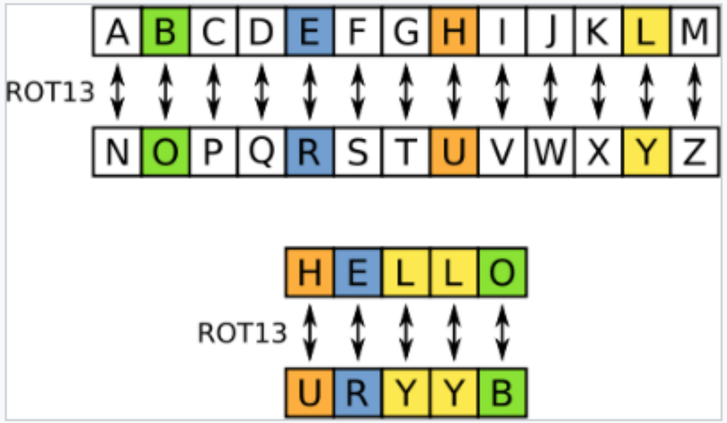
CONSOLE APP

A walkthrough

Rot13 Example

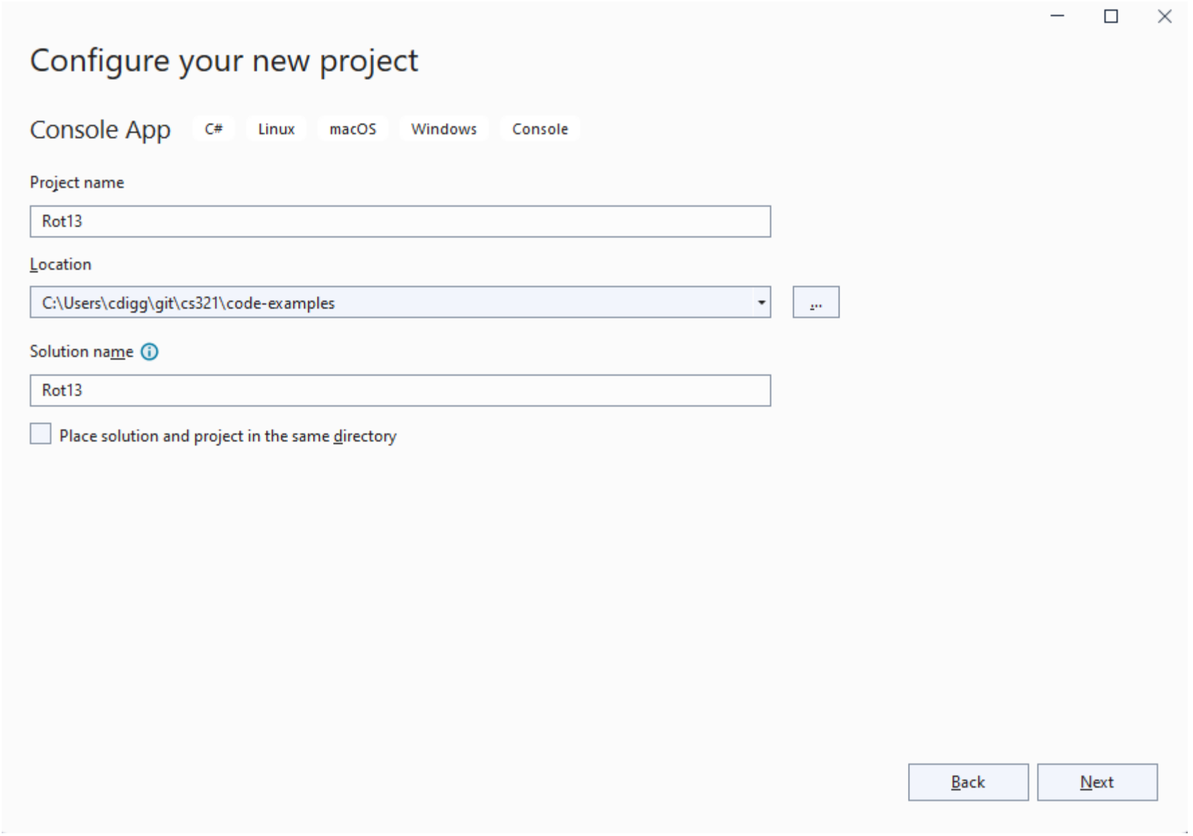


ROT13 replaces each letter by its partner 13 characters further along the alphabet. For example, HELLO becomes URYYB (or, conversely, URYYB becomes HELLO again).

Program Requirements

- Convert letters in every line of text using the Rot13 Cipher
- Support different offsets that are set on the command-line
- Check an environment variable for the offset

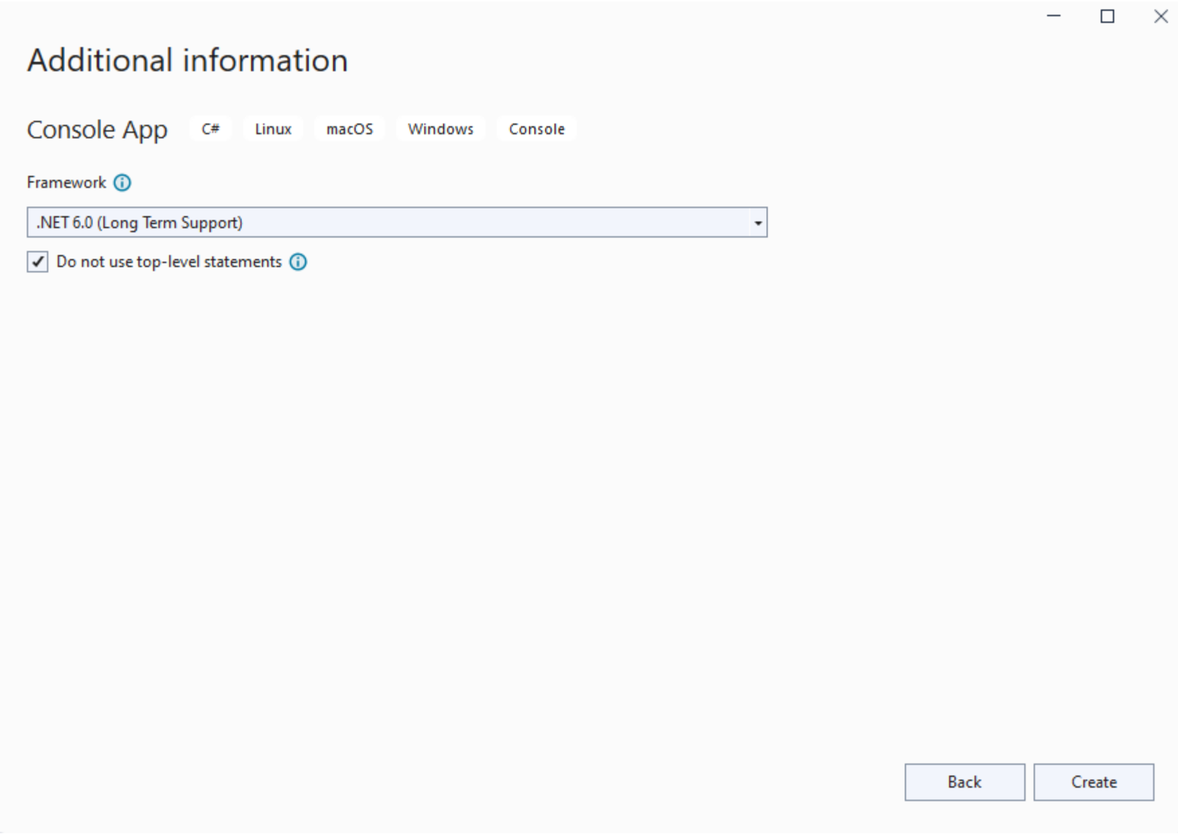
Create a Console Project



One Folder per Project

- Solution in the upper level
- Starting with the console project
- We will add unit tests and shared code after

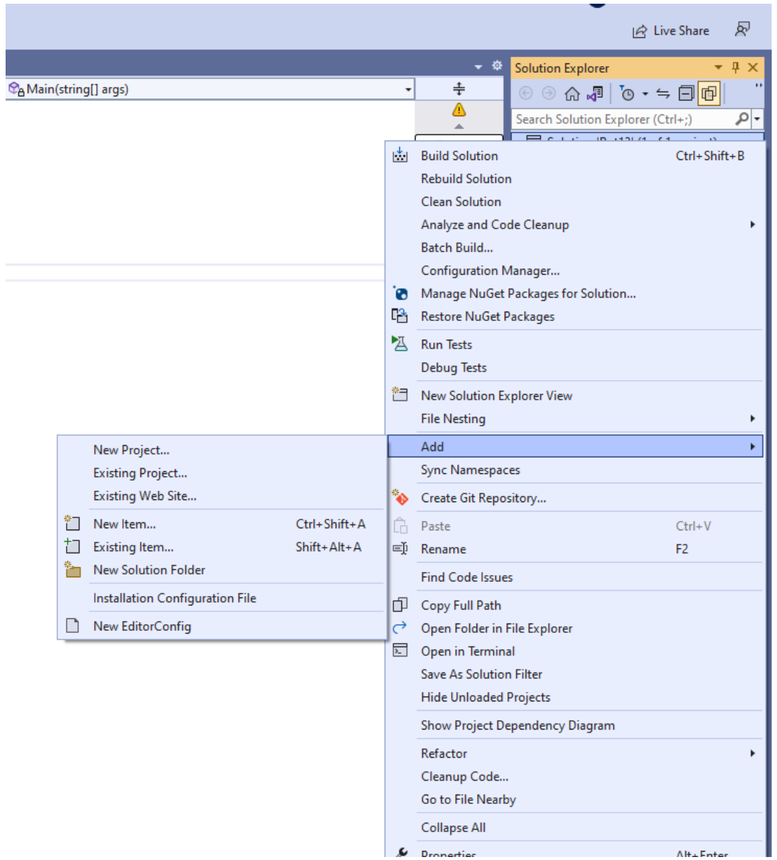
Choose the .NET Framework



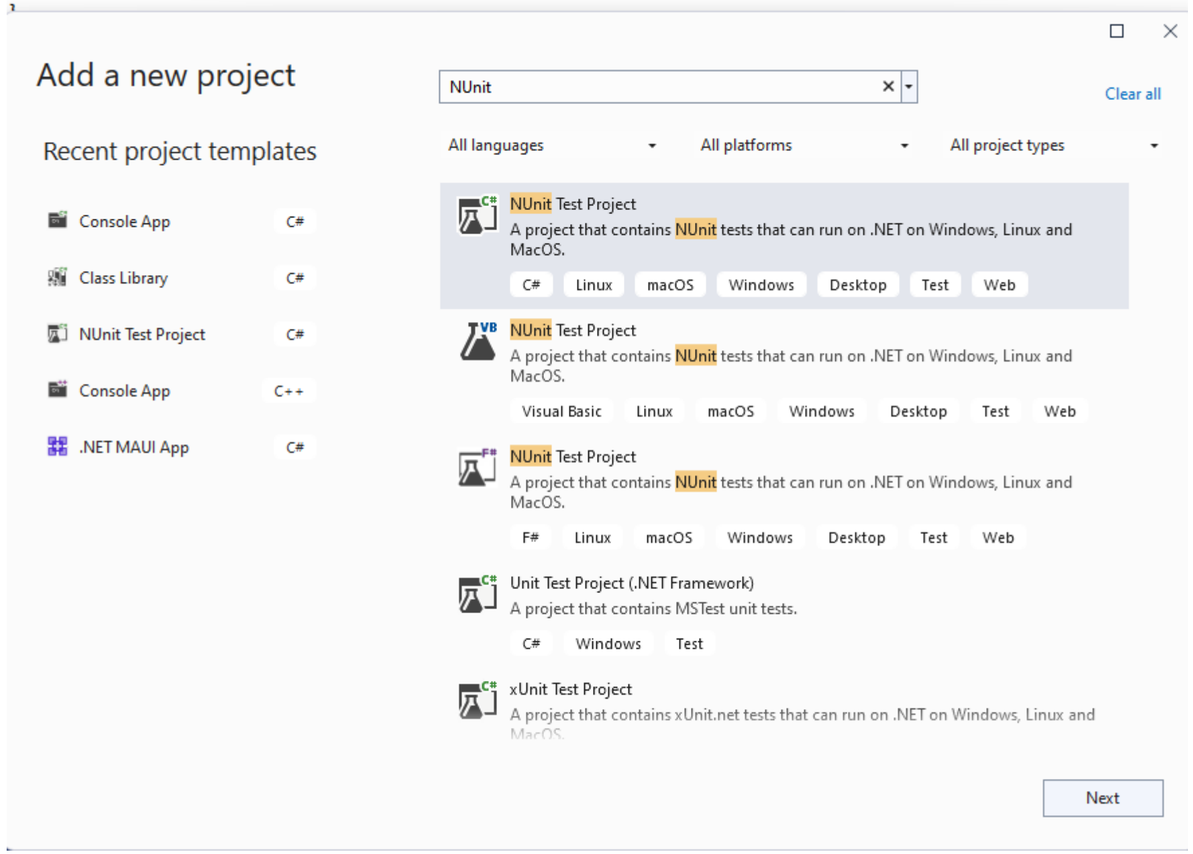
```
namespace Rot13
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

GENERATED CODE

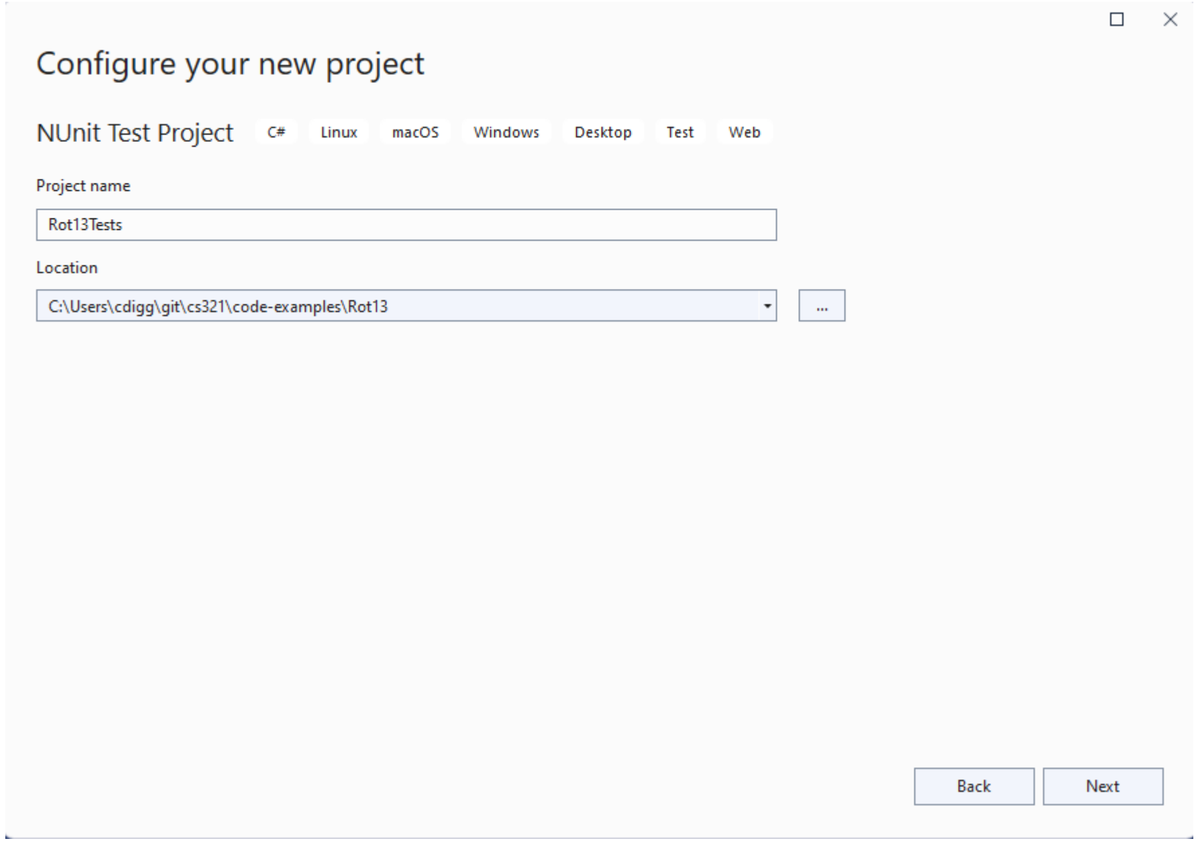
Right Click on Solution – Add New Project



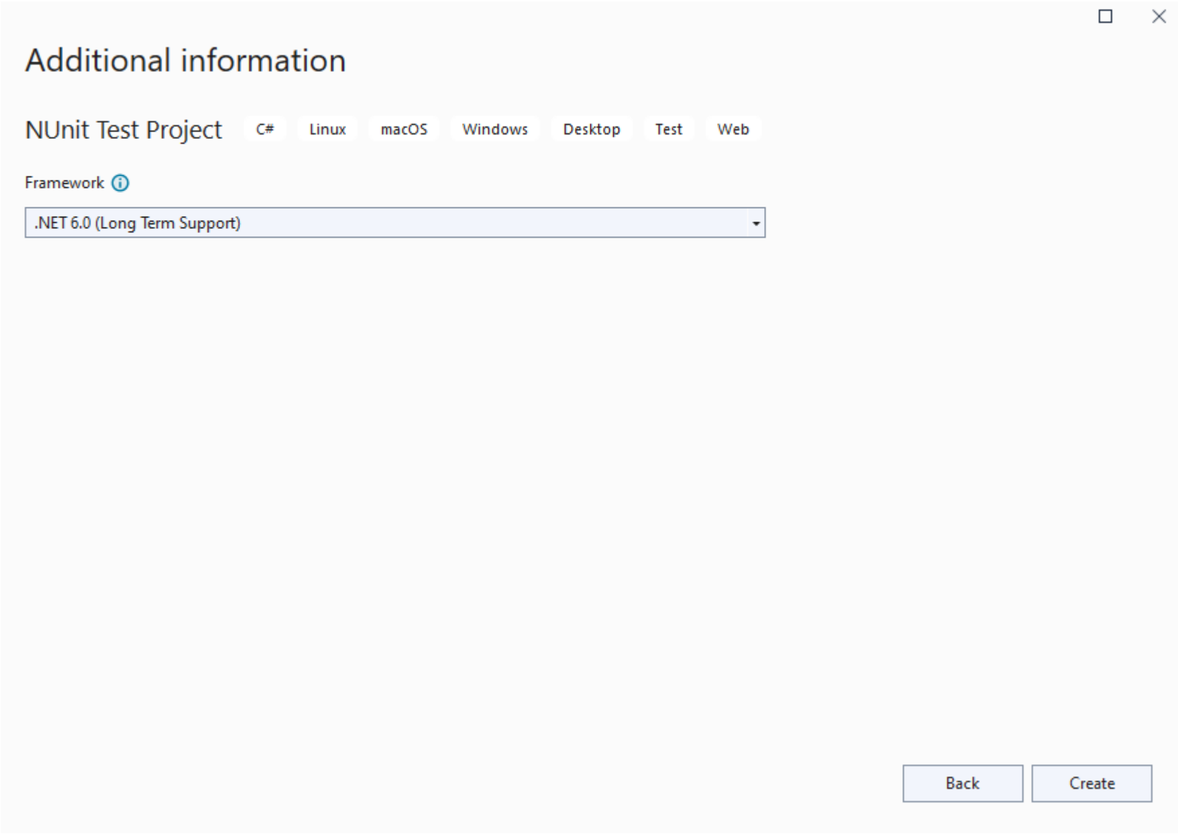
Create a new NUnit Test Project



Name the Project



Set the .NET Framework



Generated Code

```
1  namespace Rot13Tests
2  {
3      public class Tests
4      {
5          [SetUp]
6          public void Setup()
7          {
8          }
9
10         [Test]
11         public void Test1()
12         {
13             Assert.Pass();
14         }
15     }
16 }
```

We want to write a smoke test

- Just make sure we can start the program from the entry point

Make unit test class static

```
1   namespace Rot13Tests
2   {
3       public static class Tests
4       {
5           [Test]
6           public static void Test1()
7           {
8               Rot13.Program.Main();
9           }
10      }
11  }
```

What does the error say?

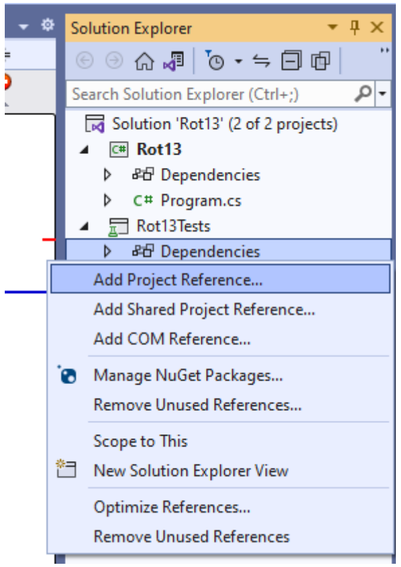
```
namespace Rot13Tests
{
    public static class Tests
    {
        [Test]
        public static void Test1()
        {
            Rot13.Program.Main();
        }
    }
}
```

CS0103: The name 'Rot13' does not exist in the current context

Cannot resolve symbol 'Rot13'

Show potential fixes (Alt+Enter or Ctrl+.)

Add Missing Project Reference



New Problem

```
namespace Rot13Tests
{
    public static class Tests
    {
        [Test]
        public static void Test1()
        {
            Rot13.Program.Main();
        }
    }
}
```

CS0122: 'Program' is inaccessible due to its protection level

Cannot access internal class 'Program' here

What is Internal

- Called an access modifier
- Prevents access to functions and types from other libraries and executables

Summary table

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

Make the Program class public

```
1  namespace Rot13
2  {
3      public class Program
4      {
5          static void Main(string[] args)
6          {
7              Console.WriteLine("Hello, World!");
8          }
9      }
10 }
```

Now Visibility Problem on Main

```
namespace Rot13Tests
{
    public static class Tests
    {
        [Test]
        public static void Test1()
        {
            Rot13.Program.Main();
        }
    }
}
```

void Rot13.Program.Main(string[] args)

CS0122: 'Program.Main(string[])' is inaccessible due to its protection level

Cannot access private method 'Main(string[])' here

Show potential fixes (Alt+Enter or Ctrl+.)

Make class static and add public to main

```
1  namespace Rot13
2  {
3      public static class Program
4      {
5          public static void Main(string[] args)
6          {
7              Console.WriteLine("Hello, World!");
8          }
9      }
10 }
```

Why a static class?

- We will never create a “Rot13.Program” object
- We only use the “Rot13.Program” class
- It will only have static methods
- If you don’t new to “new” it, make it static

One last problem

```
1  namespace Rot13Tests
2  {
3      public static class Tests
4      {
5          [Test]
6          public static void Test1()
7          {
8              Rot13.Program.Main();
9          }
10     }
11 }
```

Method 'Main' has 1 parameter(s) but is invoked with 0 argument(s)

Parameters and Arguments

- Parameters are variables that are bound to a function's inputs
- Arguments are the values that are passed to a function as inputs

Create an empty array and pass it

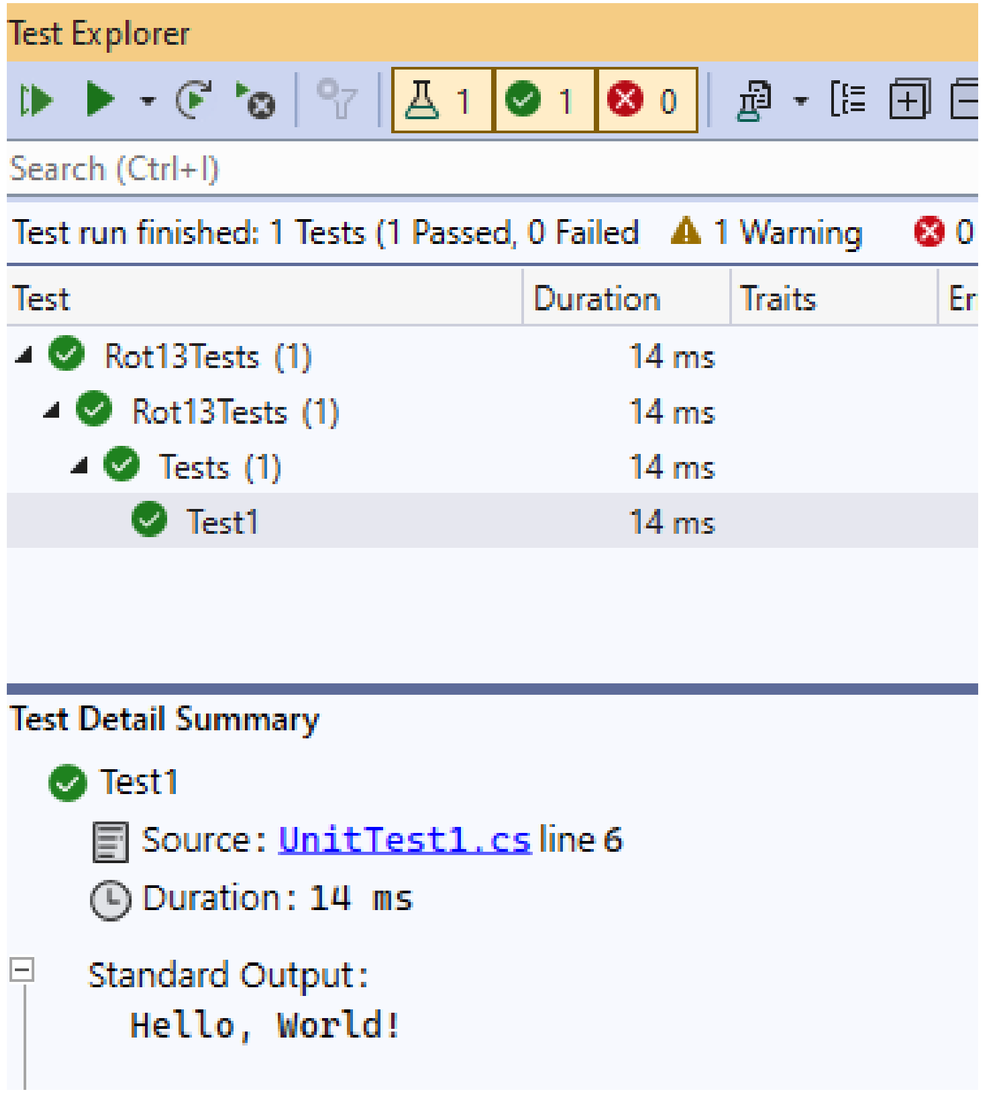
```
1  namespace Rot13Tests
2  {
3      public static class Tests
4      {
5          [Test]
6          public static void Test1()
7          {
8              Rot13.Program.Main(Array.Empty<string>());
9          }
10     }
11 }
```

What is Array.Empty<T>()?

- What can you say about this?

Array.Empty<T>()

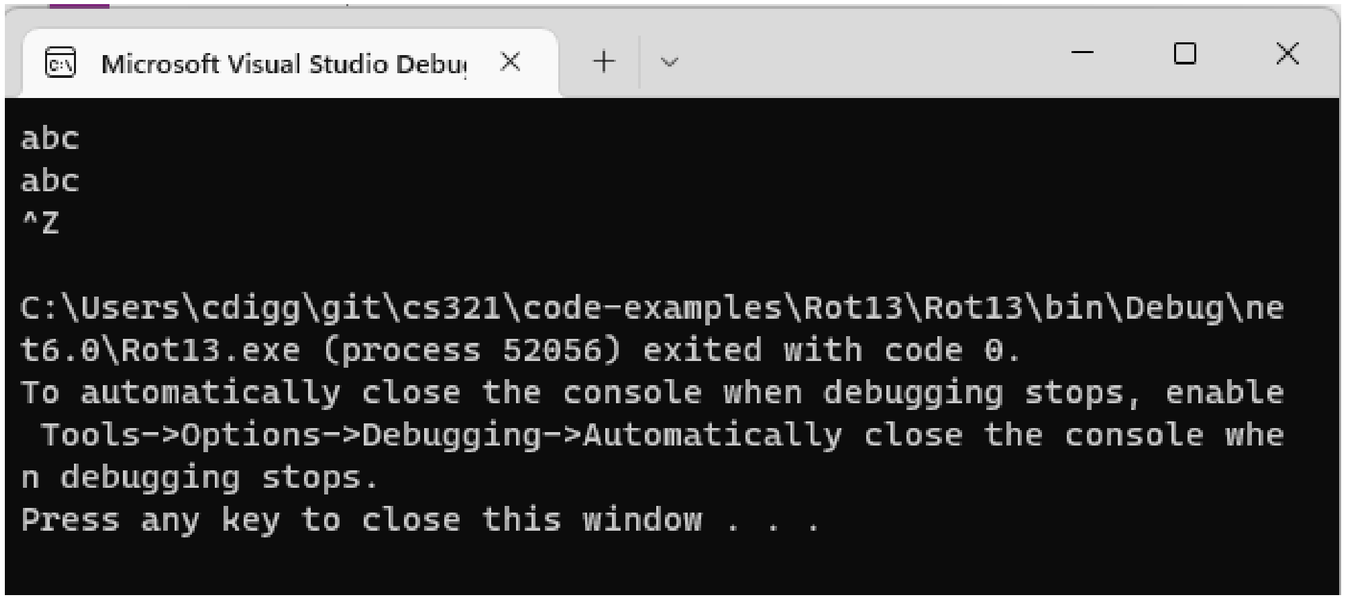
- It is a function invocation
- It is an expression
- It is a static method - because Array is a type
- It is public - because we can call it from a non-derived type other than array
- It is not internal – because we can call it from another assembly
- It is generic – it takes a type parameter
- The expression has a type of “T[]”
- It returns a non-null value of run-time type of “T[]”



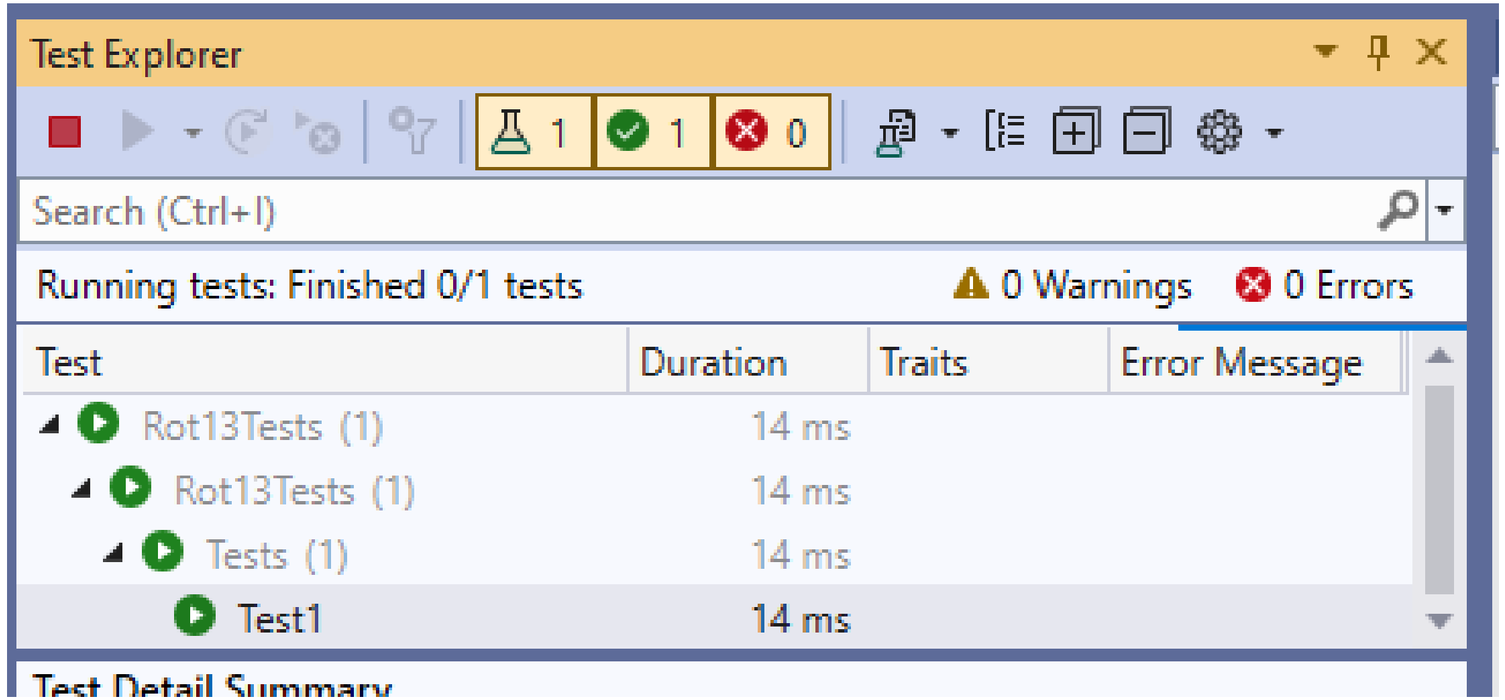
TEST RESULTS

Create a Stub Implementation

```
1  namespace Rot13
2  {
3      public static class Program
4      {
5          public static void Main(string[] args)
6          {
7              var line = Console.ReadLine();
8              while (line != null)
9              {
10                 Console.WriteLine(Rot13(line));
11                 line = Console.ReadLine();
12             }
13         }
14
15         public static string Rot13(string input)
16         {
17             // TODO: implement this method
18             return input;
19         }
```



RUN THE PROGRAM FROM IDE



PROBLEM: TEST NOW HANGS!

It is waiting for standard input

- Tests provide no console

How to test standard input?

- https://stackoverflow.com/questions/13601175/imitate-the-standard-input-in-tests

1 Answer

Sorted by: Highest score (default)

4

You can replace the console input with your own object, say `StringReader`, and provide any input you want to it:

```
var oldIn = Console.In;
try
{
    Console.SetIn(new StringReader("some input"));

    using (App app = new App())
    {
        // input = Console.In.ReadToEnd(); happens here
        result = app.Run(args);
    }

    if (result != 0)
    {
        Assert.Fail("Failed");
    }
}
finally
{
    Console.SetIn(oldIn);
}
```

Share Edit Follow Flag

edited Nov 28, 2012 at 9:55

answered Nov 28, 2012 at 8:58

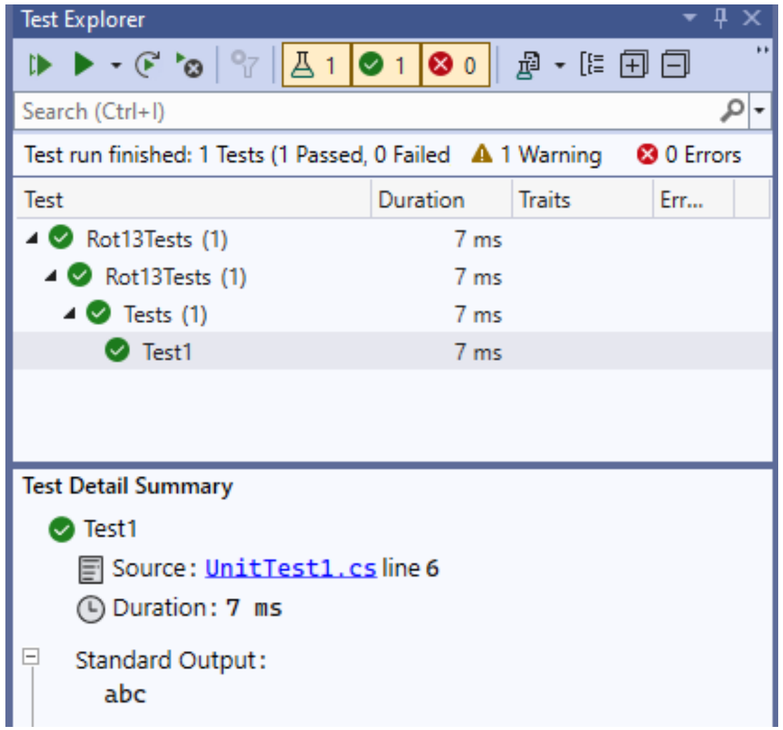
Andrei

55.4k • 9 • 85 • 107

Use Console.SetIn()

```
1  namespace Rot13Tests
2  {
3      public static class Tests
4      {
5          [Test]
6          public static void Test1()
7          {
8              var stringReader = new StringReader("abc");
9              Console.SetIn(stringReader);
10             Rot13.Program.Main(Array.Empty<string>());
11         }
12     }
13 }
```

Test now Passes



A very subtle problem

- Why did the answer have a “try” and “finally”?

1 Answer

Sorted by: Highest score (default)

4

You can replace the console input with your own object, say `StringReader`, and provide any input you want to it:

```
var oldIn = Console.In;
try
{
    Console.SetIn(new StringReader("some input"));

    using (App app = new App())
    {
        // input = Console.In.ReadToEnd(); happens here
        result = app.Run(args);
    }

    if (result != 0)
    {
        Assert.Fail("Failed");
    }
}
finally
{
    Console.SetIn(oldIn);
}
```

Share Edit Follow Flag

edited Nov 28, 2012 at 9:55

answered Nov 28, 2012 at 8:58

Andrei

55.4k • 9 • 85 • 107

Try / Finally Statement

- After we leave the try block always call the finally block
- No matter how we leave that block of code
- For example could be a return statement or an exception
- Useful to make sure a clean-up always happens
- We will come back to this later

If not …

- Imagine running multiple tests
- If you leave a failing test the standard input might not be restored
- For our immediate purposes: not important

Now let's implement Rot13

```
public static string Rot13(string input)
{
    // TODO: implement this method
    return input;
}
```

What if we had a Rot13 function for char?

```
/// <summary>
/// Given a character in the range A-Z or a-z, returns the character offset by 13 places,
/// otherwise returns the character unchanged.
/// </summary>
public static char Rot13(char c)
{
```

Long Form

```
public static string Rot13_LongWay(string input)
{
    var list = new List<char>();
    foreach (var c in input)
        list.Add(Rot13(c));
    var chars = list.ToArray();
    return new string(chars);
}
```

Automated Refactoring Suggestion

```
public static string Rot13_LongWay(string input)
{
    var list = new List<char>();
    foreach (var c in input)

    return new string(chars);
}
```

Loop can be converted into LINQ-expression but another 'GetEnumerator' method will be used

First Refactoring Result

```
public static string Rot13_LongWay(string input)
{
    var chars = input.Select(c => Rot13(c)).ToArray();
    return new string(chars);
}
```

Next Refactoring Suggestion

```
t(c => Rot13(c)).ToArray();
);

string inp
```

char Program.Rot13(char c) (+ 1 overload)

Given a character in the range A-Z or a-z, returns the cha

Convert into method group

Final Automated Refactoring

```
public static string Rot13_LongWay(string input)
{
    var chars = input.Select(Rot13).ToArray();
    return new string(chars);
}
```

Even Shorter Form

```
public static string Rot13(string input)
{
    return new string(input.ToCharArray().Select(Rot13).ToArray());
}
```

Advantages of Short Form (LINQ)

- Less change of error
- More explicit about desired result
- Less explicit about technique

Could go another step

```
22
23  public static string Rot13(string input)
24  {
25      return new string(input.ToCharArray().Select(Rot13).ToArray());
    }
```

To expression body

Expression body form: implicit return

```
public static string Rot13(string input)
    => new string(input.ToCharArray().Select(Rot13).ToArray());
```

Now a new suggestion appears

```
public static string Rot13(string input)
    => new string(input.ToCharArray().Select(Rot13).ToArray());

/// <summar
/// Given a
/// otherwi
/// </summa
public stat
{
    // If n
    if (!ch
```

string.String(char[]? value) (+ 8 overloads)
Initializes a new instance of the string class to the Unicode characters indicated in the specified character array.

IDE0090: 'new' expression can be simplified

IDE0090: 'new' expression can be simplified

Redundant type specification

Show potential fixes (Alt+Enter or Ctrl+.)

Target-Typed New

- Types of the new expression can be inferred in some cases
- Called a target-typed new
- Available in C# 9 and beyond
- I find it very convenient

Side by Side: You Decide

```
public static string Rot13_LongWay(string input)
{
    var list = new List<char>();
    foreach (var c in input)
        list.Add(Rot13(c));
    var chars = list.ToArray();
    return new string(chars);
}

public static string Rot13_ShortWay(string input)
    => new(input.ToCharArray().Select(Rot13).ToArray());
```

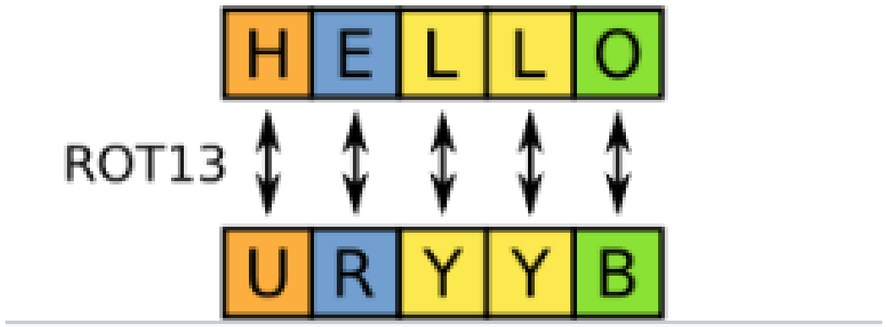
Possible implementation of Rot13

```
/// <summary>
/// Given a character in the range A-Z or a-z, returns the character offset by 13 places,
/// otherwise returns the character unchanged.
/// </summary>
public static char Rot13(char c)
{
    // If not a letter, return the character
    if (!char.IsLetter(c))
    {
        return c;
    }

    // Convert the character into a code
    var code = (int)c;

    // Compute the new character code
    var newCode = code + 13;

    // If it was an upper-case letter, check if the new code is beyond 'Z'
    if (char.IsUpper(c) && newCode > 'Z')
    {
        // Loop back to the beginning of the alphabet
        newCode -= 26;
    }

    // If it was a lower-case letter, check if the new code is beyond lower-case 'z'
    if (char.IsLower(c) && newCode > 'z')
    {
        // Loop back to the beginning of the alphabet
        newCode -= 26;
    }

    return (char)newCode;
}
```

Let’s Test it

```
[Test]
public static void Rot13CharTest()
{
    Assert.AreEqual('1', Rot13.Program.Rot13('1'));
    Assert.AreEqual('a', Rot13.Program.Rot13('n'));
    Assert.AreEqual('A', Rot13.Program.Rot13('N'));
    Assert.AreEqual('m', Rot13.Program.Rot13('z'));
    Assert.AreEqual('M', Rot13.Program.Rot13('Z'));
}
```

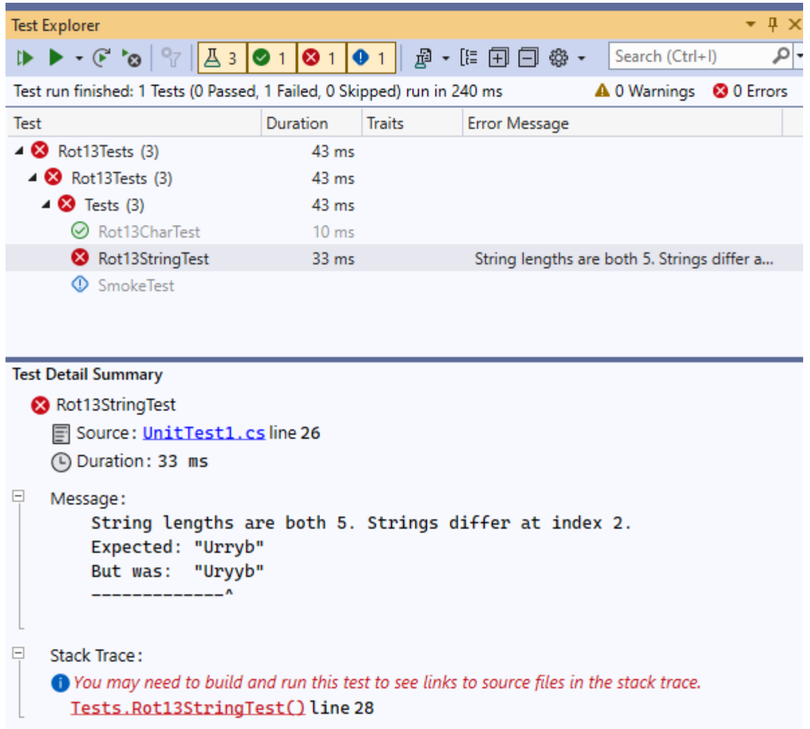
Test Example from Wikipedia



Try in Code

```
[Test]
public static void Rot13StringTest()
{
    Assert.AreEqual("Urryb", Rot13.Program.Rot13("Hello"));
}
```

Test Failed


Test Explorer
3
1
1
1
Search (Ctrl+I)
Test run finished: 1 Tests (0 Passed, 1 Failed, 0 Skipped) run in 240 ms
0 Warnings
0 Errors
Test
Duration
Traits
Error Message
Rot13Tests (3)
43 ms
Rot13Tests (3)
43 ms
Tests (3)
43 ms
Rot13CharTest
10 ms
Rot13StringTest
33 ms
String lengths are both 5. Strings differ a...
SmokeTest
Test Detail Summary
Rot13StringTest
Source: UnitTest1.cs line 26
Duration: 33 ms
Message:
String lengths are both 5. Strings differ at index 2.
Expected: "Urryb"
But was: "Uryyb"
-------------^
Stack Trace:
You may need to build and run this test to see links to source files in the stack trace.
Tests.Rot13StringTest() line 28

Can try debugging it

```
public static void SmokeTest()
{
```

Exception User-Unhandled

NUnit.Framework.AssertionException: ' String lengths are both 5. Strings differ at index 2.
Expected: "Urryb"
But was: "Uryyb"
----------^
'

Show Call Stack | View Details | Copy Details | Start Live Share session

Exception Settings
- [ ] Break when this exception type is thrown
- [x] Break when this exception type is user-unhandled

Except when thrown from:
- [ ] Rot13Tests.dll

Open Exception Settings | Edit Conditions

```
}
[Test]
public static void Rot13StringTest()
{
    Assert.AreEqual("Urryb", Rot13.Program.Rot13("Hello"));
}
```

Problem can be

- Either my test
- Or my implementation

In this case … was my test

```
[Test]
public static void Rot13StringTest()
{
    Assert.AreEqual("Uryyb", Rot13.Program.Rot13("Hello"));
}
```

Why all the squiggles?

```
[Test]
public static void Rot13CharTest()
{
    Assert.AreEqual('1', Rot13.Program.Rot13('1'));
}
```

class NUnit.Framework.Assert

The Assert class contains a collection of static methods that implement the most common assertions used in NUnit.

NUnit2005: Consider using the constraint model, Assert.That(actual, Is.EqualTo(expected)), instead of the classic model, Assert.AreEqual (expected, actual)

Suggested Refactoring Constraint Model

```
Assert.AreEqual('m', Rot13.Program.Rot13('z'));
Assert.AreEqual('M', Rot13.Program.Rot13('Z'));
```

Transform to constraint model

Suppress or Configure issues

NUnit2005 Consider using the constraint model, Assert.That(actual, Is.EqualTo(expected)), instead of the classic model, Assert.AreEqual(expected, actual)

Lines 21 to 23

```
    Assert.AreEqual('m', Rot13.Program.Rot13('z'));
    Assert.AreEqual('M', Rot13.Program.Rot13('Z'));
    Assert.That(Rot13.Program.Rot13('Z'), Is.EqualTo('M'));
}
```

Preview changes

Fix all occurrences in: Document | Project | Solution | Containing Member | Containing Type

Apply Refactoring to Document

```
public static class Tests
{
    [Test]
    public static void SmokeTest()
    {
        var stringReader = new StringReader("abc");
        Console.SetIn(stringReader);
        Rot13.Program.Main(Array.Empty<string>());
    }

    [Test]
    public static void Rot13CharTest()
    {
        Assert.That(Rot13.Program.Rot13('1'), Is.EqualTo('1'));
        Assert.That(Rot13.Program.Rot13('n'), Is.EqualTo('a'));
        Assert.That(Rot13.Program.Rot13('N'), Is.EqualTo('A'));
        Assert.That(Rot13.Program.Rot13('z'), Is.EqualTo('m'));
        Assert.That(Rot13.Program.Rot13('Z'), Is.EqualTo('M'));
    }

    [Test]
    public static void Rot13StringTest()
    {
        Assert.That(Rot13.Program.Rot13("Hello"), Is.EqualTo("Uryyb"));
    }
}
```

Your choose your preference

- Currently I find the “classic model” easier to read
- But squiggles are distracting
- And the constraint model is now recommended
- Lesson: things seem strange until you become accustomed to them

Using Directive

- We don't need to have "Rot13.Program." everywhere
- It is because we haven't included the namespace
- Try adding a using directive (read-up on this)
- We can now use "Program."

Isn’t this better?

```
public static class Tests
{
    [Test]
    public static void SmokeTest()
    {
        var stringReader = new StringReader("abc");
        Console.SetIn(stringReader);
        Rot13.Program.Main(Array.Empty<string>());
    }

    [Test]
    public static void Rot13CharTest()
    {
        Assert.That(Rot13.Program.Rot13('1'), Is.EqualTo('1'));
        Assert.That(Rot13.Program.Rot13('n'), Is.EqualTo('a'));
        Assert.That(Rot13.Program.Rot13('N'), Is.EqualTo('A'));
        Assert.That(Rot13.Program.Rot13('z'), Is.EqualTo('m'));
        Assert.That(Rot13.Program.Rot13('Z'), Is.EqualTo('M'));
    }

    [Test]
    public static void Rot13StringTest()
    {
        Assert.That(Rot13.Program.Rot13("Hello"), Is.EqualTo("Uryyb"));
    }
}
```

```
using Rot13;

namespace Rot13Tests
{
    public static class Tests
    {
        [Test]
        public static void SmokeTest()
        {
            var stringReader = new StringReader("abc");
            Console.SetIn(stringReader);
            Program.Main(Array.Empty<string>());
        }

        [Test]
        public static void Rot13CharTest()
        {
            Assert.That(Program.Rot13('1'), Is.EqualTo('1'));
            Assert.That(Program.Rot13('n'), Is.EqualTo('a'));
            Assert.That(Program.Rot13('N'), Is.EqualTo('A'));
            Assert.That(Program.Rot13('z'), Is.EqualTo('m'));
            Assert.That(Program.Rot13('Z'), Is.EqualTo('M'));
        }

        [Test]
        public static void Rot13StringTest()
        {
            Assert.That(Program.Rot13("Hello"), Is.EqualTo("Uryyb"));
        }
    }
```

Using static

- We can simplify further with the "using static" directive

```
1  using Rot13;
2  using static Rot13.Program;
3  using static NUnit.Framework.Assert;
4
```

After Automated Refactoring

```
1   using Rot13;
2   using static Rot13.Program;
3   using static NUnit.Framework.Assert;
4
5   namespace Rot13Tests
6   {
7       public static class Tests
8       {
9           [Test]
10          public static void SmokeTest()
11          {
12              var stringReader = new StringReader("abc");
13              Console.SetIn(stringReader);
14              Main(Array.Empty<string>());
15          }
16
17          [Test]
18          public static void Rot13CharTest()
19          {
20              That(Program.Rot13('1'), Is.EqualTo('1'));
21              That(Program.Rot13('n'), Is.EqualTo('a'));
22              That(Program.Rot13('N'), Is.EqualTo('A'));
23              That(Program.Rot13('z'), Is.EqualTo('m'));
24              That(Program.Rot13('Z'), Is.EqualTo('M'));
25          }
26
27          [Test]
28          public static void Rot13StringTest()
29          {
30              That(Program.Rot13("Hello"), Is.EqualTo("Uryyb"));
31          }
32      }
33  }
```

Wait … why still Program.Rot13

- Because “Rot13” alone is ambiguous (namespace or function?)

```
[Test]
public static void Rot13CharTest()
{
    That(Rot13('1'), Is.EqualTo('1'));
    Th
    That(Pr
    That(Pr
    That(Pr
}

[Test]
public static void Rot13StringTest()
```

{ } namespace Rot13

CS0118: 'Rot13' is a namespace but is used like a variable

Namespace name is not valid at this point

Show potential fixes (Alt+Enter or Ctrl+.)

Moral

- Simplify as much as you can but there are limits
- Compilers cannot guess what you meant if there is ambiguity

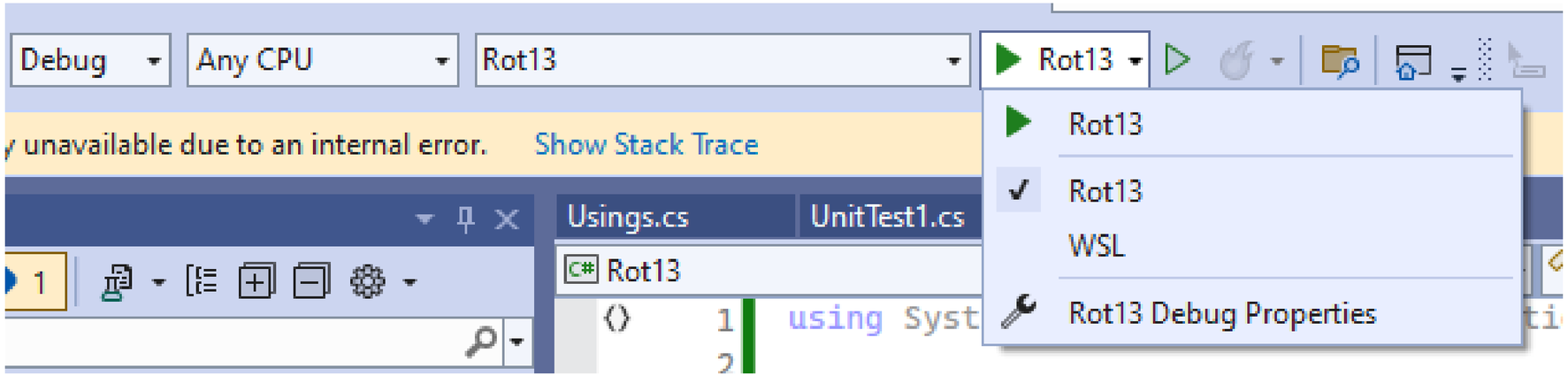
What if we want to read from a file?

```
public static void SetStandardInputFromFile(string fileName)
{
    // Redirect the input from a stream reader
    var fileReader = new StreamReader(fileName);
    Console.SetIn(fileReader);
}
```

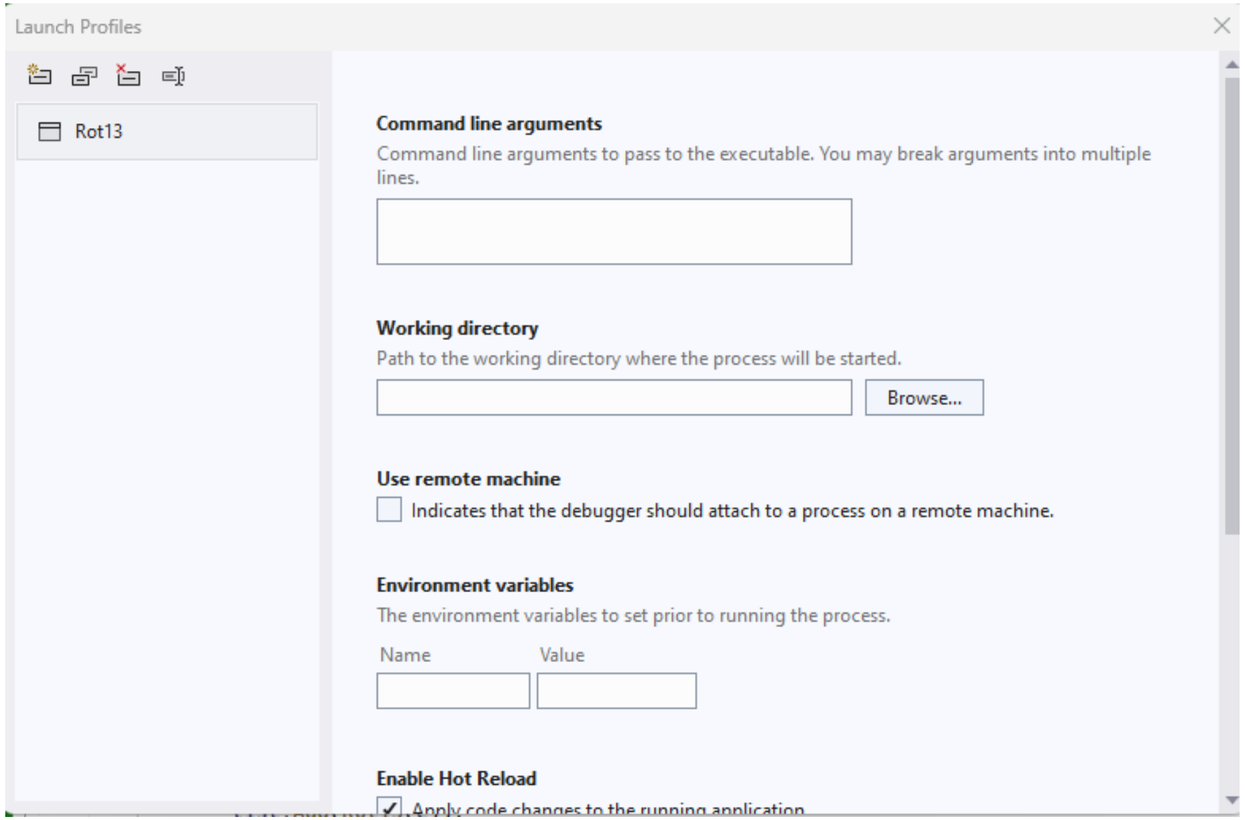
Parsing Command Line Arguments

```
public static void Main(string[] args)
{
    if (args.Length > 0)
    {
        var fileName = args[0];
        if (!File.Exists(fileName))
        {
            Console.WriteLine($"Expected first command-line argument {fileName} to be a valid filename");
            return;
        }

        SetStandardInputFromFile(fileName);
    }
```

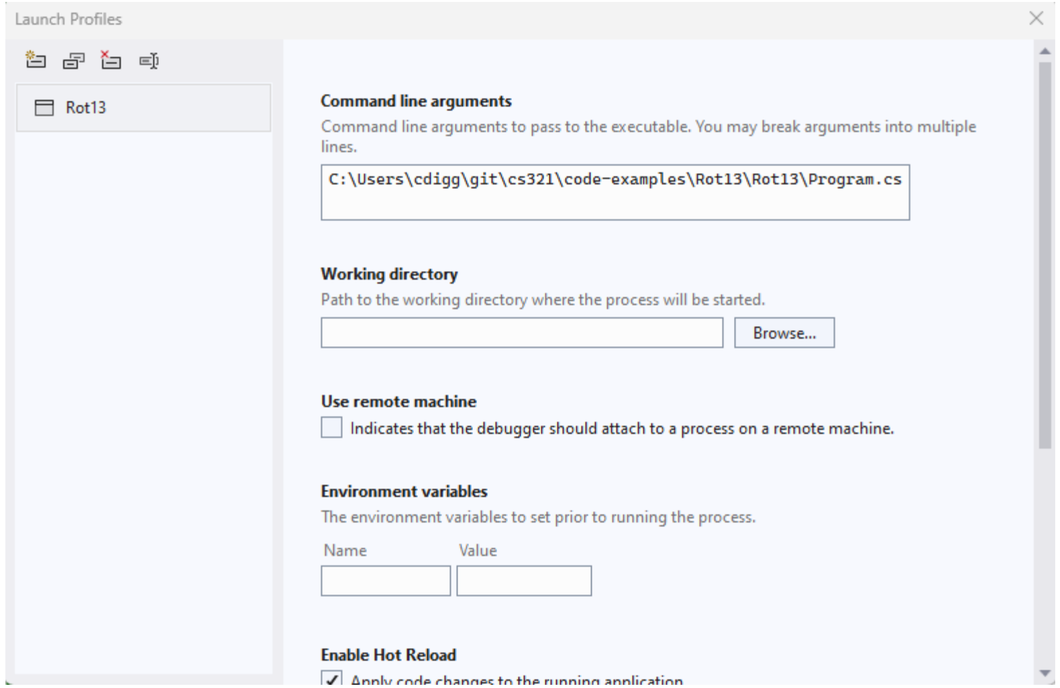
Manual Testing of Command Line



Launches the Debug Profiles Menu



Set the Command Line Argument

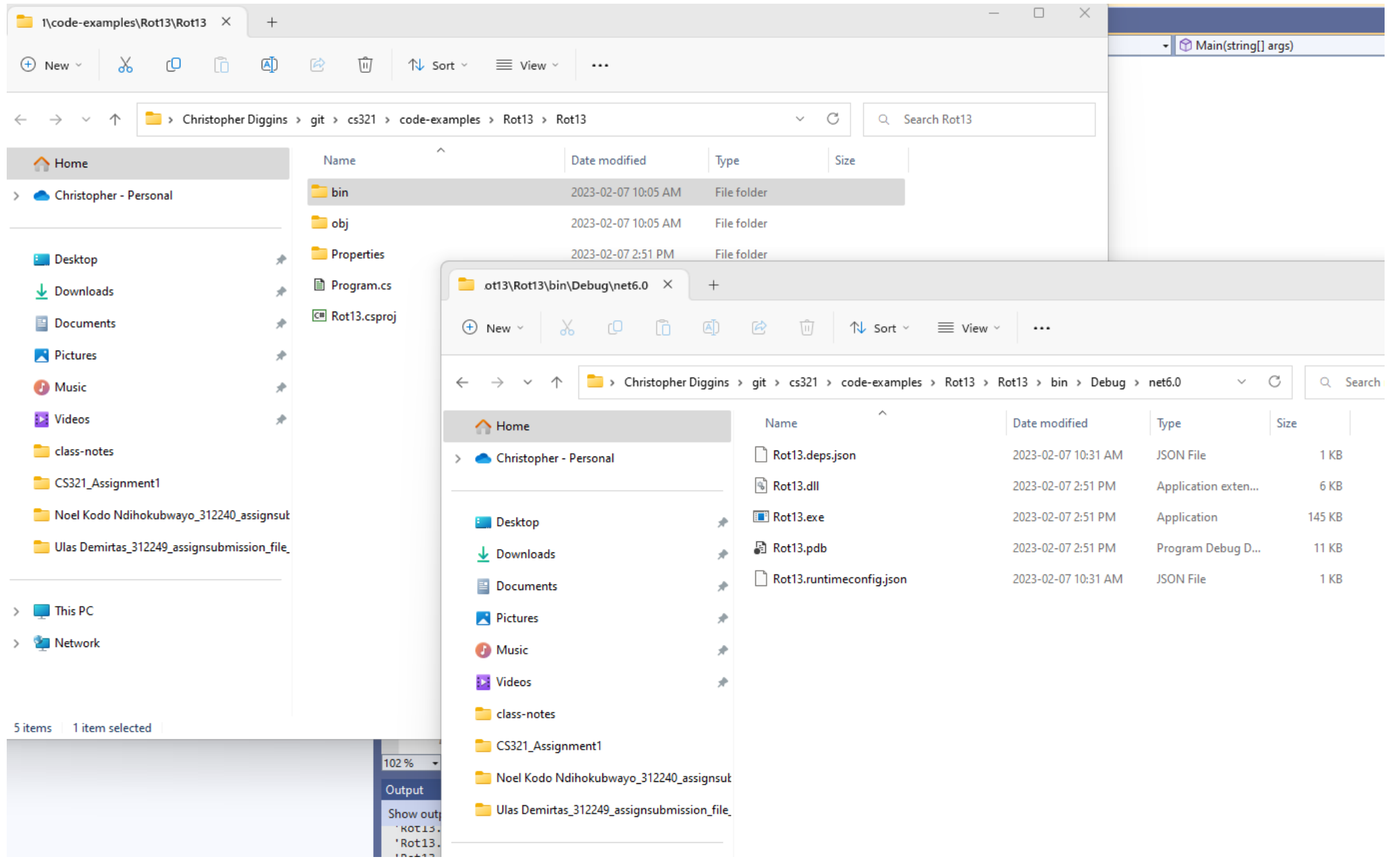


Microsoft Visual Studio Debu

```
hfvat Flfgrz.PbzcbaragZbqry.QngnNaabgngvbaf;

anzrfcnpr Ebg13
{
    choyvp fgngvp pynff Cebtenz
    {
        choyvp fgngvp ibvq Znva(fgevat[] netf)
        {
            vs (netf.Yratgu > 0)
            {
                ine svyrAnzr = netf[0];
                vs (!Svyr.Rkvfgf(svyrAnzr))
                {
                    Pbafbyr.JevgrYvar($"Rkcrpgrq svefg pbzznaq-yvar nethzrag {svyrAnzr}
gb or n inyvq svyranzr");
                    erghea;
                }

                FrgFgnaqneqVachgSebzSvyr(svyrAnzr);
            }

            ine yvar = Pbafbyr.ErnqYvar();
            juvyr (yvar != ahyy)
            {
                Pbafbyr.JevgrYvar(Ebg13(yvar));
                yvar = Pbafbyr.ErnqYvar();
            }
        }

        choyvp fgngvp ibvq FrgFgnaqneqVachgSebzSvyr(fgevat svyrAnzr)
        {
            // Erqverpg gur vachg sebz n fgernz ernqre
            ine svyrErnqre = arj FgernzErnqre(svyrAnzr);
            Pbafbyr.FrgVa(svyrErnqre);
```

SUCCESS! GIBBERISH

Testing from Command Prompt



I wrote a console app for Julius Caesar