TDD

Test Driven Development

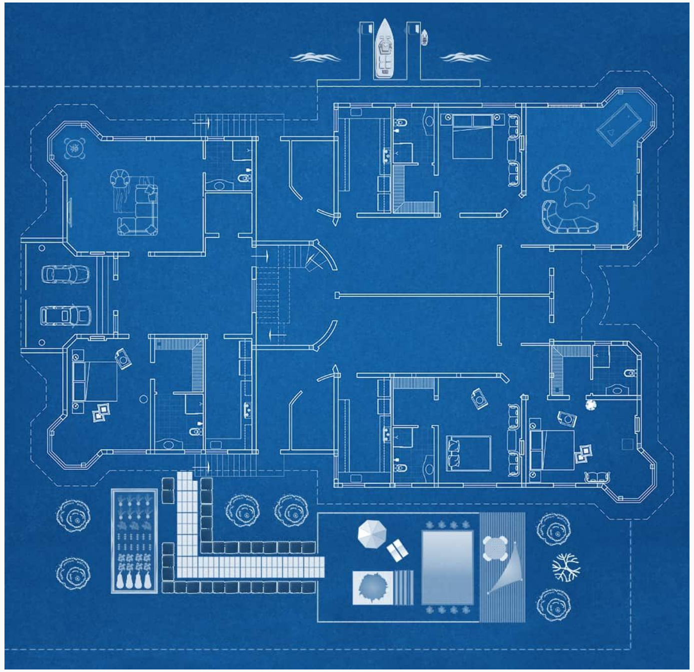
Tests *drive* the development of our code

Test Driven Development

- It **drives the design process of our code and helps with reducing bugs** in the code we write.

Test Driven Development

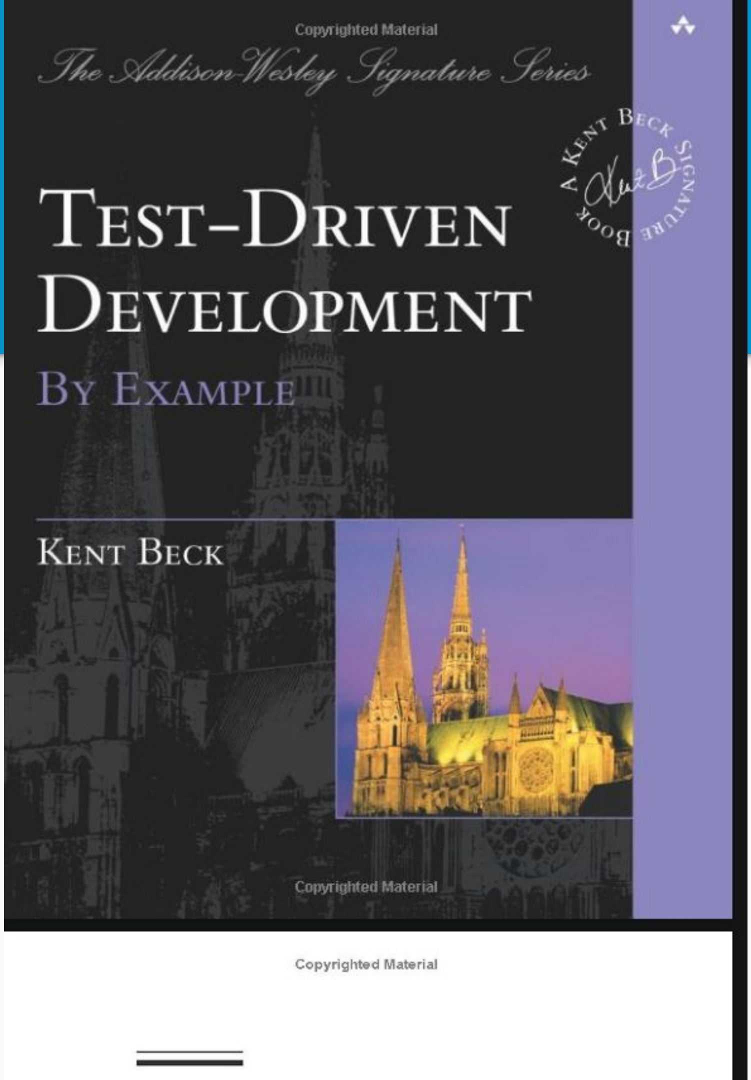
- Clear plan for the functionality of our code
- Makes it simpler and easier to maintain



History

- Many credit **Kent Beck for inventing TDD**, but Kent himself says he merely rediscovered it.

- One of the earliest references of TDD can be found from 1957 in the *Digital Computer Programming D.D. McCracken*, so the process itself is not something new.

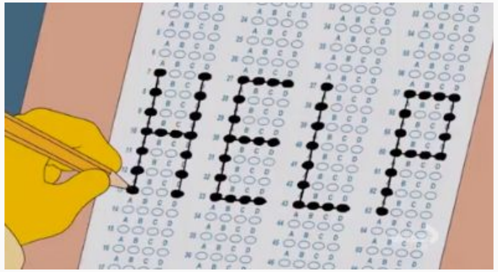


Test Driven Development Advantages

- Avoid bugs
- Clear plan for code functionality
- It encourages decoupling - (which allows changes to be made to one thing without affecting another thing)
- You accumulate tests over time that can be quickly run without the time consuming manual tests

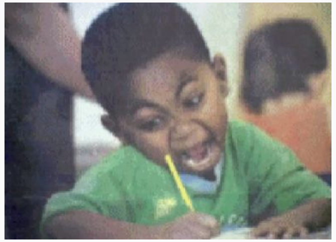
TDD Definition

- **Test-driven development (TDD)** is a software development process relying on software requirements being converted to test cases before software is fully developed, and tracking all software development by repeatedly testing the software against all test cases.



Test First, Code Later

- Tests cases made **1st**
- Software development **2nd**



Test First, Code Later

TDD is a process that relies on the repetition of a very short development cycle:

1. Requirements are turned into very specific test cases
2. Then the software is made so that the tests pass.

Not using TDD:

Would allow software to be added that is not proven to meet requirements.

(Bscly Time wasted)

Using TDD

- The Tests drive our code!

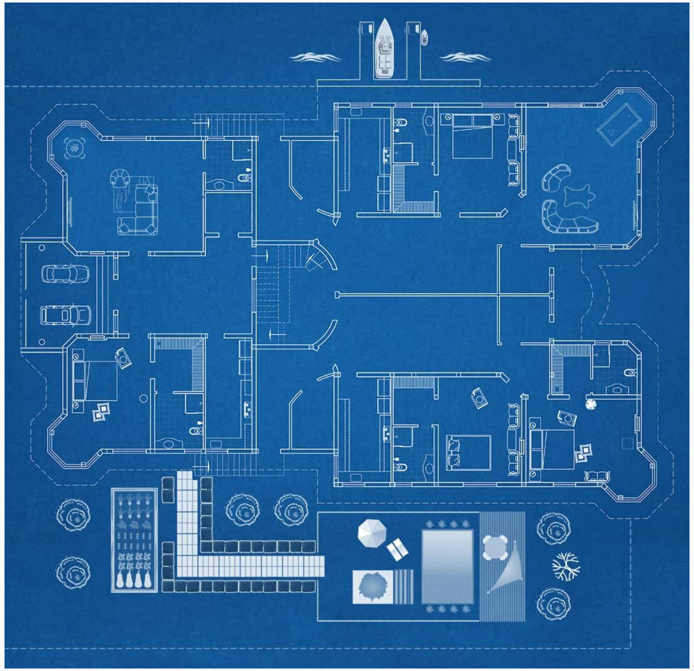
Unit Testing

- Consists of tiny testable parts of a program independently tested for expected functionality.
- Unit testing plays an important role in *Test Driven Development*.
- The purpose is to validate that each unit of the software performs as designed.
- **A unit is the smallest testable part of any software.**

Unit Testing

“Application”

- Test for the kitchen
- Test for the bedroom
- Test for the bathroom
- Test for the living room



How Create a Unit Test:

AAA

How to Create a Unit Test:

1. Arrange
2. Act
3. Assert

1. Arrange

- Prepare your code

2. Act

- Call the method

1. instance.MethodName();
2. Class.MethodName();

3. Assert

- Compare

1. Arrange

- **Prepare your code by:**

Making an instance of the class that will contain the code we are testing

Example:

```
var instance = new ClassName();
```

2. Act

- Call the method

1. instance.MethodName();
2. ClassName.MethodName();

(If testing a static class, you can skip “Arrange”)

3. Assert

- Compare or check against a constant
- Verify that the code we wrote behaves as expected

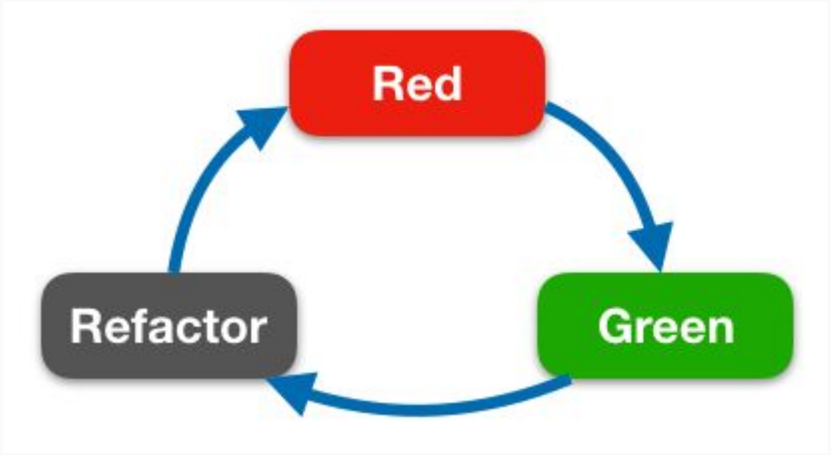
Expected Actual

Writing Unit Tests Process

1. Red
2. Green
3. Refactor

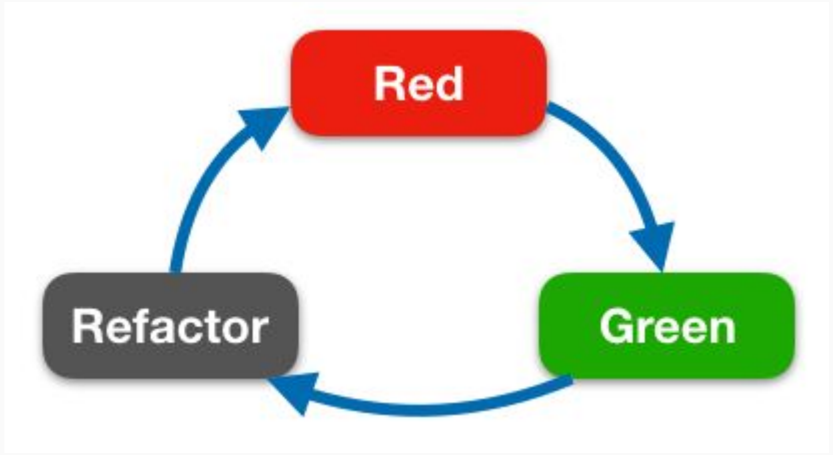
Writing Unit Tests Process

1. Red
2. Green
3. Refactor



3 Step Process!

1. Red
2. Green
3. Refactor

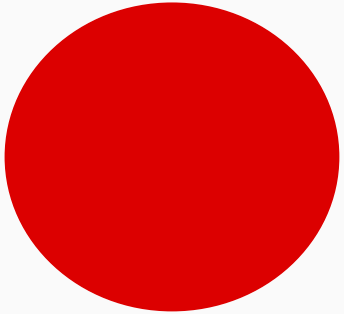


1. Red

Write the test and then watch it fail.

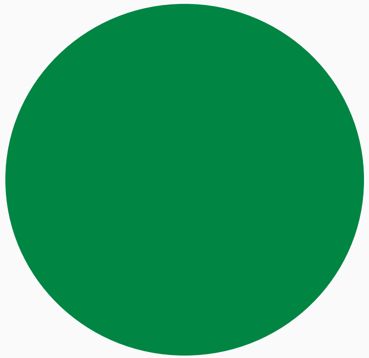
(*it fails because the code needed in order for the test to pass hasn’t been written yet!*).

We create a test that will fail on purpose so that we know with a degree of confidence that our tests will fail when expected to - making it red



2. Green

Now we write code in our software or application so that our test will pass - making it green

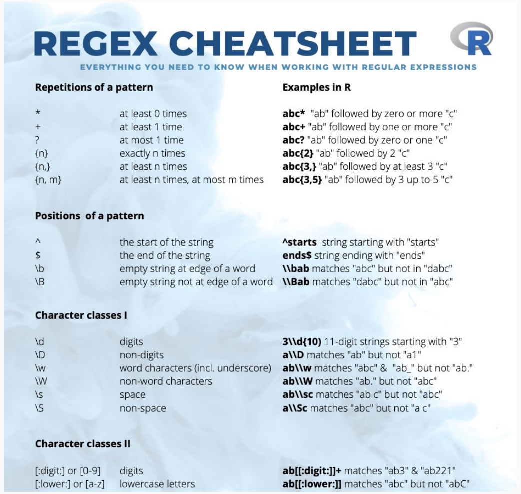


3. Refactor

Change the passing code we wrote without changing the behavior of the code itself.

We want our code to still behave as we expect.

This allows us to improve code readability and possibly reduce complexity.



REGEX CHEATSHEET

EVERYTHING YOU NEED TO KNOW WHEN WORKING WITH REGULAR EXPRESSIONS

Repetitions of a pattern		**Examples in R**
*	at least 0 times	**abc*** "ab" followed by zero or more "c"
+	at least 1 time	**abc+** "ab" followed by one or more "c"
?	at most 1 time	**abc?** "ab" followed by zero or one "c"
{n}	exactly n times	**abc{2}** "ab" followed by 2 "c"
{n,}	at least n times	**abc{3,}** "ab" followed by at least 3 "c"
{n, m}	at least n times, at most m times	**abc{3,5}** "ab" followed by 3 up to 5 "c"

Positions of a pattern

^	the start of the string	**^starts** string starting with "starts"
$	the end of the string	**ends$** string ending with "ends"
\b	empty string at edge of a word	**\\bab** matches "abc" but not in "dabc"
\B	empty string not at edge of a word	**\\Bab** matches "dabc" but not in "abc"

Character classes I

\d	digits	**3\\d{10)** 11-digit strings starting with "3"
\D	non-digits	**a\\D** matches "ab" but not "a1"
\w	word characters (incl. underscore)	**ab\\w** matches "abc" & "ab_" but not "ab."
\W	non-word characters	**ab\\W** matches "ab." but not "abc"
\s	space	**ab\\sc** matches "ab c" but not "abc"
\S	non-space	**a\\Sc** matches "abc" but not "a c"

Character classes II

[:digit:] or [0-9]	digits	**ab[[:digit:]]+** matches "ab3" & "ab221"
[:lower:] or [a-z]	lowercase letters	**ab[[:lower:]]** matches "abc" but not "abC"

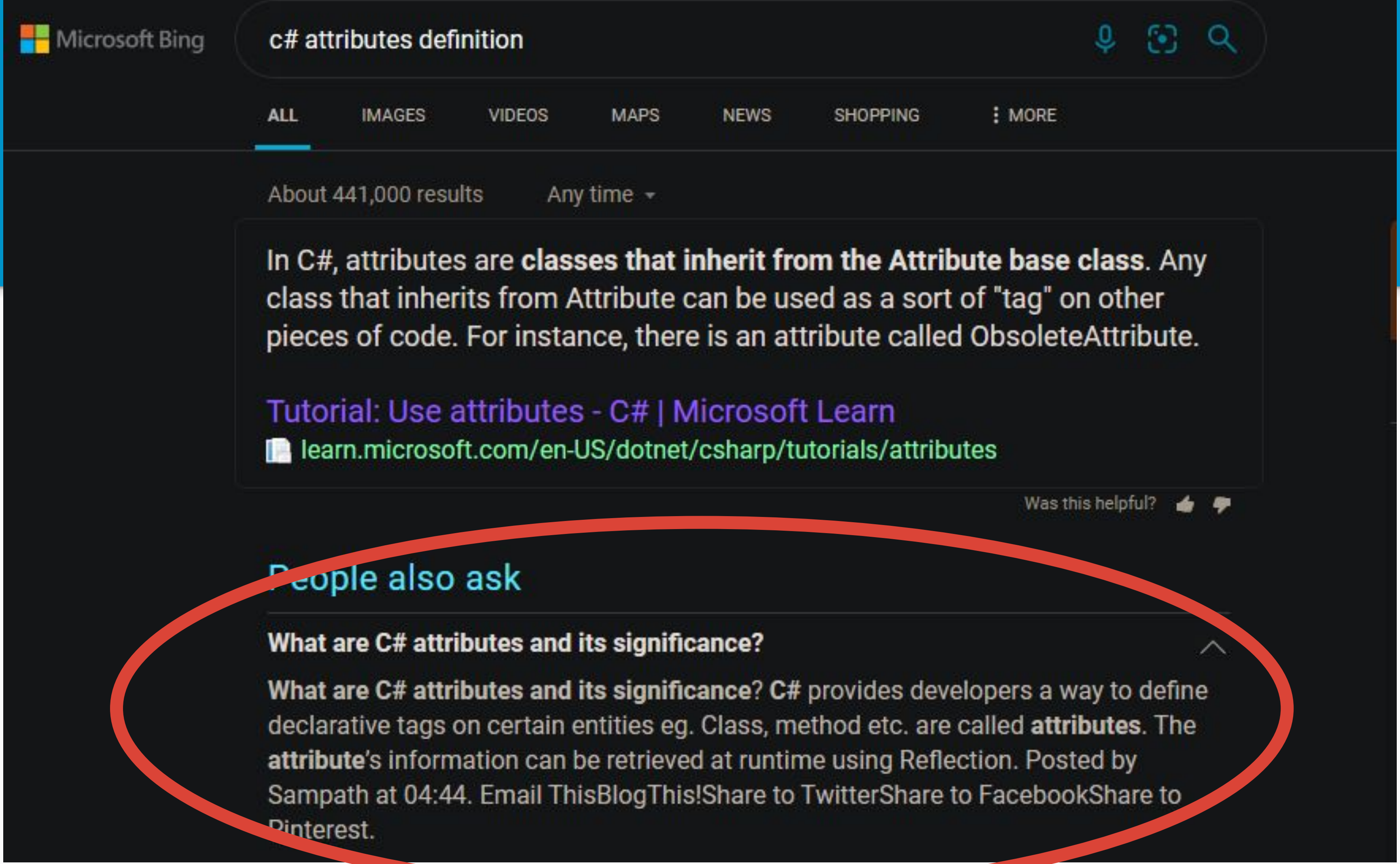
Tool we will use:

XUnit -

Unit testing tool for the .NET framework

Attributes

- Tags that can be added to your code
- We will use them for our tests



c# attributes definition

About 441,000 results

In C#, attributes are **classes that inherit from the Attribute base class**. Any class that inherits from Attribute can be used as a sort of "tag" on other pieces of code. For instance, there is an attribute called ObsoleteAttribute.

Tutorial: Use attributes - C# | Microsoft Learn

learn.microsoft.com/en-US/dotnet/csharp/tutorials/attributes

People also ask

What are C# attributes and its significance?

What are C# attributes and its significance? **C#** provides developers a way to define declarative tags on certain entities eg. Class, method etc. are called **attributes**. The **attribute**'s information can be retrieved at runtime using Reflection. Posted by Sampath at 04:44. Email ThisBlogThis!Share to TwitterShare to FacebookShare to Pinterest.

Testing Methods

- No parameters	→	instance.MethodName();
- With Parameters	→	instance.MethodName(argument);

Testing Methods

-	No parameters	→	instance.MethodName();
-	With Parameters	→	instance.MethodName(argument);

Will use attributes to test!

Attributes

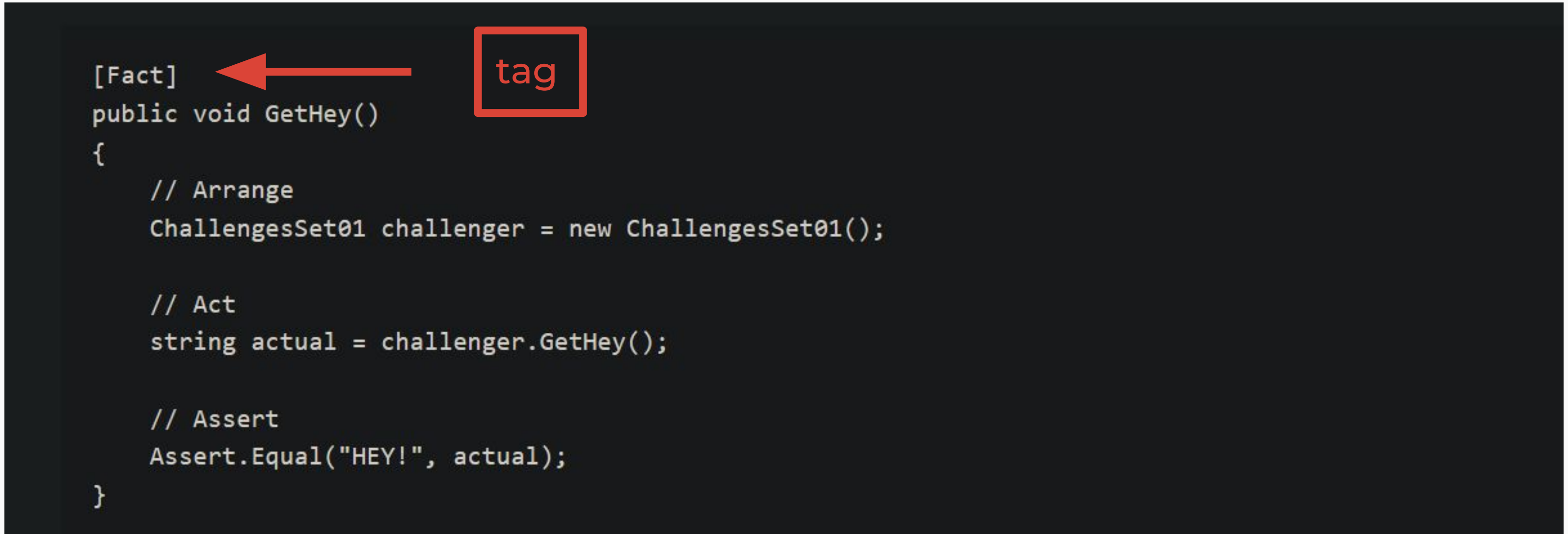
2 tags:

1. Fact
2. Theory

Testing Methods

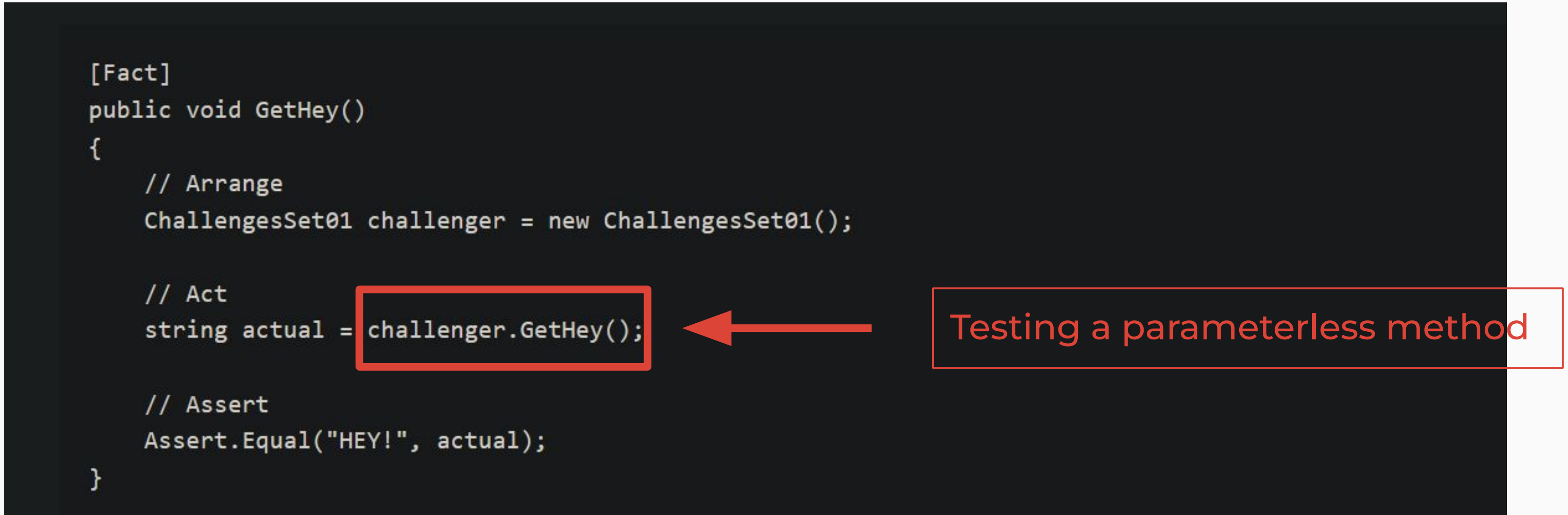
-	No parameters	→	instance.MethodName();	→	Fact
-	With Parameters	→	instance.MethodName(argument);	→	Theory

Fact Attribute



```
[Fact]
public void GetHey()
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    string actual = challenger.GetHey();

    // Assert
    Assert.Equal("HEY!", actual);
}
```

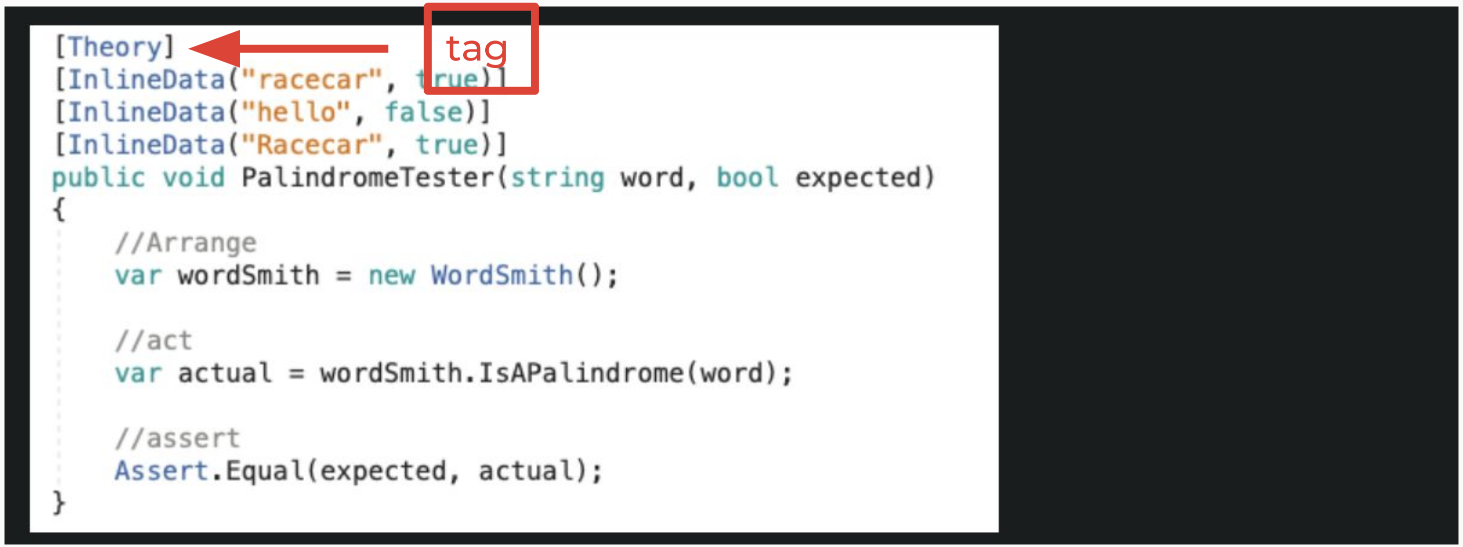
Fact Attribute



```
[Fact]
public void GetHey()
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    string actual = challenger.GetHey();

    // Assert
    Assert.Equal("HEY!", actual);
}
```

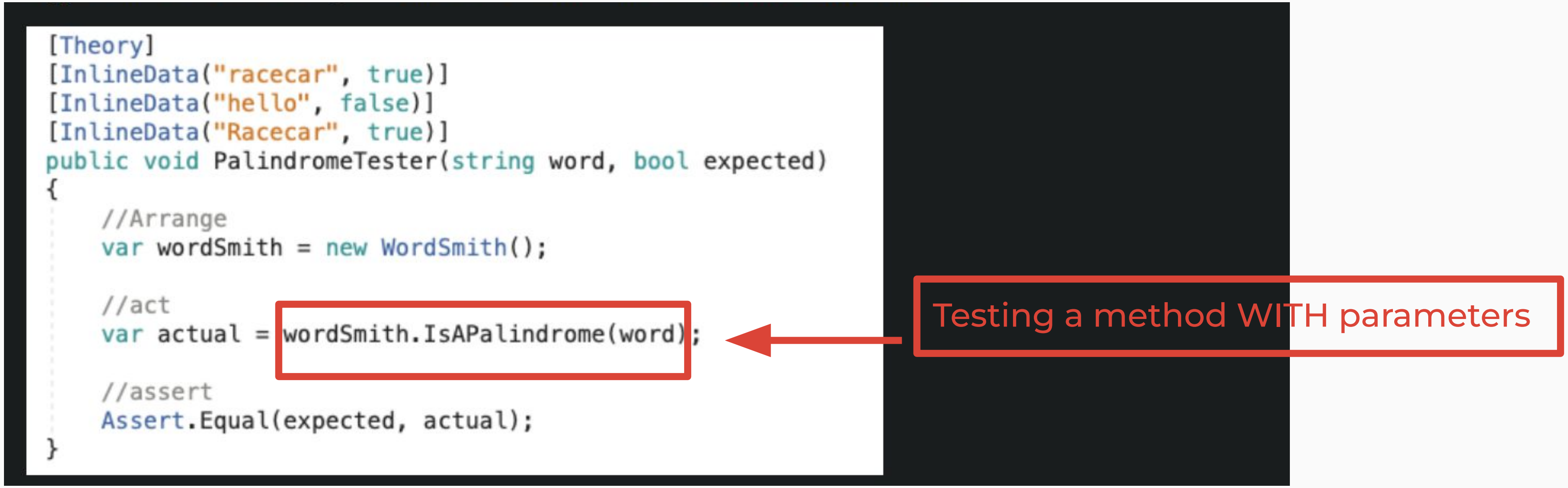
Testing a parameterless method

Theory Attribute



```
[Theory]
[InlineData("racecar", true)]
[InlineData("hello", false)]
[InlineData("Racecar", true)]
public void PalindromeTester(string word, bool expected)
{
    //Arrange
    var wordSmith = new WordSmith();

    //act
    var actual = wordSmith.IsAPalindrome(word);

    //assert
    Assert.Equal(expected, actual);
}
```

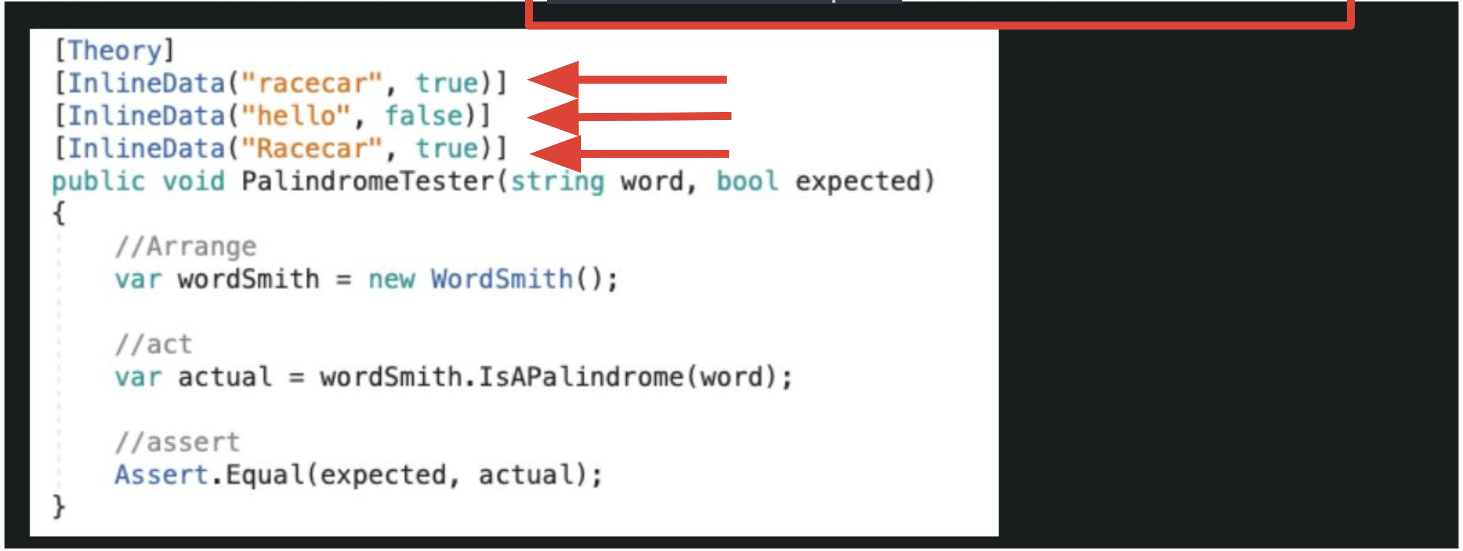
Theory Attribute



```
[Theory]
[InlineData("racecar", true)]
[InlineData("hello", false)]
[InlineData("Racecar", true)]
public void PalindromeTester(string word, bool expected)
{
    //Arrange
    var wordSmith = new WordSmith();

    //act
    var actual = wordSmith.IsAPalindrome(word);

    //assert
    Assert.Equal(expected, actual);
}
```

Theory Attribute

Theory attribute is used when you want to run the same test with different inputs.



```
[Theory]
[InlineData("racecar", true)]
[InlineData("hello", false)]
[InlineData("Racecar", true)]
public void PalindromeTester(string word, bool expected)
{
    //Arrange
    var wordSmith = new WordSmith();

    //act
    var actual = wordSmith.IsAPalindrome(word);

    //assert
    Assert.Equal(expected, actual);
}
```

Scenario:

Let’s say I want to write a function that calculates how much an employee makes per hour given their annual salary amount.

Scenario:

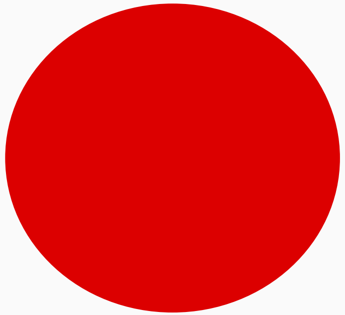
Let’s say I want to write a function that calculates how much an employee makes per hour given their annual salary amount.

TDD - Process

Step 1

TDD - Process

1. **Write my unit test** - and watch it fail -RED- (no code logic is written at this point))



TDD - Process

Step 2

TDD - Process

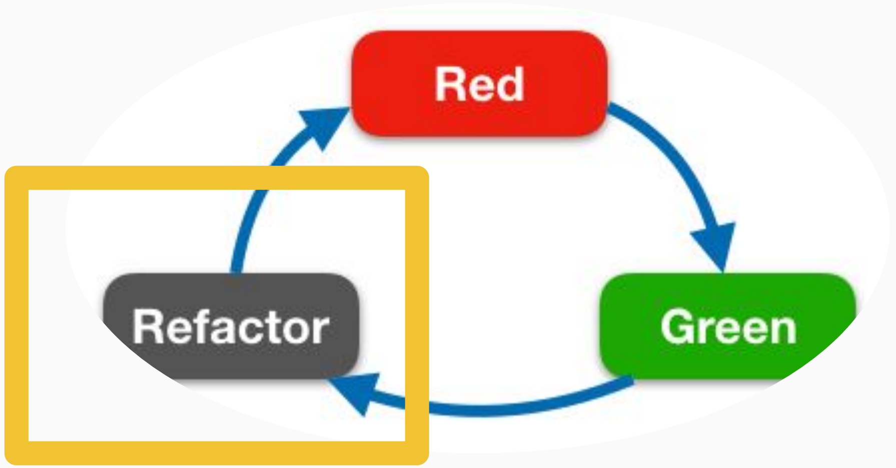
2. **Write the code for the tests!** Define the method, namedHourlyPay, and keep **writing the code until my test passes** -GREEN-

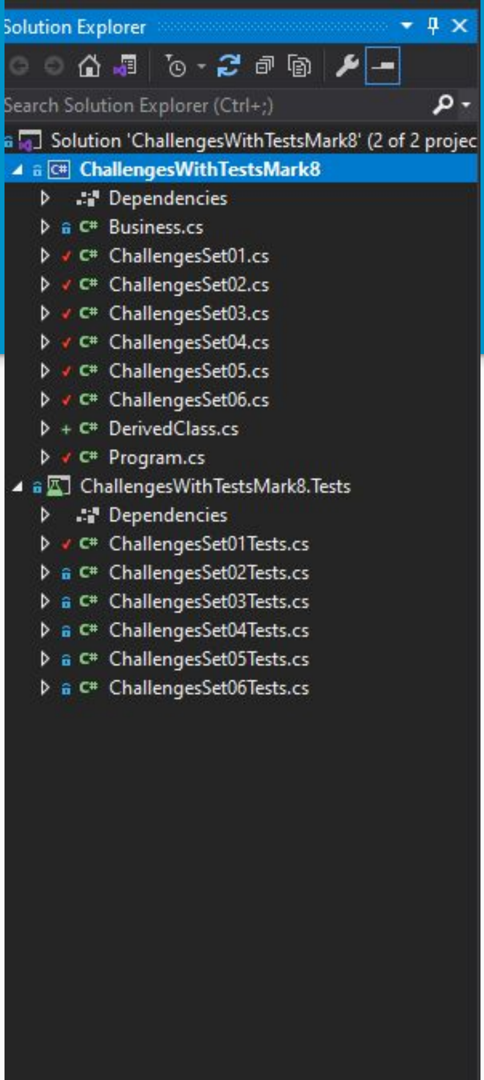
TDD - Process

Step 3

TDD - Process

- **Refactor** - Finally, I could refactor my code for refinement

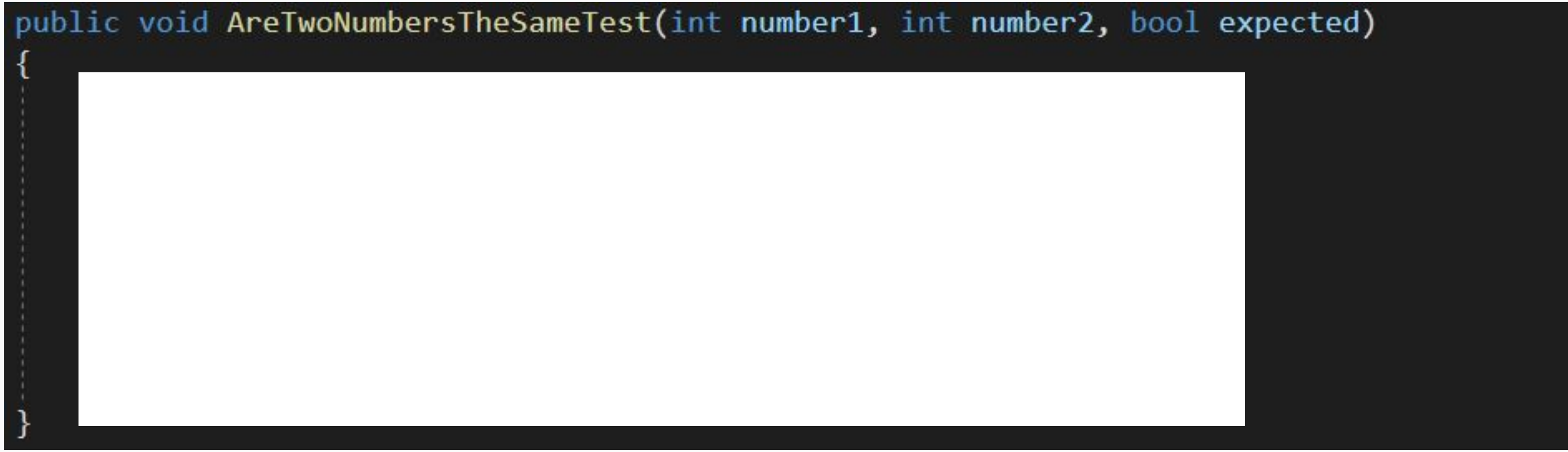



Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'ChallengesWithTestsMark8' (2 of 2 projec
- ChallengesWithTestsMark8
  - Dependencies
  - Business.cs
  - ChallengesSet01.cs
  - ChallengesSet02.cs
  - ChallengesSet03.cs
  - ChallengesSet04.cs
  - ChallengesSet05.cs
  - ChallengesSet06.cs
  - DerivedClass.cs
  - Program.cs
- ChallengesWithTestsMark8.Tests
  - Dependencies
  - ChallengesSet01Tests.cs
  - ChallengesSet02Tests.cs
  - ChallengesSet03Tests.cs
  - ChallengesSet04Tests.cs
  - ChallengesSet05Tests.cs
  - ChallengesSet06Tests.cs

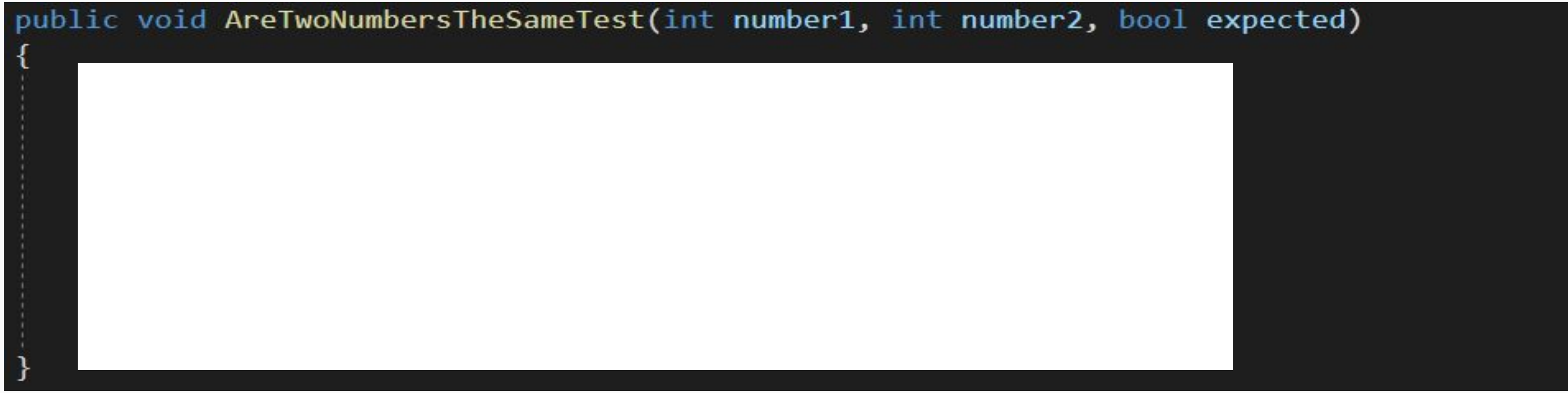
Make a Method to Test your Code



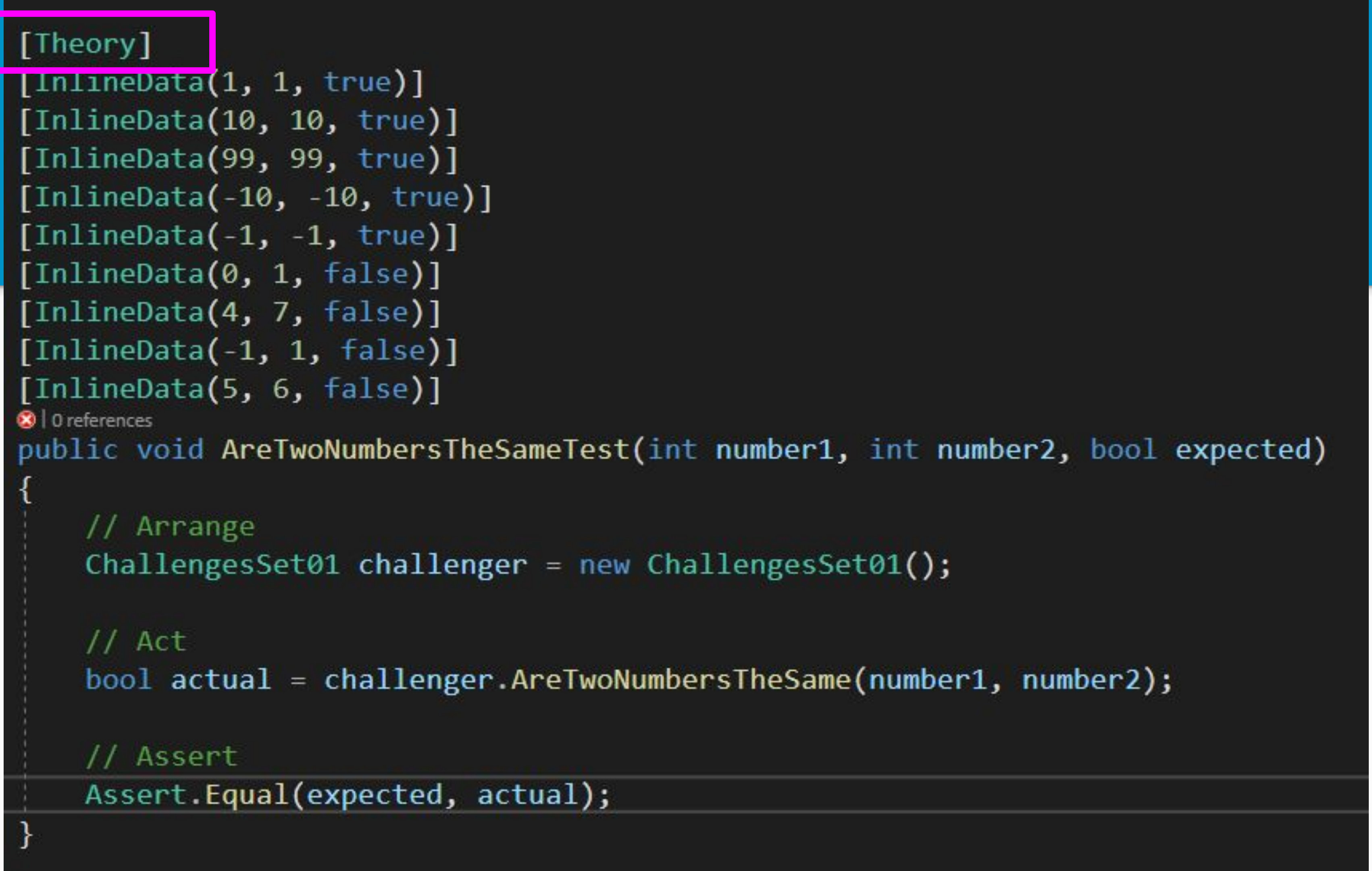
```
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{
}
```

The method will have two parameters

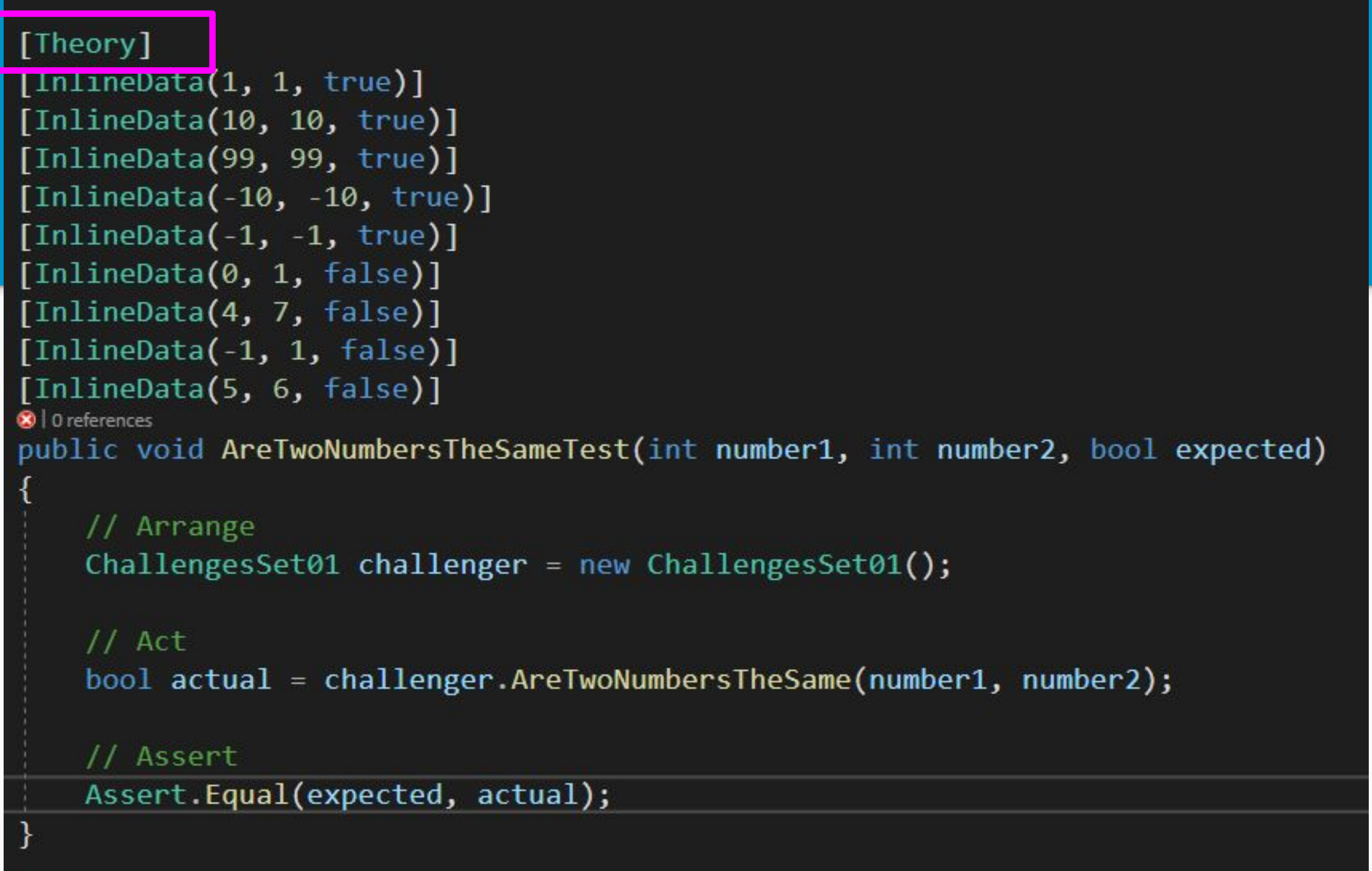
What kind of attribute (or tag) will I use?



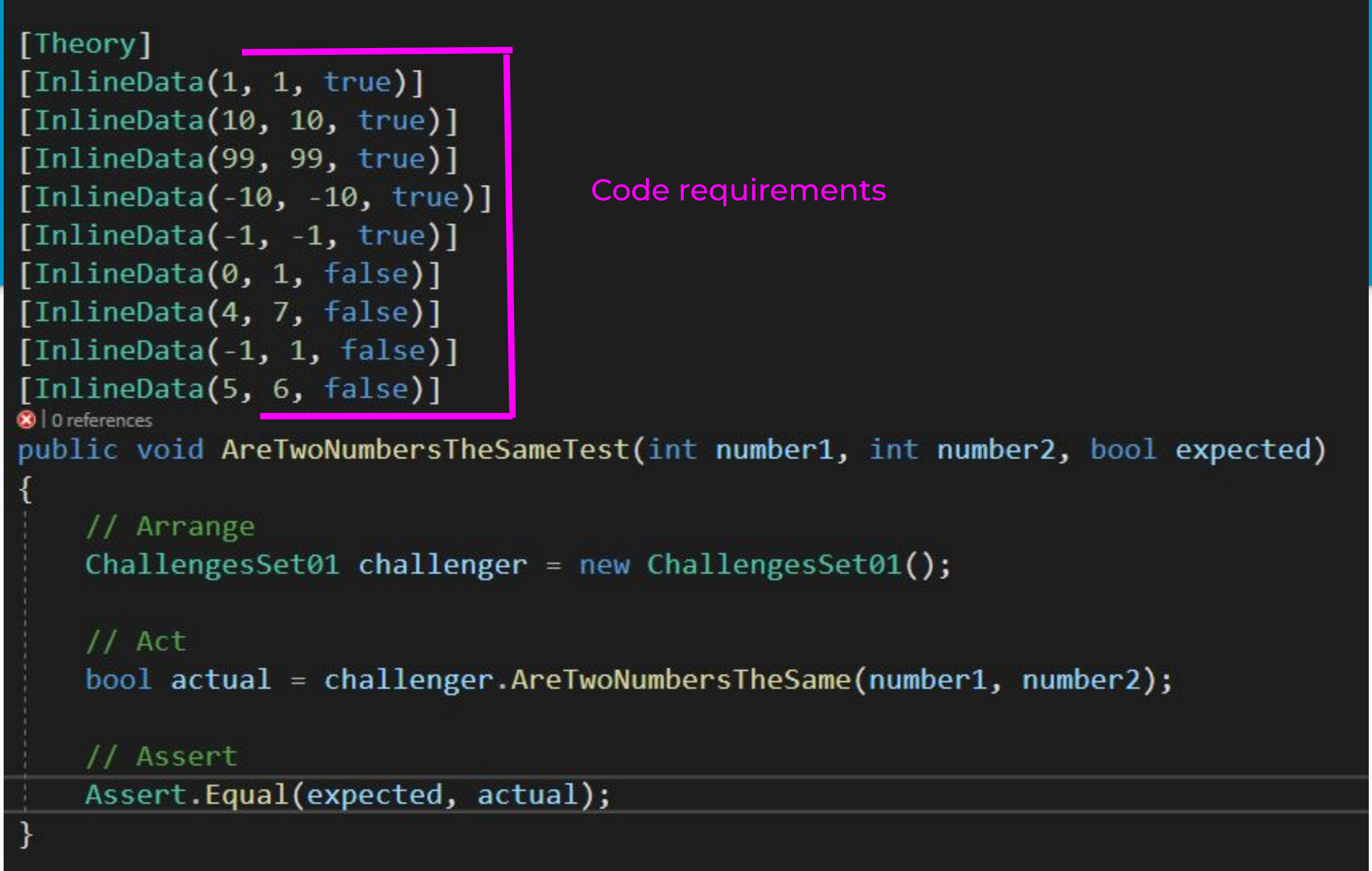
```
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{

}
```



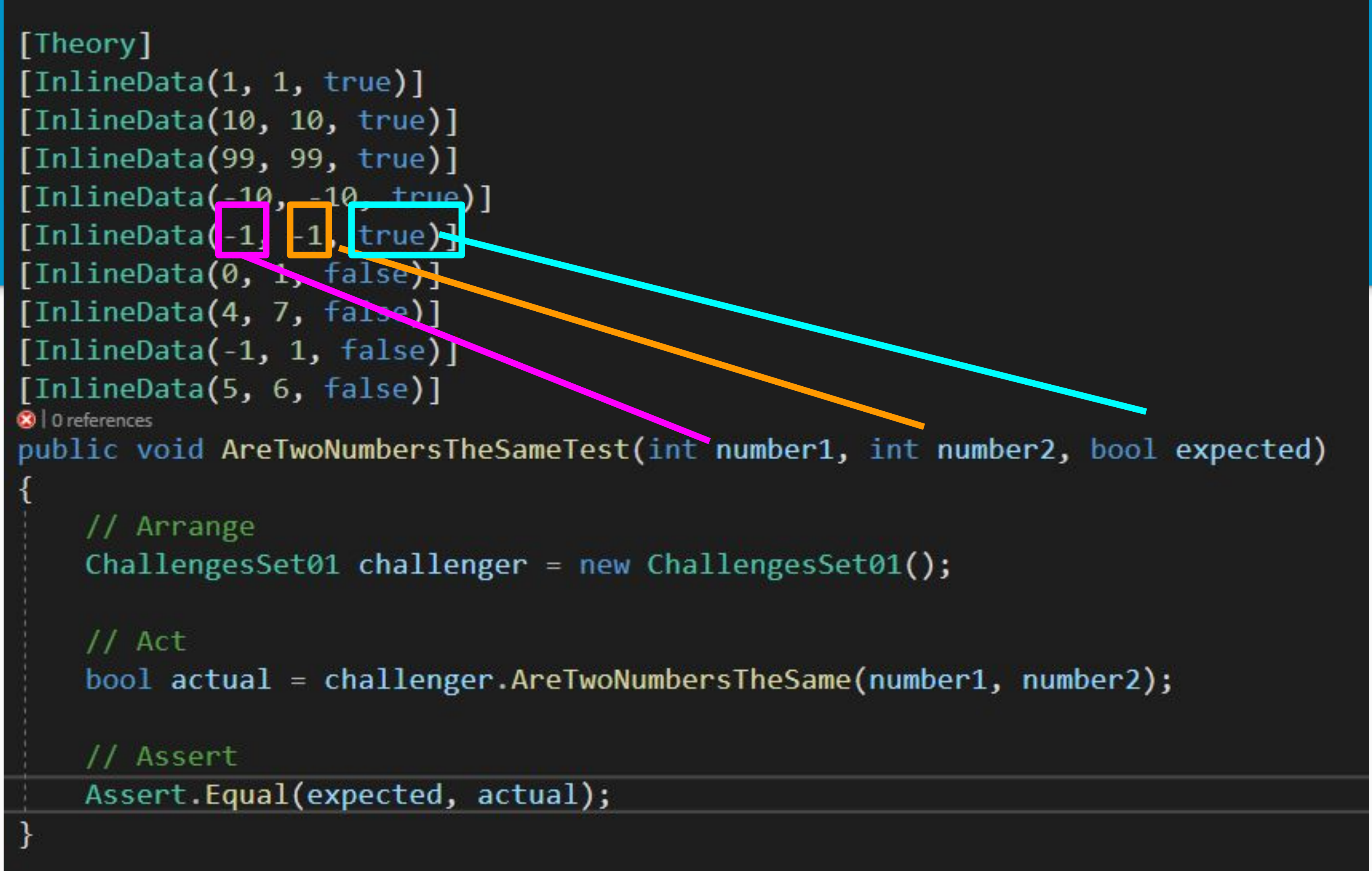
```
[Theory]
[InlineData(1, 1, true)]
[InlineData(10, 10, true)]
[InlineData(99, 99, true)]
[InlineData(-10, -10, true)]
[InlineData(-1, -1, true)]
[InlineData(0, 1, false)]
[InlineData(4, 7, false)]
[InlineData(-1, 1, false)]
[InlineData(5, 6, false)]
| 0 references
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    bool actual = challenger.AreTwoNumbersTheSame(number1, number2);

    // Assert
    Assert.Equal(expected, actual);
}
```

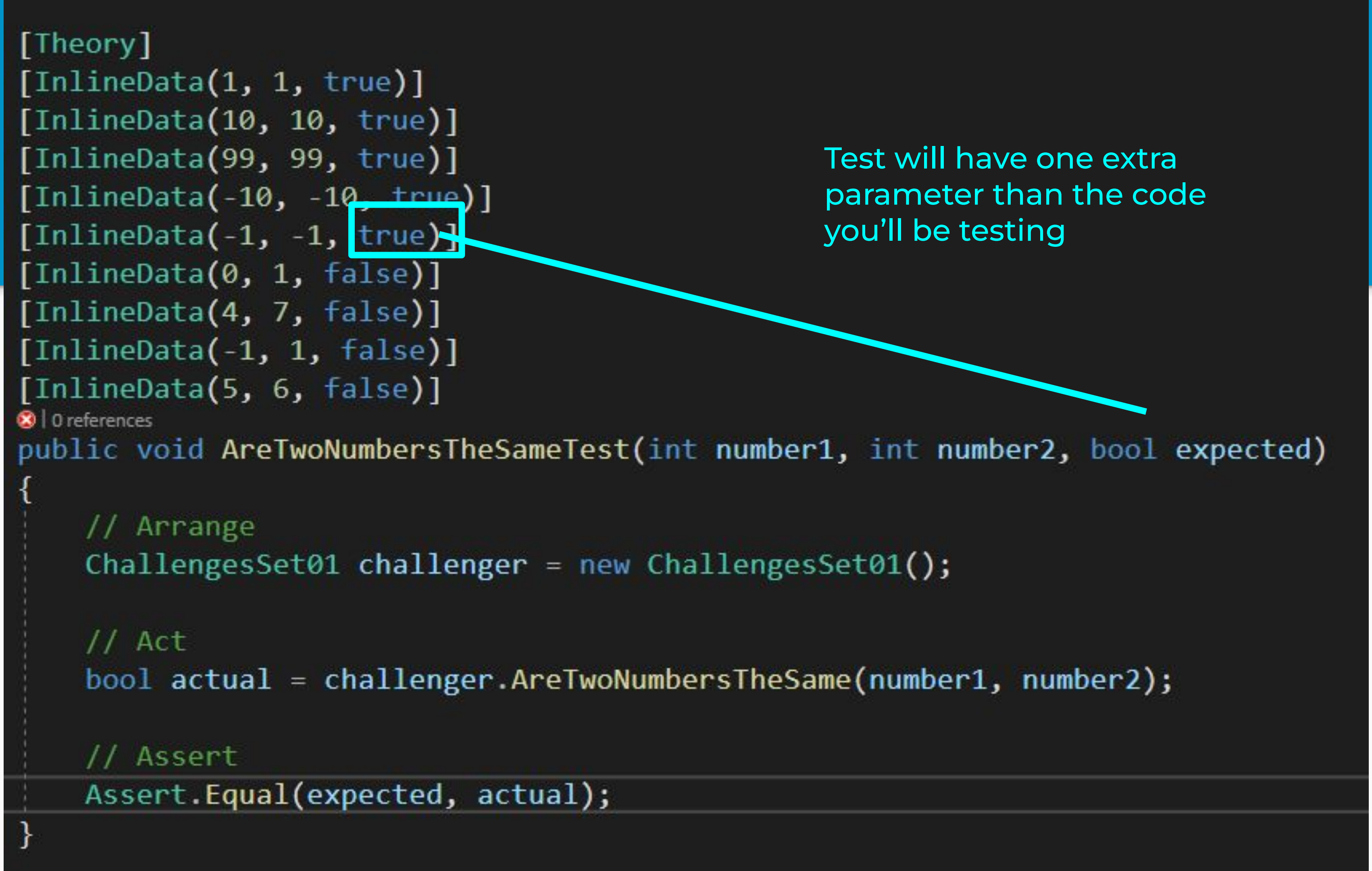


```
[Theory]
[InlineData(1, 1, true)]
[InlineData(10, 10, true)]
[InlineData(99, 99, true)]
[InlineData(-10, -10, true)]
[InlineData(-1, -1, true)]
[InlineData(0, 1, false)]
[InlineData(4, 7, false)]
[InlineData(-1, 1, false)]
[InlineData(5, 6, false)]
| 0 references
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    bool actual = challenger.AreTwoNumbersTheSame(number1, number2);

    // Assert
    Assert.Equal(expected, actual);
}
```



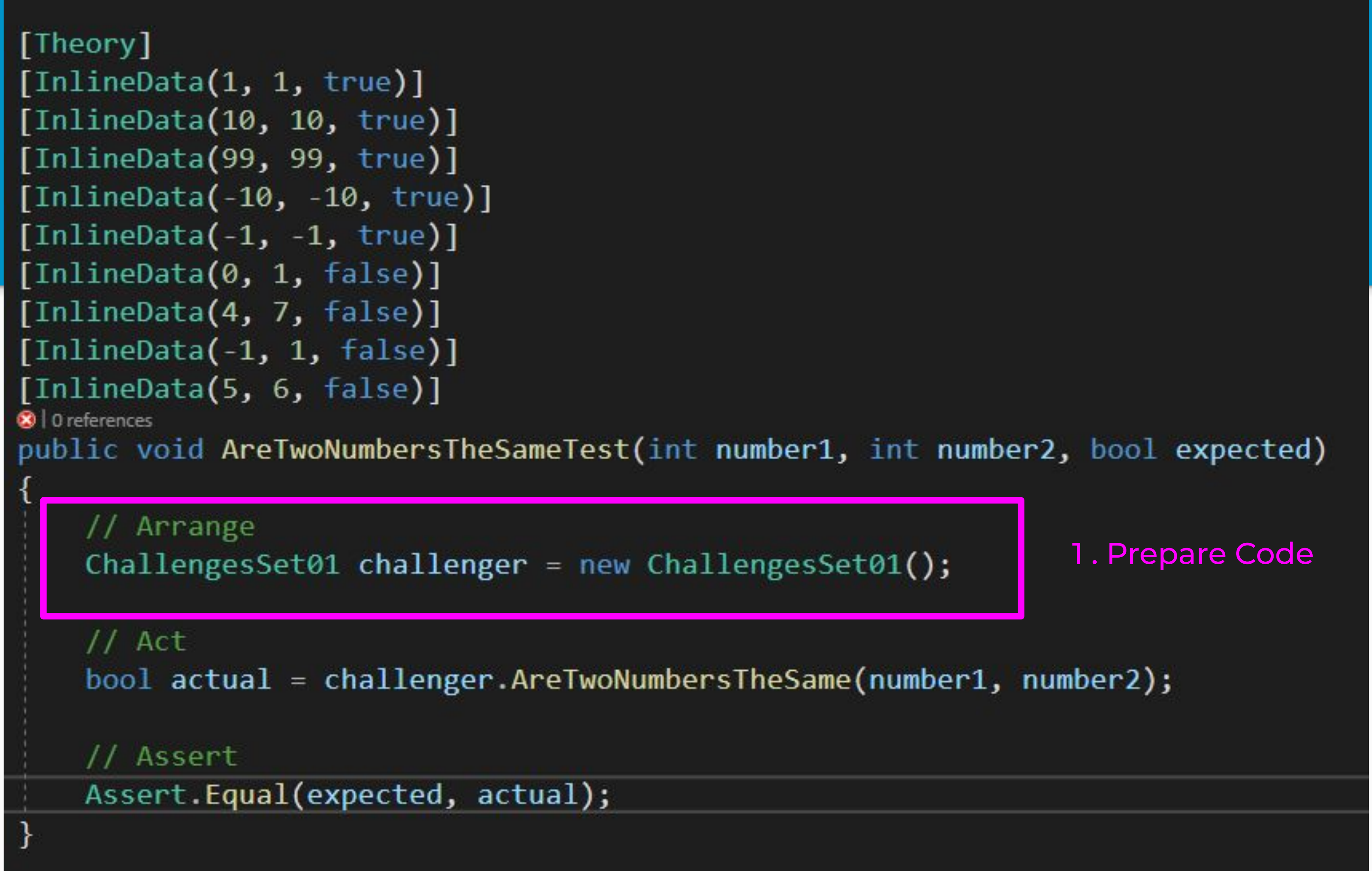
```
[Theory]
[InlineData(1, 1, true)]
[InlineData(10, 10, true)]
[InlineData(99, 99, true)]
[InlineData(-10, -10, true)]
[InlineData(-1, -1, true)]
[InlineData(0, 1, false)]
[InlineData(4, 7, false)]
[InlineData(-1, 1, false)]
[InlineData(5, 6, false)]
0 references
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    bool actual = challenger.AreTwoNumbersTheSame(number1, number2);

    // Assert
    Assert.Equal(expected, actual);
}
```


```
[Theory]
[InlineData(1, 1, true)]
[InlineData(10, 10, true)]
[InlineData(99, 99, true)]
[InlineData(-10, -10, true)]
[InlineData(-1, -1, true)]
[InlineData(0, 1, false)]
[InlineData(4, 7, false)]
[InlineData(-1, 1, false)]
[InlineData(5, 6, false)]
0 references
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    bool actual = challenger.AreTwoNumbersTheSame(number1, number2);

    // Assert
    Assert.Equal(expected, actual);
}
```



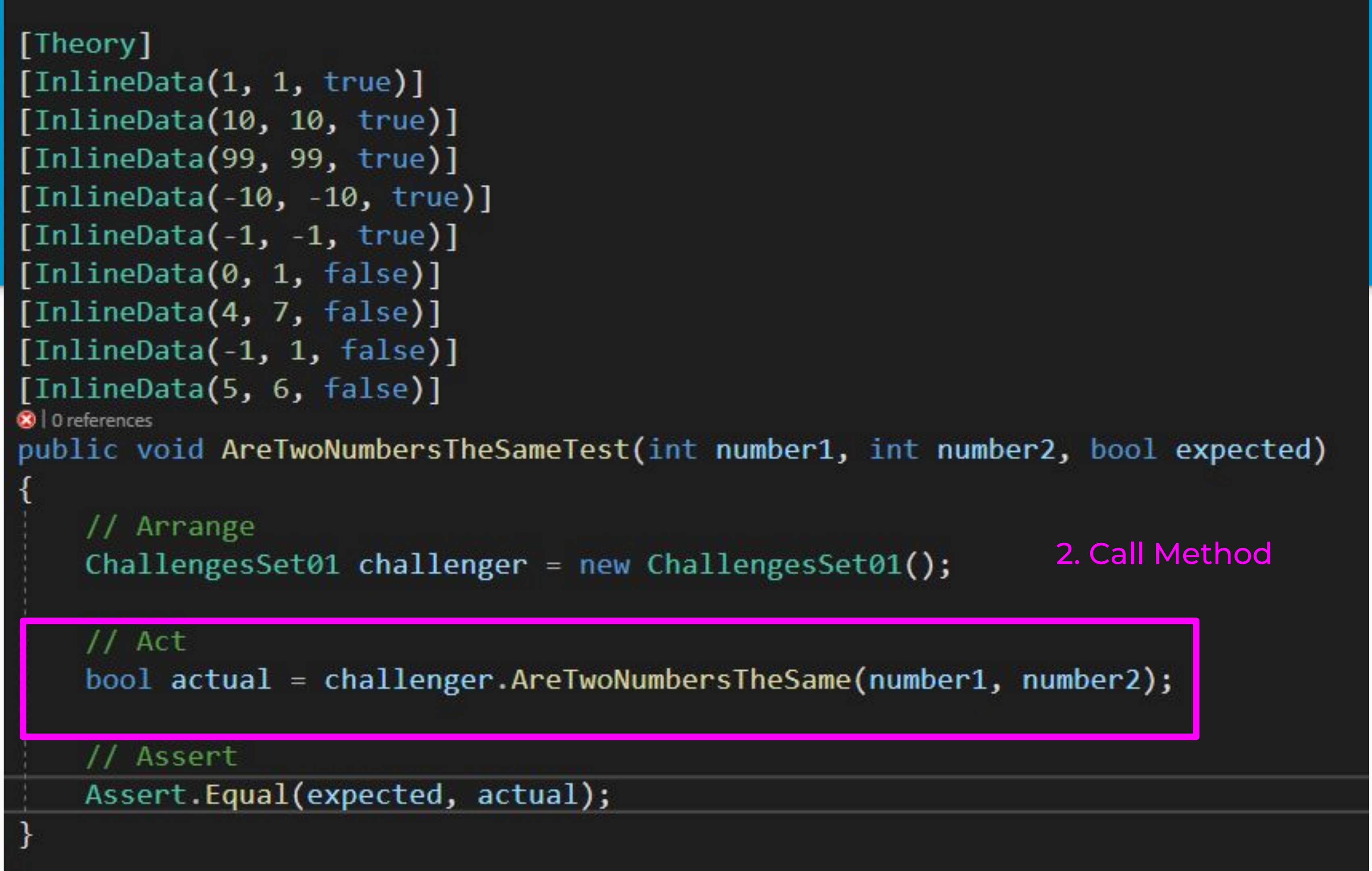
```
[Theory]
[InlineData(1, 1, true)]
[InlineData(10, 10, true)]
[InlineData(99, 99, true)]
[InlineData(-10, -10, true)]
[InlineData(-1, -1, true)]
[InlineData(0, 1, false)]
[InlineData(4, 7, false)]
[InlineData(-1, 1, false)]
[InlineData(5, 6, false)]
0 references
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    bool actual = challenger.AreTwoNumbersTheSame(number1, number2);

    // Assert
    Assert.Equal(expected, actual);
}
```

Test will have one extra parameter than the code you'll be testing

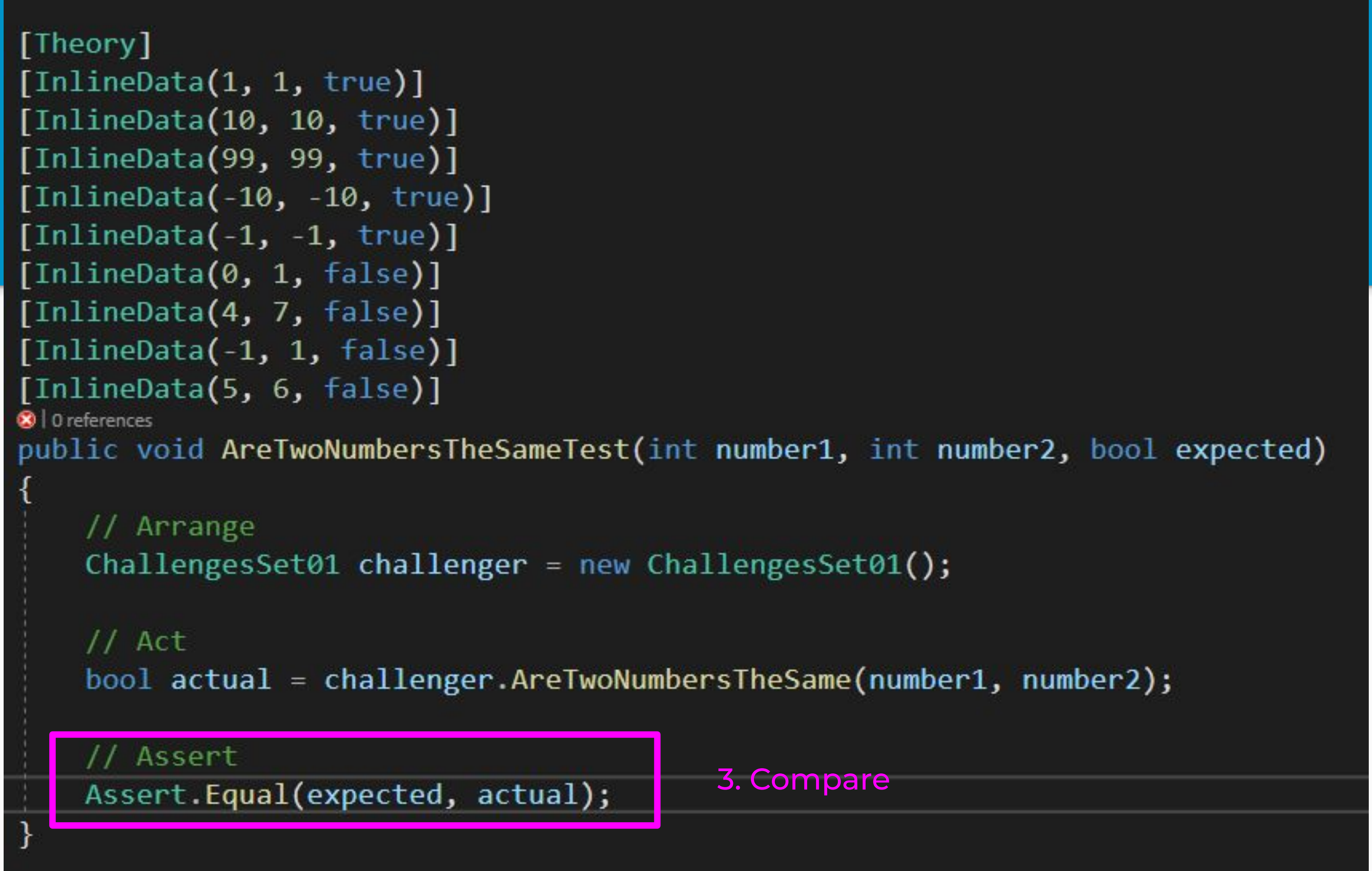


```
[Theory]
[InlineData(1, 1, true)]
[InlineData(10, 10, true)]
[InlineData(99, 99, true)]
[InlineData(-10, -10, true)]
[InlineData(-1, -1, true)]
[InlineData(0, 1, false)]
[InlineData(4, 7, false)]
[InlineData(-1, 1, false)]
[InlineData(5, 6, false)]
0 references
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    bool actual = challenger.AreTwoNumbersTheSame(number1, number2);

    // Assert
    Assert.Equal(expected, actual);
}
```

1. Prepare Code

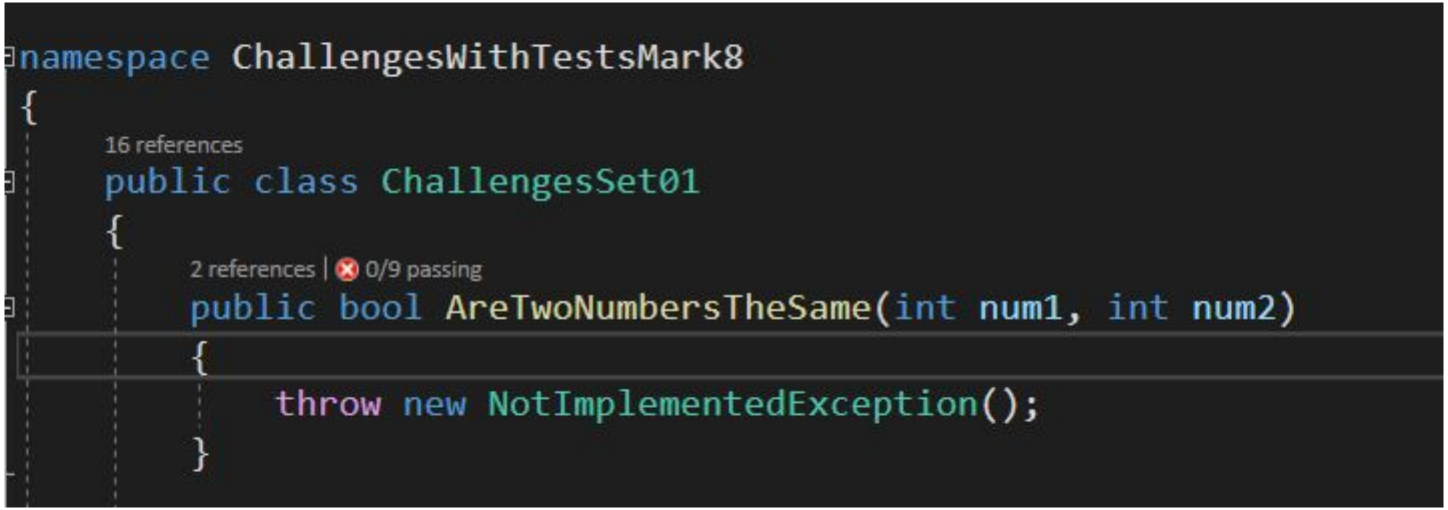


```
[Theory]
[InlineData(1, 1, true)]
[InlineData(10, 10, true)]
[InlineData(99, 99, true)]
[InlineData(-10, -10, true)]
[InlineData(-1, -1, true)]
[InlineData(0, 1, false)]
[InlineData(4, 7, false)]
[InlineData(-1, 1, false)]
[InlineData(5, 6, false)]
| 0 references
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    bool actual = challenger.AreTwoNumbersTheSame(number1, number2);

    // Assert
    Assert.Equal(expected, actual);
}
```



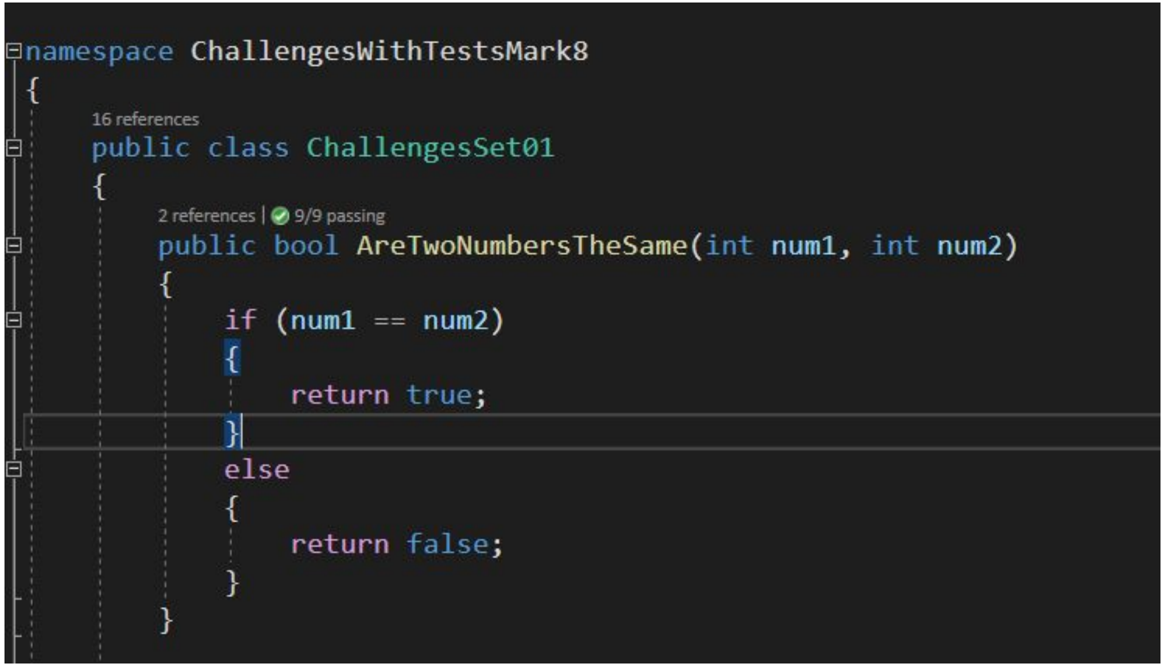
```
[Theory]
[InlineData(1, 1, true)]
[InlineData(10, 10, true)]
[InlineData(99, 99, true)]
[InlineData(-10, -10, true)]
[InlineData(-1, -1, true)]
[InlineData(0, 1, false)]
[InlineData(4, 7, false)]
[InlineData(-1, 1, false)]
[InlineData(5, 6, false)]
| 0 references
public void AreTwoNumbersTheSameTest(int number1, int number2, bool expected)
{
    // Arrange
    ChallengesSet01 challenger = new ChallengesSet01();

    // Act
    bool actual = challenger.AreTwoNumbersTheSame(number1, number2);

    // Assert
    Assert.Equal(expected, actual);
}
```

Red



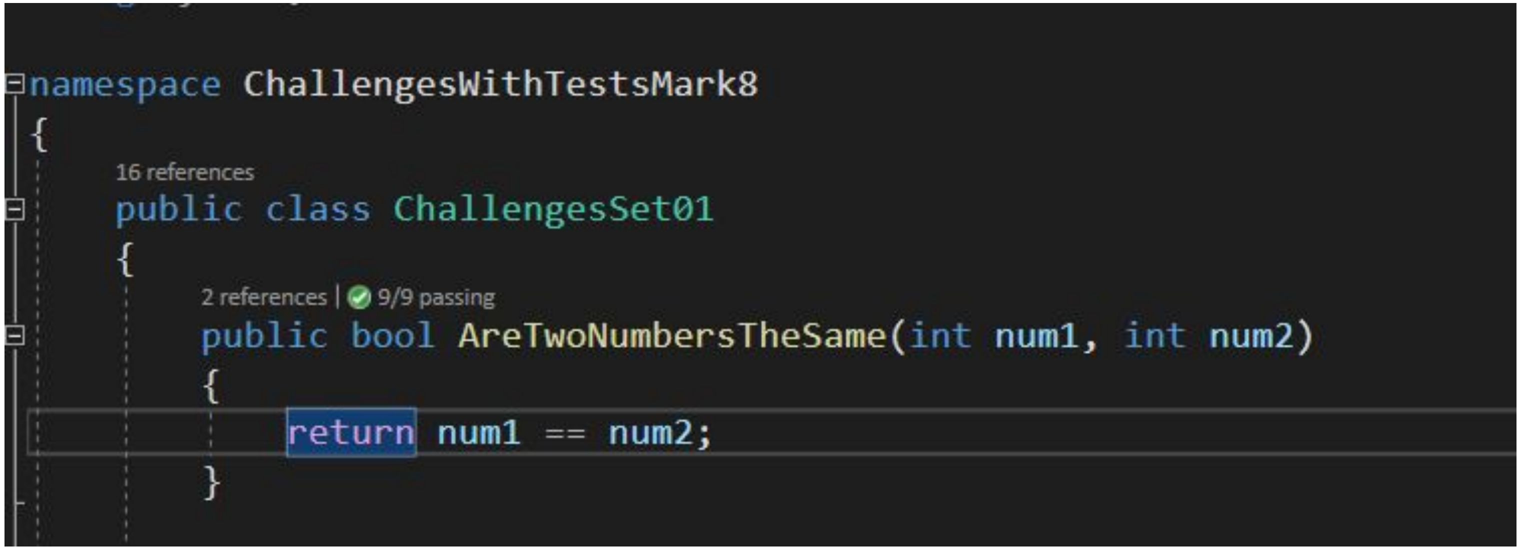
```
namespace ChallengesWithTestsMark8
{
    16 references
    public class ChallengesSet01
    {
        2 references | ✗ 0/9 passing
        public bool AreTwoNumbersTheSame(int num1, int num2)
        {
            throw new NotImplementedException();
        }
```

Green



```
namespace ChallengesWithTestsMark8
{
    16 references
    public class ChallengesSet01
    {
        2 references | 9/9 passing
        public bool AreTwoNumbersTheSame(int num1, int num2)
        {
            if (num1 == num2)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

Refactor

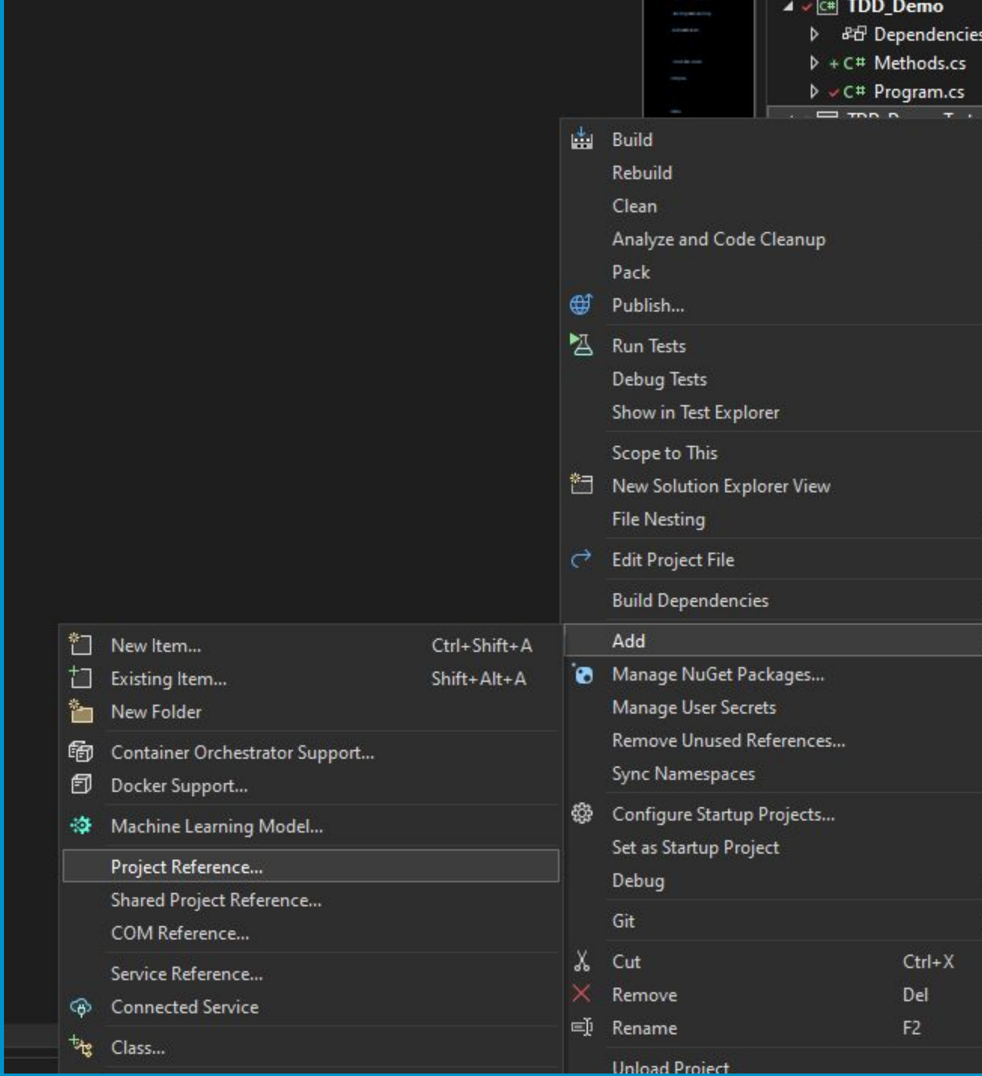


```
namespace ChallengesWithTestsMark8
{
    16 references
    public class ChallengesSet01
    {
        2 references | 9/9 passing
        public bool AreTwoNumbersTheSame(int num1, int num2)
        {
            return num1 == num2;
        }
```

Setup

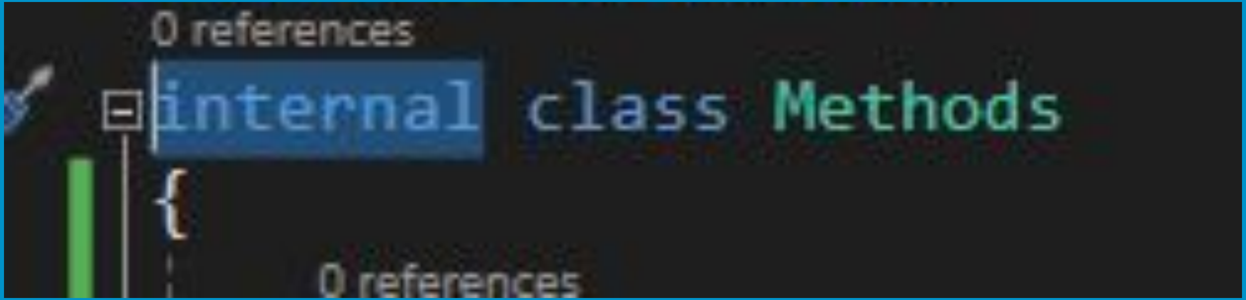
Tests will reference the code

- Right click on test project file
- Click add
- Project Reference



Reminder

Make sure the class is public



```
0 references
internal class Methods
{
        0 references
```