

# Exercise - Debug with Visual Studio Code

20 minutes

100 XP

It's time to put your newly acquired debugging knowledge into practice. It's your first day on the job, and it's time to put your .NET debugging skills to work by fixing a bug in the company's flagship product, a Fibonacci calculator.

## Create a sample .NET project for debugging

To set up Visual Studio Code for .NET debugging, we'll first need a .NET project. Visual Studio Code includes an integrated terminal, which makes creating a new project easy.

1. In Visual Studio Code, select **File > Open Folder**.
2. Create a new folder named `dotNetDebugging` in the location of your choice. Then choose **Select Folder**.
3. Open the integrated terminal from Visual Studio Code by selecting **View > Terminal** from the main menu.
4. In the terminal window, copy and paste the following command:

```
.NET CLI
```

```
dotnet new console
```

Copy

This command creates a `Program.cs` file in your folder with a basic "Hello World" program already written. It also creates a C# project file named `DotNetDebugging.csproj`.

5. In the terminal window, copy and paste the following command to run the "Hello World" program.

```
.NET CLI
```

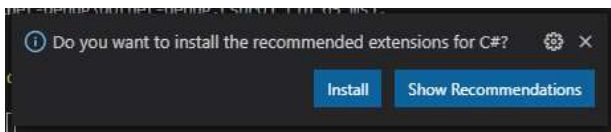
```
dotnet run
```

Copy

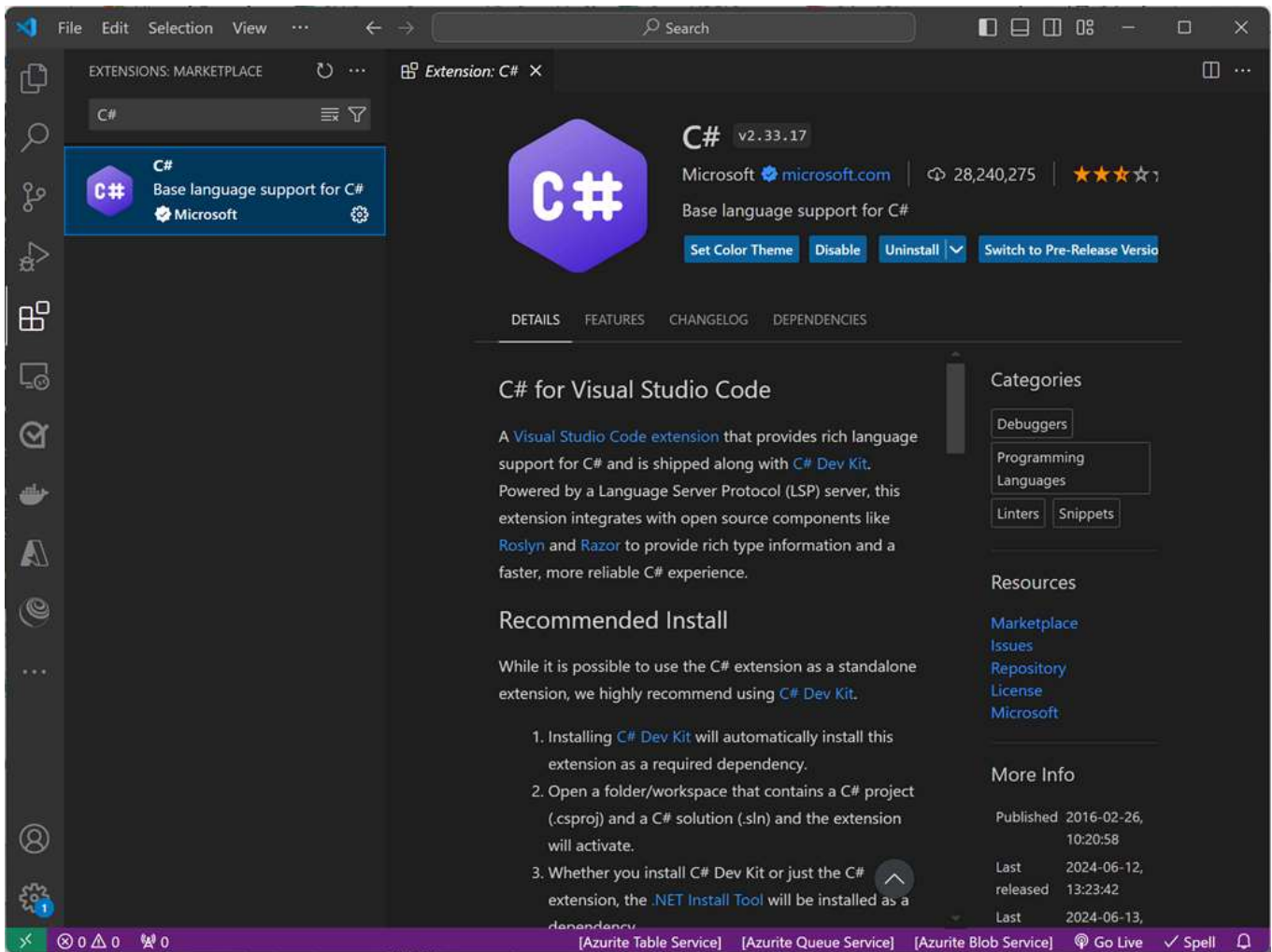
The terminal window displays "Hello World!" as output.

## Set up Visual Studio Code for .NET debugging

1. Open `Program.cs` by selecting it.
2. The first time you open a C# file in Visual Studio Code, you'll receive a prompt to install recommended extensions for C#. If you see this prompt, select the **Install** button in the prompt.



3. Visual Studio Code will install the C# extension and show another prompt to add required assets to build and debug your project. Select the **Yes** button.



4. You can close the **Extension: C#** tab to focus on the code we'll be debugging.

## Add the Fibonacci program logic

Our current project writes a "Hello World" message to the console, which doesn't give us much to debug. Instead, you'll use a short .NET program to compute the  $N^{\text{th}}$  number of the Fibonacci sequence.

The Fibonacci sequence is a suite of numbers that starts with the numbers 0 and 1, with every other following number being the sum of the two previous ones. The sequence continues as shown here:

text

Copy

0, 1, 1, 2, 3, 5, 8, 13, 21...

1. Open **Program.cs** by selecting it.
2. Replace the contents of **Program.cs** with the following code:

Copy

```
C#  
  
int result = Fibonacci(5);  
Console.WriteLine(result);  
  
static int Fibonacci(int n)  
{  
    int n1 = 0;  
    int n2 = 1;  
    int sum;  
  
    for (int i = 2; i < n; i++)  
    {  
        sum = n1 + n2;  
        n1 = n2;
```

```

        n2 = sum;
    }

    return n == 0 ? n1 : n2;
}

```

## Note

This code contains an error, which we'll debug later in this module. We don't recommend that you use it in any mission-critical Fibonacci applications until we get that bug fixed.

3. Save the file by selecting `ctrl+s` for Windows and Linux. Select `cmd+s` for Mac.

4. Let's take a look at how the updated code works before we debug it. Run the program by entering the following command in the terminal:

```
.NET CLI
```

```
dotnet run
```

Copy

The screenshot shows the Visual Studio Code interface. The Explorer pane on the left shows the project structure with files like `.vscode`, `bin`, `obj`, `DotNetDebugging.csproj`, and `Program.cs`. The main editor displays `Program.cs` with the following code:

```

1  int result = Fibonacci(5);
2  Console.WriteLine(result);
3
4  static int Fibonacci(int n)
5  {
6      int n1 = 0;
7      int n2 = 1;
8      int sum;
9
10     for (int i = 2; i < n; i++)
11     {
12         sum = n1 + n2;
13         n1 = n2;
14         n2 = sum;
15     }
16
17     return n == 0 ? n1 : n2;
18 }

```

The bottom panel shows the Terminal with the following output:

```

C:\Users\Jon\Desktop\DotNetDebugging>dotnet run
Hello, World!

C:\Users\Jon\Desktop\DotNetDebugging>dotnet run
3

C:\Users\Jon\Desktop\DotNetDebugging>

```

The status bar at the bottom indicates the current position is Line 18, Column 2, with 4 spaces, UTF-8 with BOM encoding, and CRLF line endings.

5. The result, 3, is shown in the terminal output. When you consult this Fibonacci sequence chart that shows the zero-based sequence position for each value in parenthesis, you'll see that the result should have been 5. It's time to get familiar with the debugger and fix this program.

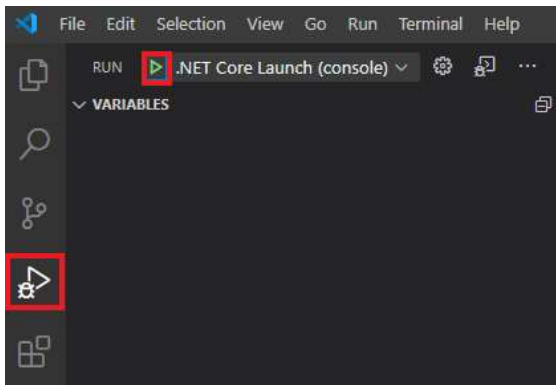
```
text
```

```
0 (0), 1 (1), 1 (2), 2 (3), 3 (4), 5 (5), 8 (6), 13 (7), 21 (8)...
```

Copy

# Analyze the issues

1. Start the program by selecting the **Run and Debug** tab on the left, then selecting the **Start debugging** button. You might need to select the **Run and Debug** button first, then select the `Program.cs` file.



You should see the program finish quickly. That's normal because you haven't added any breakpoints yet.

2. If the debug console doesn't appear, select `Ctrl+Shift+Y` for Windows and Linux or `Cmd+Shift+Y` for Mac. You should see several lines of diagnostic information, followed by these lines at the end:

Output

Copy

```
...
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.0\System.Threading.dll'. Skipped loading symbols. Module is optimized an
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.0\System.Text.Encoding.Extensions.dll'. Skipped loading symbols. Module
3
The program '[88820] DotNetDebugging.dll' has exited with code 0 (0x0).
```

The lines at the top tell you that the default debugging settings enable the "Just My Code" option. This means the debugger will only debug your code and won't step into the source code for .NET unless you disable this mode. This option allows you to focus on debugging your code.

At the end of the debug console output, you'll see the program writes 3 to the console and then exits with code 0. Usually a program exit code of 0 indicates that the program ran and exited without crashing. However, there's a difference between crashing and returning the correct value. In this case, we asked the program to calculate the fifth value of the Fibonacci sequence:

text

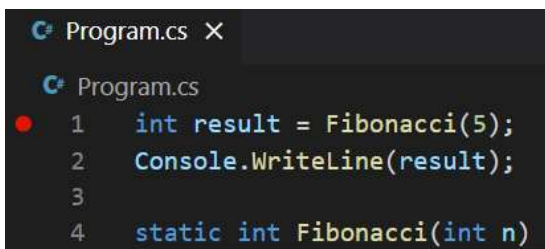
Copy

```
0 (0), 1 (1), 1 (2), 2 (3), 3 (4), 5 (5), 8 (6), 13 (7), 21 (8)...
```

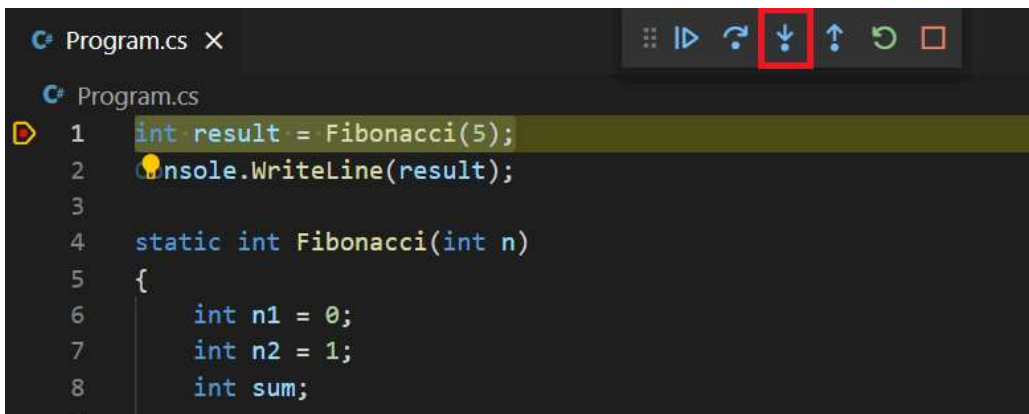
The fifth value in this list is 5, but our program returned 3. Let's use the debugger to diagnose and fix this error.

## Use breakpoints and step-by-step execution

1. Add a breakpoint by clicking in the left margin at line 1 on `int result = Fibonacci(5);`.



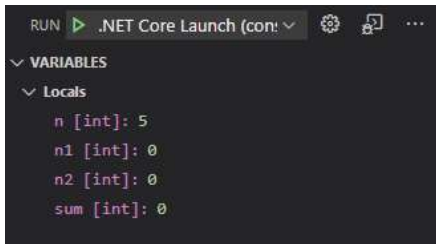
2. Start debugging again. The program begins to execute. It breaks (pauses execution) on line 1 because of the breakpoint you set. Use the debugger controls to step into the `Fibonacci()` function.



```
Program.cs
1 int result = Fibonacci(5);
2 Console.WriteLine(result);
3
4 static int Fibonacci(int n)
5 {
6     int n1 = 0;
7     int n2 = 1;
8     int sum;
```

## Check the variables state

Now, take some time to inspect the different variables' values by using the **Variables** panel.



```
RUN .NET Core Launch (con: v
VARIABLES
  Locals
    n [int]: 5
    n1 [int]: 0
    n2 [int]: 0
    sum [int]: 0
```

- What is the value shown for the `n` parameter?
- At the beginning of the function's execution, what are the values for the local variables `n1`, `n2`, and `sum`?

1. Next, we'll advance into the `for` loop by using the **Step Over** debugger control.



2. Continue advancing until you hit the first line inside of the `for` loop, on the line that reads:

```
C#
sum = n1 + n2;
```

Copy

### Note

You might have noticed that to move through the `for(...) {}` line requires multiple steps in commands. This situation occurs because there are multiple *statements* on this line. When you step, you move on to the next statement in your code. Usually, there's one statement per line. If that's not the case, you need multiple steps to move on to the next line.

## Think about the code

An important part of debugging is to stop and take some informed guesses about what you think portions of the code (both functions and blocks, such as loops) are trying to do. It's okay if you're not sure, that's part of the debugging process. But being actively engaged in the debugging process will help you locate bugs a lot more quickly.

Before we dig in further, let's remember that the Fibonacci sequence is a series of numbers that starts with the numbers 0 and 1, with every other following number being the sum of the two previous ones.

That means that:

```
text
Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(2) = 1 (0 + 1)
Fibonacci(3) = 2 (1 + 1)
```

Copy

```
Fibonacci(4) = 3 (1 + 2)
Fibonacci(5) = 5 (2 + 3)
```

Understanding that definition and looking at this `for` loop, we can deduce that:

1. The loop counts from 2 to `n` (the Fibonacci sequence number we're looking for).
2. If `n` is less than 2, the loop will never run. The `return` statement at the end of the function will return 0 if `n` is 0, and 1 if `n` is 1 or 2. These are the zero, first, and second values in the Fibonacci series, by definition.
3. The more interesting case is when `n` is greater than 2. In those cases, the current value is defined as the sum of the previous two values. So for this loop, `n1` and `n2` are the previous two values, and `sum` is the value for the current iteration. Because of that, each time we figure out the sum of the previous two values and set it to `sum`, we update our `n1` and `n2` values.

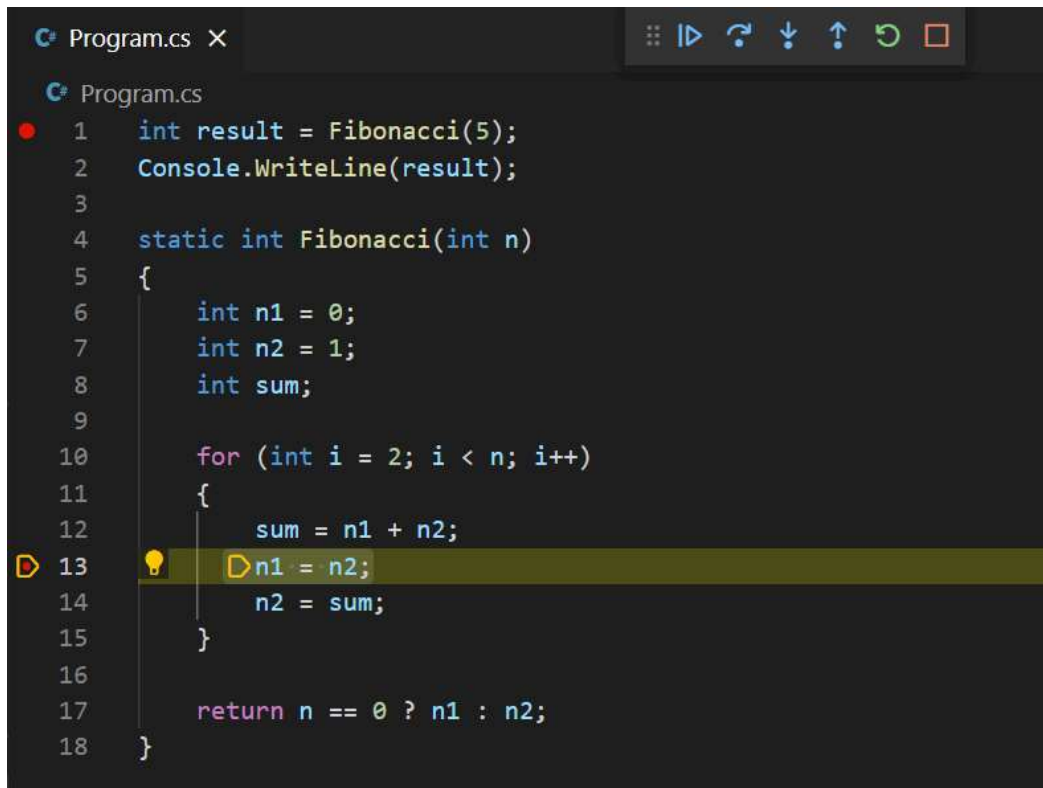
Okay, we don't need to overthink it past that. We can lean on our debugger a bit. But it's worth thinking about the code to see if it does what we expect and be more informed when it doesn't.

## Locate the bug with breakpoints

Stepping through your code can be helpful but tedious, especially when you're working with loops or other code that's called repeatedly. Rather than stepping through the loop over and over, we can set a new breakpoint on the first line of the loop.

When we're doing this, it's important to be strategic about where we put our breakpoints. We're especially interested in the value of `sum`, because it represents the current maximum Fibonacci value. Because of that, let's put our breakpoint on the line *after* `sum` is set.

1. Add a second breakpoint on line 13.



The screenshot shows a Visual Studio Code editor window with a file named `Program.cs`. The code is a C# program that calculates the 5th Fibonacci number. It defines a `Fibonacci` function that takes an integer `n` and returns the `n`th Fibonacci number. The function uses a `for` loop to calculate the sum of the previous two numbers, `n1` and `n2`, and updates them. A breakpoint is set on line 13, which is `n1 = n2;`. The breakpoint is represented by a yellow lightbulb icon on the left margin. The code is as follows:

```
1 int result = Fibonacci(5);
2 Console.WriteLine(result);
3
4 static int Fibonacci(int n)
5 {
6     int n1 = 0;
7     int n2 = 1;
8     int sum;
9
10    for (int i = 2; i < n; i++)
11    {
12        sum = n1 + n2;
13        n1 = n2;
14        n2 = sum;
15    }
16
17    return n == 0 ? n1 : n2;
18 }
```

### Note

If you notice that you keep running your code and then stepping a line or two, you can easily update your breakpoints to more efficient lines.

2. Now that we have a good breakpoint set in the loop, use the **Continue** debugger control to advance until the breakpoint is hit. Looking at our local variables, we see the following lines:

text

```
n [int]: 5
n1 [int]: 0
n2 [int]: 1
```

Copy

```
sum [int]: 1
i [int]: 2
```

These lines all seem correct. The first time through the loop, the `sum` of the previous two values is 1. Rather than stepping through line by line, we can take advantage of our breakpoints to jump to the next time through the loop.

3. Select **Continue** to continue program flow until the next breakpoint is hit, which will be on the next pass through the loop.

#### Note

Don't be too worried about skipping over the bug when you use **Continue**. You should expect that you'll often debug through the code several times to find the issue. It's often faster to run through it a few times as opposed to being too cautious when you step through.

This time, we see the following values:

text

Copy

```
n [int]: 5
n1 [int]: 1
n2 [int]: 1
sum [int]: 2
i [int]: 3
```

Let's think about it. Do these values still make sense? It seems like they do. For the third Fibonacci number, we're expecting to see our `sum` equal to 2, and it is.

4. Okay, let's select **Continue** to loop it again.

text

Copy

```
n [int]: 5
n1 [int]: 1
n2 [int]: 2
sum [int]: 3
i [int]: 4
```

Again, things are looking good. The fourth value in the series is expected to be 3.

5. At this point, you might start wondering if the code was correct all along and you imagined the bug! Let's keep with it for the last time through the loop. Select **Continue** one more time.

Wait a minute. The program finished running and printed out 3! That's not right.

Okay, not to worry. We haven't failed, we've learned. We now know that the code runs through the loop correctly until `i` equals 4, but then it exits out before computing the final value. I'm starting to get some ideas about where the bug is. Are you?

6. Let's set one more breakpoint on line 17, which reads:

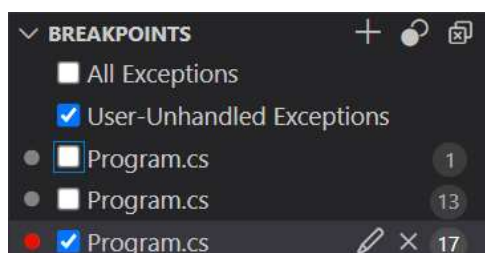
C#

Copy

```
return n == 0 ? n1 : n2;
```

This breakpoint will let us inspect the program state before the function exits. We've already learned all we can expect to from our previous breakpoints on lines 1 and 13, so we can clear them.

7. Remove the previous breakpoints on lines 1 and 13. You can do that by clicking them in the margin next to the line numbers, or by clearing the breakpoint check boxes for lines 1 and 13 in the **Breakpoints** pane in the lower left.



Now that we understand what's going on a lot better and have set a breakpoint designed to catch our program in the act of misbehaving, we should be able to catch this bug!

8. Start the debugger one last time.

```
text

n [int]: 5
n1 [int]: 2
n2 [int]: 3
sum [int]: 3
```

Copy

Well, that's not right. We specifically asked for Fibonacci(5), and we got Fibonacci(4). This function returns `n2`, and each loop iteration calculates the `sum` value and sets `n2` equal to `sum`.

Based on this information, and our previous debug run, we can see that the loop exited when `i` was 4, not 5.

Let's look at the first line of the `for` loop a little closer.

```
C#

for (int i = 2; i < n; i++)
```

Copy

Okay, wait a minute! That means that it will exit as soon as the top of the `for` loop sees that `i` is no longer less than `n`. That means that the loop code won't run for the case where `i` equals `n`. It seems like what we wanted was to run until `i <= n`, instead:

```
C#

for (int i = 2; i <= n; i++)
```

Copy

So with that change, your updated program should look like this example:

```
C#

int result = Fibonacci(5);
Console.WriteLine(result);

static int Fibonacci(int n)
{
    int n1 = 0;
    int n2 = 1;
    int sum;

    for (int i = 2; i <= n; i++)
    {
        sum = n1 + n2;
        n1 = n2;
        n2 = sum;
    }

    return n == 0 ? n1 : n2;
}
```

Copy

9. Stop the debugging session if you haven't already.

10. Next, make the preceding change to line 10 and leave our breakpoint on line 17.

11. Restart the debugger. This time, when we hit the breakpoint on line 17, we'll see the following values:

```
text

n [int]: 5
n1 [int]: 3
n2 [int]: 5
sum [int]: 5
```

Copy

Hey! It looks like we got it! Great job, you've saved the day for *Fibonacci, Inc.*!

12. Select **Continue** just to make sure the program returns the correct value.

```
text

5
```

Copy



The program '[105260] DotNetDebugging.dll' has exited with code 0 (0x0).

And that returns the correct output.

You did it! You've debugged some code you didn't write by using the .NET debugger in Visual Studio Code.

In the next unit, you'll learn how to make the code you write easier to debug by using the logging and tracing features that are built into .NET.