

Depurar aplicativos .NET de forma interativa com o depurador do Visual Studio Code.

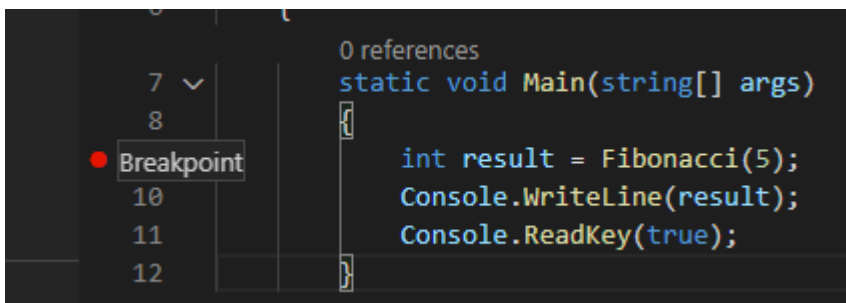
Um depurador é uma ferramenta de software usada para observar e controlar o fluxo de execução do seu programa com uma abordagem analítica. A meta para a qual ele foi criado é ajudar a encontrar a causa raiz de um bug e ajudar você a solucioná-lo. Ele funciona hospedando seu programa em um processo de execução próprio ou sendo executado como um processo separado que é anexado ao seu programa em execução, como o .NET.

Se você não está executando seu código por meio de um depurador, isso significa que você provavelmente está *adivinhand* o que está acontecendo em seu programa. O principal benefício de usar um depurador é que você pode *observar* seu programa em execução. Siga a execução do programa uma linha de código por vez. Assim, você evita a possibilidade de tentar adivinhar incorretamente.

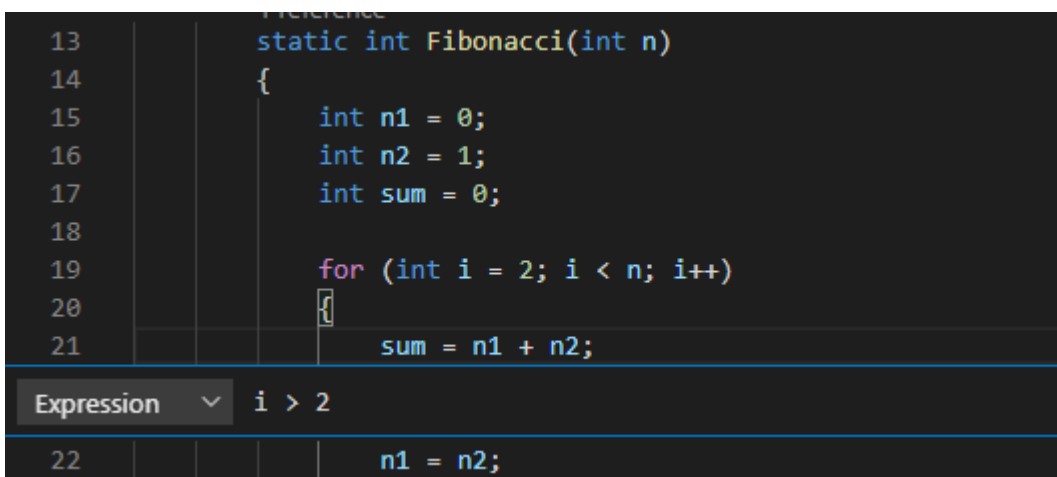
Pontos de interrupção.

Como o código é executado de modo rápido, você precisa ser capaz de pausar o programa em qualquer instrução. Você usará *pontos de interrupção* para fazer isso.

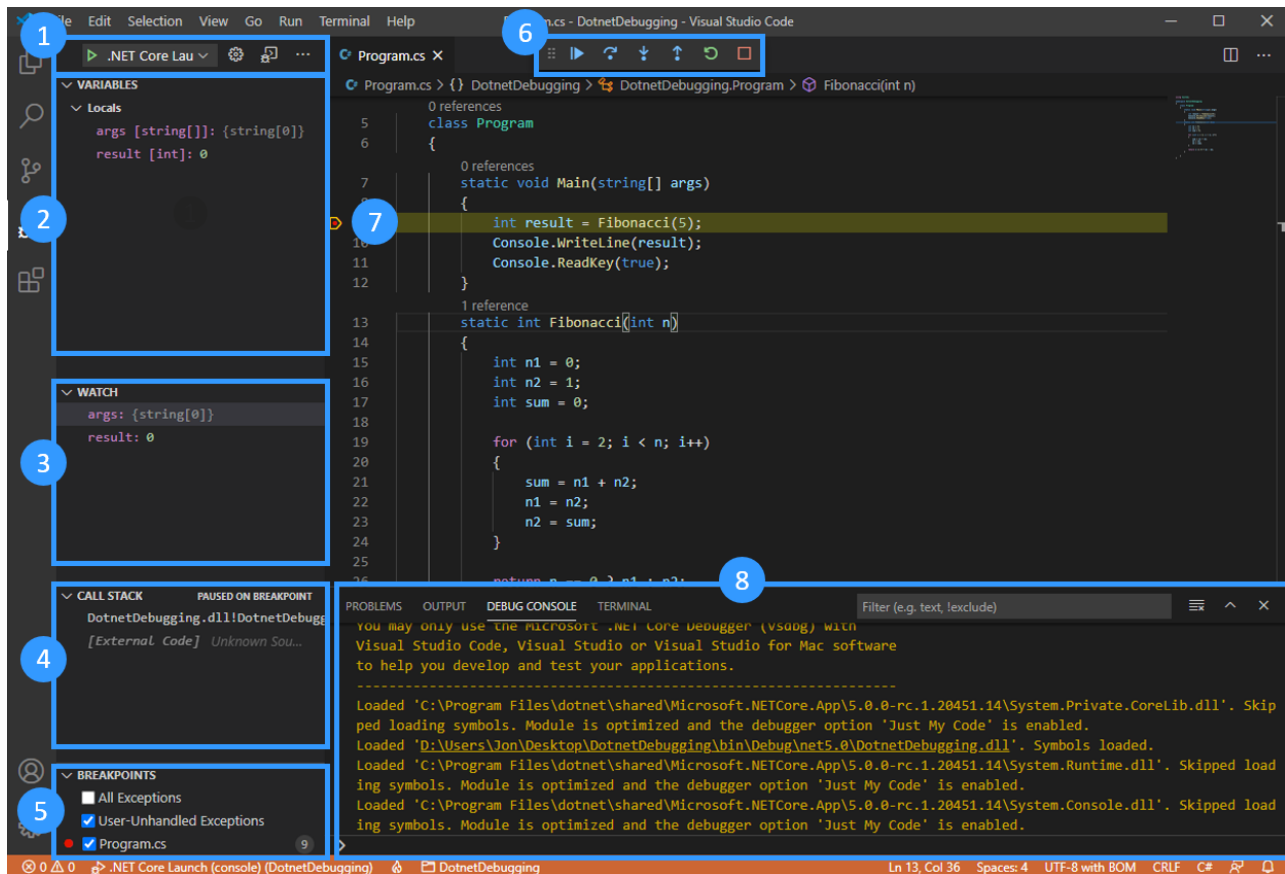
É possível adicionar um ponto de interrupção no Visual Studio Code clicando no lado esquerdo do número da linha que você deseja interromper. Você verá um círculo vermelho quando o ponto de interrupção estiver habilitado. Para removê-lo, basta clicar no círculo vermelho novamente.



Se clicar com o botão direito do mouse para adicionar um ponto de interrupção, você também poderá selecionar **Adicionar Ponto de Interrupção Condicional**. Esse tipo especial de ponto de interrupção permite que você insira uma *condição* para que a interrupção seja executada. Esse ponto de interrupção só estará ativo quando a condição especificada for atendida. Também é possível modificar um ponto de interrupção existente clicando com o botão direito do mouse nele e selecionando **Editar Ponto de Interrupção**.

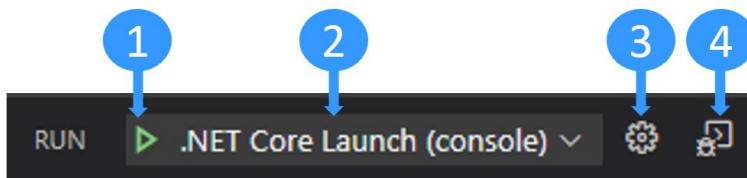


Visão geral do depurador do Visual Studio Code.



1. Controles de inicialização do depurador
2. Estado das variáveis
3. Estado das variáveis inspecionadas
4. Pilha de chamadas atual
5. Pontos de interrupção
6. Controles de execução
7. Etapa de execução atual
8. Console de depuração

1. Controles de inicialização do depurador.



1. Inicie a depuração.
2. Selecione a configuração de inicialização ativa.
3. Edite o arquivo **launch.json**. Crie um se precisar.
4. Abra o terminal de depuração.

2. Exibir e editar o estado das variáveis.

Ao analisar a causa de uma falha do programa, inspecione o estado das variáveis em busca de alterações inesperadas. Para fazer isso, você pode usar o painel **Variáveis**.

Suas variáveis são mostradas organizadas por escopo:

- As **variáveis locais** são acessíveis no escopo atual, geralmente na função atual.
- As **variáveis globais** são acessíveis de qualquer lugar do programa.
- As **variáveis de fechamento** são acessíveis do fechamento atual, se houver. Um fechamento combina o escopo local de uma função com o escopo da função externa à qual ela pertence.

É possível desdobrar escopos e variáveis selecionando a seta. Quando desdobra um objeto, você vê todas as propriedades definidas nele.

É possível alterar o valor de uma variável de maneira dinâmica clicando duas vezes nela.

Ao focalizar um parâmetro de função ou uma variável diretamente na janela do editor, é possível também espiar o respectivo valor.

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

3. Inspeccionar variáveis.

Quando você quer acompanhar o estado de uma variável no decorrer do tempo ou em funções diferentes, pode ser maçante precisar pesquisá-la toda vez. Nesse caso, o painel **Inspeção** é bem útil.

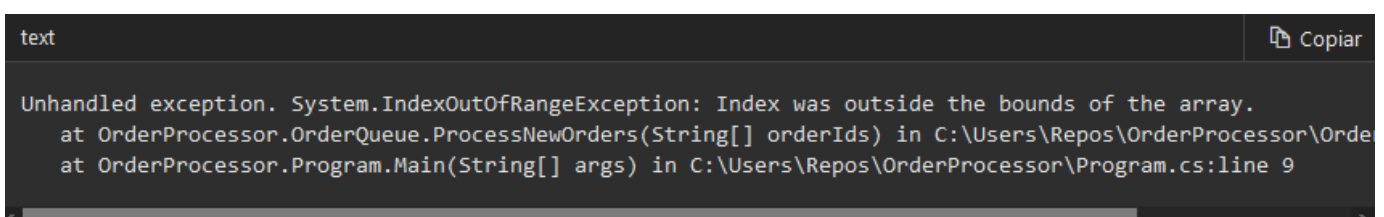
Clique no botão **mais** para inserir o nome de uma variável ou uma expressão a ser inspecionada. Como alternativa, você pode clicar com o botão direito do mouse em uma variável no painel **Variáveis** e selecionar **Adicionar para inspeção**.

Todas as expressões dentro do painel Monitoramento serão atualizadas automaticamente à medida que seu código for executado.

4. Pilha de chamadas.

Toda vez que o programa entra em uma função, uma entrada é adicionada à pilha de chamadas. Quando o aplicativo se torna complexo e você tem funções sendo chamadas dentro de outras funções repetidas vezes, a pilha de chamadas representa a trilha das chamadas de funções.

Ela é útil para localizar a origem de uma exceção. Se você enfrentar uma falha inesperada no programa, geralmente verá algo no console como o seguinte exemplo:



```
text Copiar  
Unhandled exception. System.IndexOutOfRangeException: Index was outside the bounds of the array.  
at OrderProcessor.OrderQueue.ProcessNewOrders(String[] orderIds) in C:\Users\Repos\OrderProcessor\Order  
at OrderProcessor.Program.Main(String[] args) in C:\Users\Repos\OrderProcessor\Program.cs:line 9
```

O grupo de linhas `at [...]` abaixo da mensagem de erro é chamado de *rastreamento de pilha*. O rastreamento de pilha informa o nome e a origem de cada função chamada antes da exceção. No entanto, ele pode ser um pouco difícil de decifrar, pois também inclui funções internas do runtime do .NET.

É aí que o painel **Pilha de chamadas** do Visual Studio Code é útil. Ele filtra informações indesejadas para mostrar a você apenas as funções relevantes do seu próprio código por padrão. Você pode então desenrolar essa pilha de chamadas para descobrir de onde a exceção foi originada.

5. Pontos de interrupção.

No painel **Pontos de interrupção**, você pode ver todos os pontos de interrupção que colocou no código e alternar entre eles. Você também pode alternar entre as opções para interromper em exceções capturadas ou não capturadas. Use o painel **Pontos de interrupção** para examinar o estado do programa e rastrear a origem de uma exceção, se uma ocorrer, usando a **Pilha de chamadas**.

6. Controlar a execução.

É possível controlar o fluxo de execução do programa usando esses controles.



Da esquerda para a direita, os controles são:

- **Continuar ou pausar a execução.** Se a execução for pausada, ela continuará até que o próximo ponto de interrupção seja atingido. Se o programa estiver em execução, o botão alternará para um botão pausar que você pode usar para pausar a execução.
- **Contornar.** Executa a próxima instrução de código no contexto atual.
- **Intervir.** Semelhante a **Contornar**, mas se a próxima instrução for uma chamada de função, siga para a primeira instrução de código dessa função (a mesma que o comando step).
- **Sair.** Se você estiver dentro de uma função, execute o código restante dela e volte para a instrução após a chamada de função inicial (o mesmo que o comando out).
- **Reiniciar.** Reinicie o programa desde o início.
- **Parar.** Encerre a execução e saia do depurador.

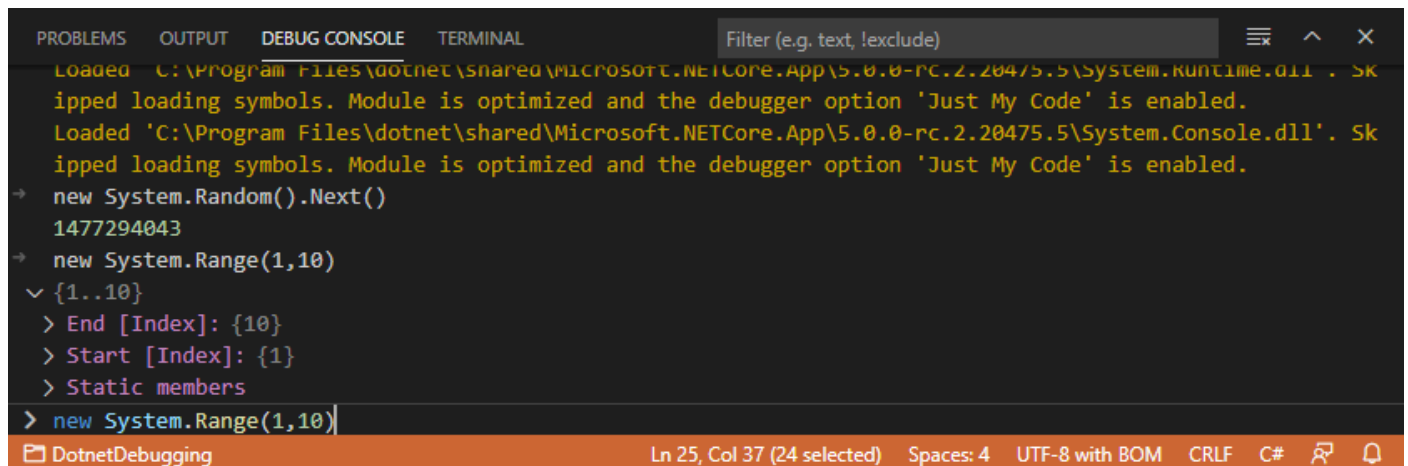
7. Etapa de execução atual

8. Usar o console de depuração.

O console de depuração pode ser exibido ou ocultado selecionando **Ctrl+Shift+Y**.

O console de depuração pode ser usado para visualizar os logs do console do aplicativo. Ele também pode ser usado para avaliar expressões ou executar código no conteúdo de execução atual, como comandos e nomes variáveis no depurador interno do .NET.

É possível inserir uma expressão .NET no campo de entrada na parte inferior do console de depuração. Em seguida, selecione **Enter** para avaliá-la. O resultado é exibido diretamente no console.



The screenshot shows the 'DEBUG CONSOLE' tab in Visual Studio Code. It displays the output of a .NET application, including loaded DLLs and the execution of C# code. The code being evaluated in the input field is `new System.Random().Next()` and `new System.Range(1,10)`. The output shows the result of `Next()` as `1477294043` and the properties of `Range(1,10)`, such as `End [Index]: {10}` and `Start [Index]: {1}`.

```
Loaded C:\Program Files\dotnet\shared\Microsoft.NETCore.App\5.0.0-rc.2.20475.5\System.Runtime.dll. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\5.0.0-rc.2.20475.5\System.Console.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
→ new System.Random().Next()
1477294043
→ new System.Range(1,10)
  {1..10}
  > End [Index]: {10}
  > Start [Index]: {1}
  > Static members
> new System.Range(1,10)
```

Usando o console de depuração, é possível verificar rapidamente um valor de variável, testar uma função com valores diferentes ou alterar o estado atual.


🕒 Observação

Embora o console de depuração seja muito útil para executar e avaliar o código .NET, ele pode ser um pouco confuso quando você está tentando executar ou depurar um aplicativo de console .NET. Isso acontece porque o console de depuração não aceita a entrada de terminal para um programa em execução.

Para lidar com a entrada de terminal durante a depuração, você pode usar o terminal integrado (uma das janelas do Visual Studio Code) ou um terminal externo. Neste tutorial, você usa o terminal integrado.

1. Abra `.vscode/launch.json`.
2. Altere a configuração de `console` para `integratedTerminal` de:


JSON

 Copiar

```
"console": "internalConsole",
```

Para:

JSON

 Copiar

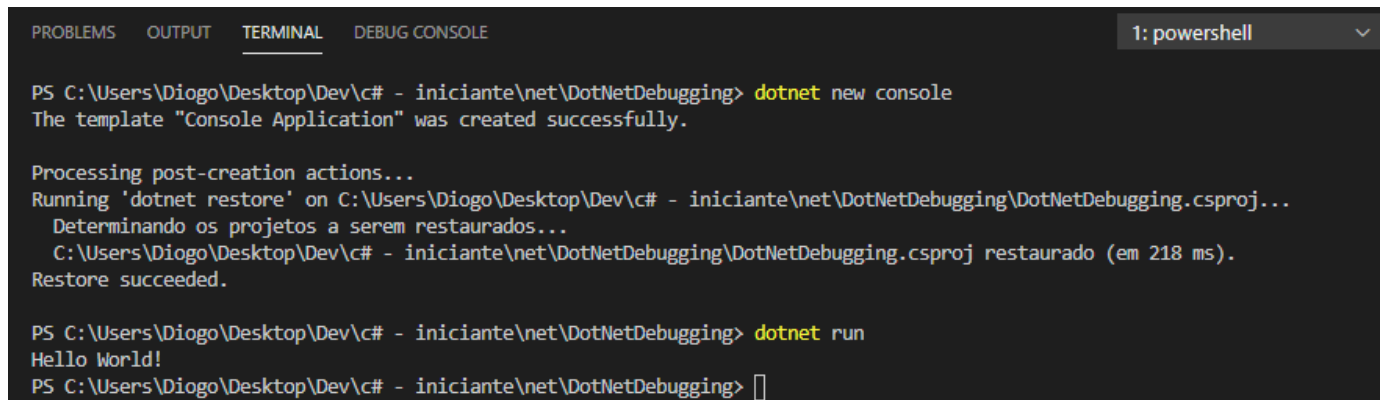
```
"console": "integratedTerminal",
```

3. Salve suas alterações.

Criar um projeto .NET de exemplo para depuração.

Vamos usar a depuração do .NET para corrigir um bug em uma calculadora Fibonacci.

1. Precisamos criar um novo projeto criando uma pasta chamada **DotNetDebugging** e abrindo no Visual Studio Code.
2. Agora vamos abrir o terminal e com o comando **dotnet new console** iniciamos um projeto básico em nossa pasta.
3. Para inicializar o projeto pela primeira vez vamos executar o comando **dotnet run**.



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 1: powershell
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDebugging> dotnet new console
The template "Console Application" was created successfully.

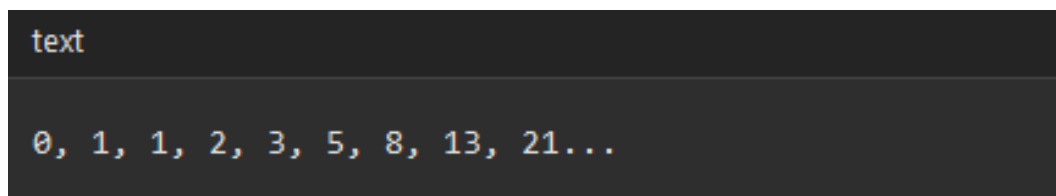
Processing post-creation actions...
Running 'dotnet restore' on C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDebugging\DotNetDebugging.csproj...
  Determinando os projetos a serem restaurados...
  C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDebugging\DotNetDebugging.csproj restaurado (em 218 ms).
Restore succeeded.

PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDebugging> dotnet run
Hello World!
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDebugging> 
```

Adicionar a lógica do programa Fibonacci.

Vamos usar um programa .NET curto para computar o *enésimo* número da sequência Fibonacci.

A sequência Fibonacci é um conjunto de números que começa com 0 e 1, em que cada número seguinte é a soma dos dois anteriores. A sequência continua conforme mostrado aqui:



```
text

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

1. Vamos substituir o conteúdo do **program.cs** pelo seguinte código.

Esse código contém um erro, que depuraremos posteriormente neste módulo. Não recomendamos o uso dele em nenhum aplicativo Fibonacci crítico até que o bug seja corrigido.

```
C#  
  
using System;  
  
namespace DotNetDebugging  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int result = Fibonacci(5);  
            Console.WriteLine(result);  
        }  
        static int Fibonacci(int n)  
        {  
            int n1 = 0;  
            int n2 = 1;  
            int sum = 0;  
  
            for (int i = 2; i < n; i++)  
            {  
                sum = n1 + n2;  
                n1 = n2;  
                n2 = sum;  
            }  
            return n == 0 ? n1 : n2;  
        }  
    }  
}
```

2. Iremos executar o código com **dotnet run**.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  
  
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDebugging> dotnet run  
3  
PS C:\Users\Diogo\Desktop\Dev\c# - iniciante\net\DotNetDebugging> |
```

Vemos que a saída do programa é **3**, mas neste caso, solicitamos que o programa calcule o quinto valor da sequência Fibonacci:

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

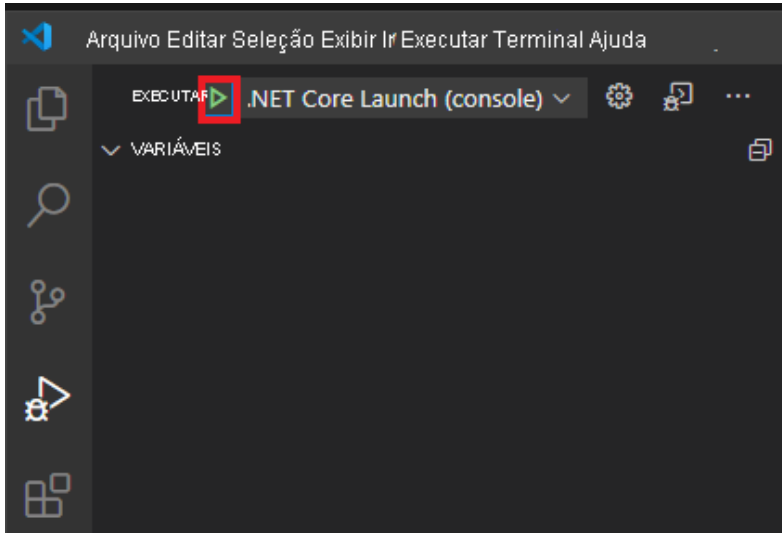
O quinto valor nessa lista é **5**, mas o programa retornou **3**. Vamos usar o depurador para diagnosticar e corrigir esse erro.

Iniciar o depurador.

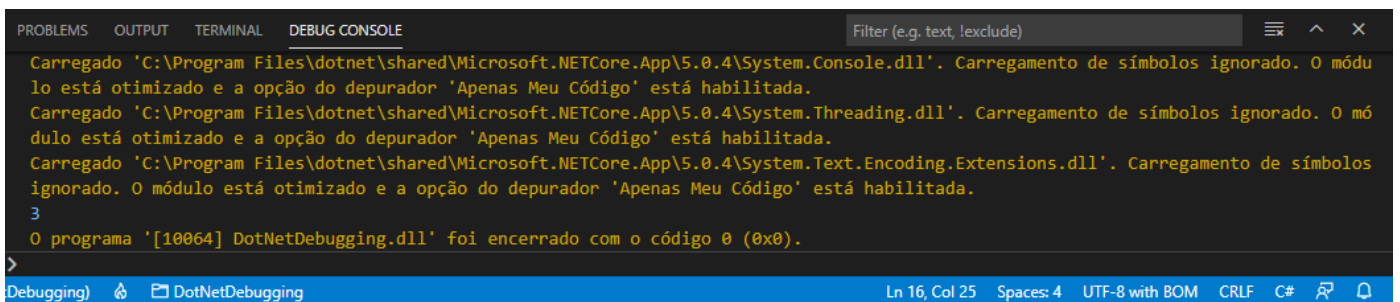
Para iniciar o depurador vamos usar o atalho **ctrl+shift+D** ou clicar no ícone do painel



Inicie o programa selecionando a guia **Executar** e o botão **Iniciar depuração**.

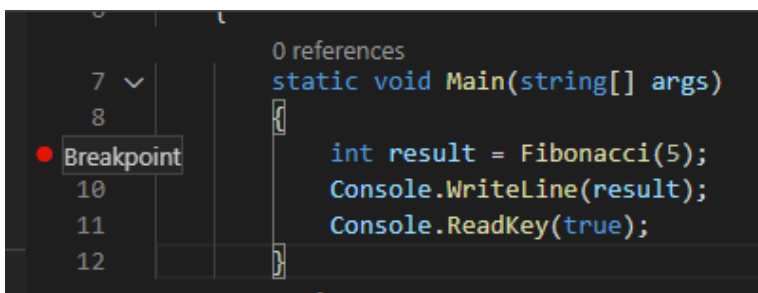


O programa será executado por completo pois não existe um ponto de interrupção. Após a execução o console de depuração deve aparecer.

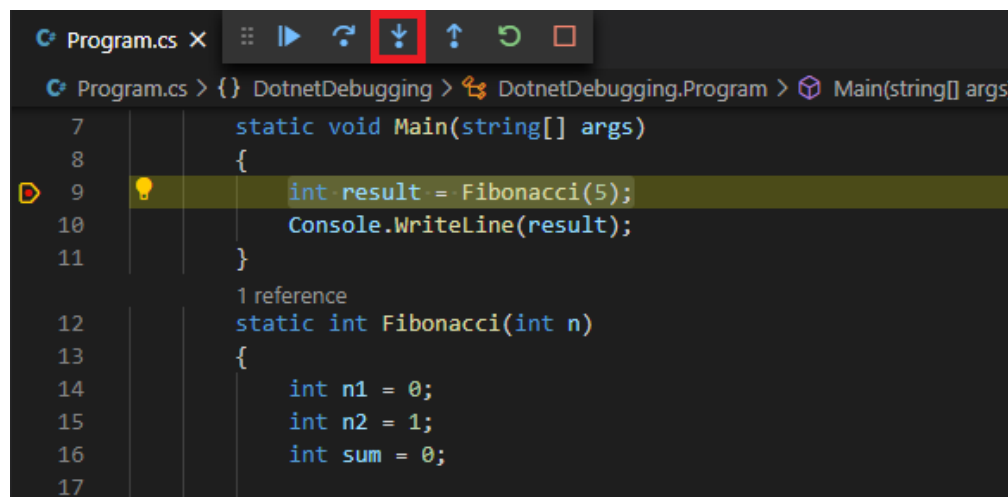


Usar pontos de interrupção e a execução passo a passo.

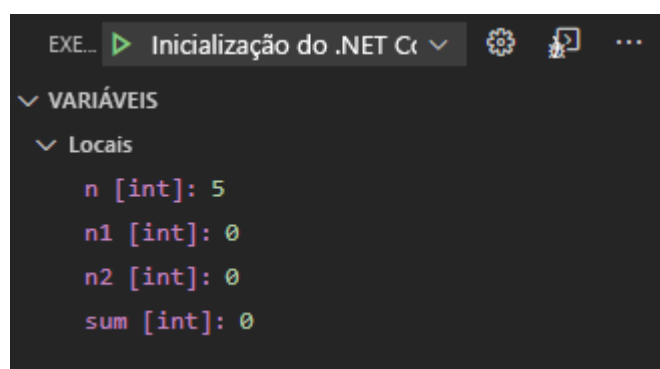
1. Adicione um ponto de interrupção clicando na margem esquerda na linha **9** em **int result = Fibonacci(5);**



2. Inicie a depuração novamente. O programa começa a ser executado. Ele é interrompido (a execução é pausada) na linha 9 devido ao ponto de interrupção que você definiu. Use os controles do depurador para entrar na função **Fibonacci()**.



3. Agora, dedique algum tempo para inspecionar os valores das diferentes variáveis usando o painel **Variáveis**.



- Qual é o valor mostrado para o parâmetro *n*?
- No início da execução da função, quais são os valores das variáveis locais **n1**, **n2** e **sum**?

4. Em seguida, vamos avançar para o loop `for` usando o controle do depurador **Depuração Parcial** seguindo o resultado das variáveis e analisando o código.



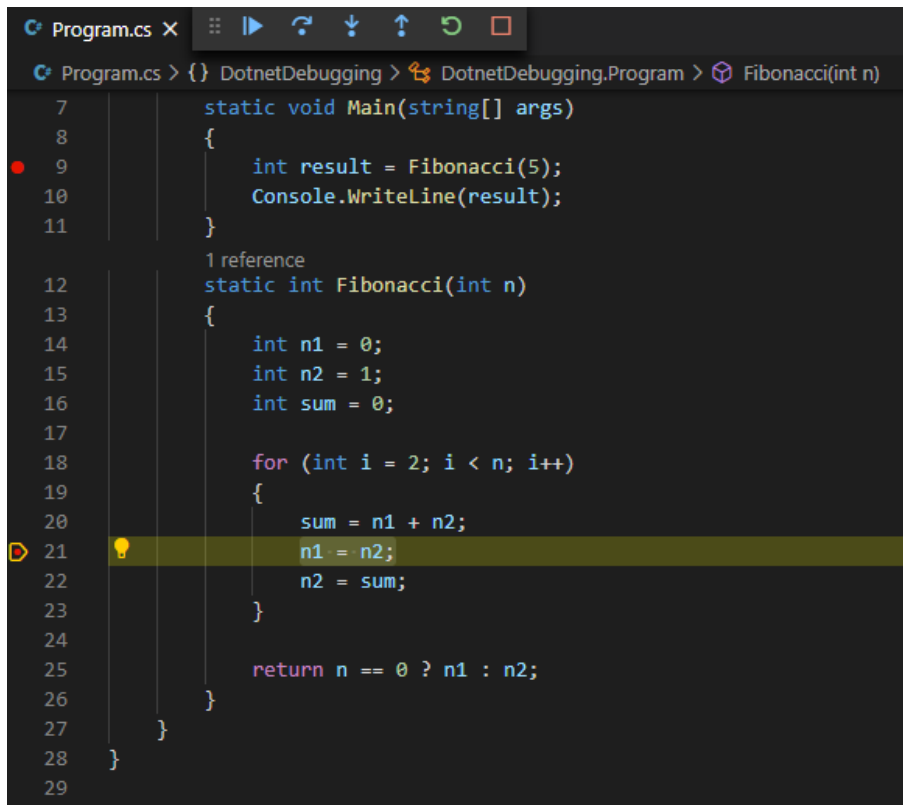
Após entender a definição e analisar o loop **for**, podemos deduzir que:

- O loop conta de **2** até **n** (o número da sequência Fibonacci que estamos buscando).
- Se **n** for menor que **2**, o loop nunca será executado. A instrução **return** no final da função retornará **0** se **n** for **0** e **1** se **n** for **1** ou **2**. Esses são o valor zero e o primeiro e segundo valores na série Fibonacci, por definição.
- O caso mais interessante é quando **n** é maior que **2**. Nesses casos, o valor atual é definido como a soma dos dois valores anteriores. Portanto, para esse loop, **n1** e **n2** são os dois valores anteriores, e **sum** é o valor para a iteração atual. Por isso, sempre que descobrimos a soma dos dois valores anteriores e a definimos como **sum**, atualizamos nossos valores **n1** e **n2**.

Localizar o bug com pontos de interrupção.

Durante esse processo, é importante inserir os pontos de interrupção de forma estratégica. Estamos interessados, sobretudo, no valor de **sum**, já que ele representa o valor máximo atual de Fibonacci. Por isso, colocaremos nosso ponto de interrupção na linha *após* a definição de **sum**.

5. Adicione um segundo ponto de interrupção na linha 21.

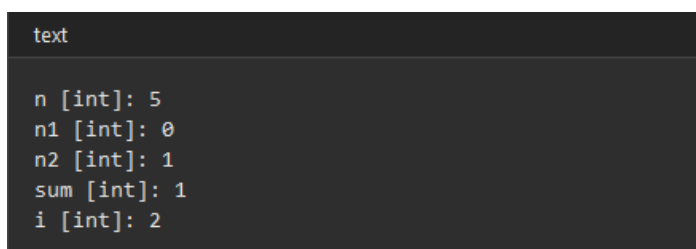


The screenshot shows the Visual Studio Code editor with a C# file named Program.cs. The code is as follows:

```
7 static void Main(string[] args)
8 {
9     int result = Fibonacci(5);
10    Console.WriteLine(result);
11 }
12
13 1 reference
14 static int Fibonacci(int n)
15 {
16     int n1 = 0;
17     int n2 = 1;
18     int sum = 0;
19
20     for (int i = 2; i < n; i++)
21     {
22         sum = n1 + n2;
23         n1 = n2;
24         n2 = sum;
25     }
26
27     return n == 0 ? n1 : n2;
28 }
29 }
```

A red dot indicates a breakpoint at line 9. A yellow lightbulb icon indicates a second breakpoint at line 21, which is highlighted with a green background.

6. Agora que temos um bom ponto de interrupção definido no loop, use o controle de depurador **Continuar** para avançar até que o ponto de interrupção seja atingido. Ao observarmos nossas variáveis locais, vemos as seguintes linhas:



The screenshot shows the 'text' window in Visual Studio Code displaying the values of local variables:

```
n [int]: 5
n1 [int]: 0
n2 [int]: 1
sum [int]: 1
i [int]: 2
```

Todas essas linhas parecem corretas. Na primeira vez que o loop é executado, a `sum` dos dois valores anteriores é 1. Em vez de passar linha por linha, podemos aproveitar nossos pontos de interrupção para ir direto até a próxima execução do loop.

7. Selecione **Continuar** para dar continuidade ao fluxo do programa até atingir o próximo ponto de interrupção, que estará na próxima passagem do loop.

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

Vamos pensar nisso. Esses valores ainda fazem sentido? Parece que sim. Para o terceiro número Fibonacci, esperamos ver nossa **sum** igual a 2, o que acontece.

8. Ok, vamos selecionar **Continuar** para fazer um loop novamente.

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

Mais uma vez, tudo parece bem. O quarto valor na série deve ser 3.

9. Neste momento, você pode se questionar se imaginou toda a situação do bug e se o código já estava certo desde o início. Vamos manter essa dúvida e executar o loop pela última vez. Selecione **Continuar** novamente.

Espere um pouco. O programa concluiu a execução e exibiu 3! Isso não está certo.

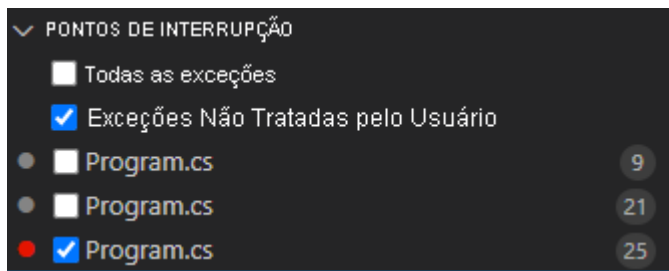
Não se preocupe. Nós não falhamos, nós aprendemos. Agora sabemos que o código executa o loop corretamente até que `i` seja igual a 4, mas ele é encerrado antes de calcular o valor final.

10. Vamos definir mais um ponto de interrupção na linha 25, em que está escrito:

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

Esse ponto de interrupção nos permitirá inspecionar o estado do programa antes do encerramento da função. Já vimos tudo o que podemos esperar com os pontos de interrupção anteriores nas linhas 9 e 21, então podemos limpá-los.

11. Remova os pontos de interrupção das linhas 9 e 21. Para isso, clique neles na margem ao lado dos números de linha ou desmarque os pontos de interrupção das linhas 9 e 21 no painel de pontos de interrupção no canto inferior esquerdo.



Agora que entendemos melhor o que está acontecendo e definimos um ponto de interrupção projetado para detectar o comportamento inadequado do programa, podemos identificar esse bug.

12. Inicie o depurador uma última vez.

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

Com base nessas informações e em nossa execução de depuração anterior, podemos ver que o loop foi encerrado quando **i** era **4**, e não **5**.

Vamos examinar a primeira linha do loop `for` um pouco melhor.

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

Espere um pouco! Isso significa que ele será encerrado assim que o topo do loop **for** observar **i** menor que **n**. Isso significa que o código de loop não será executado caso **i** seja igual a **n**. Parece que queríamos uma execução até **i <= n**.

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

13. Interrompa a sessão de depuração se ainda não tiver feito isso. Em seguida, faça a alteração anterior na linha 18 e deixe o ponto de interrupção na linha 25. Reinicie o depurador. Desta vez, quando atingirmos o ponto de interrupção na linha 25, veremos os seguintes valores:

text

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

Ei! Parece que deu certo! Ótimo trabalho.

14. Selecione **Continuar** apenas para garantir que o programa retorne o valor correto.

text

```
5  
The program '[105260] DotNetDebugging.dll' has exited with code 0 (0x0).
```