Exercise - Logging and tracing

4 minutes

100 XP

Now that the application has started development, it's good to add more diagnostics to the logic to help developers as they add new features. We can use our new knowledge of debug diagnostics to accomplish this task.

Write to the debug console

Before we debug the application, let's add more debug diagnostics. Additional diagnostics will help diagnose the application while it's being run under debug.

At the top of the `Program.cs` file, we'll add a new `using` statement to bring in `System.Diagnostics` so we can use the `Debug` methods.

C#

Copy

```csharp
using System.Diagnostics;
```

Add a `WriteLine` statement at the start of the `Fibonacci` method to get clarity when you debug through the code.

C#

Copy

```csharp
Debug.WriteLine($"Entering {nameof(Fibonacci)} method");
Debug.WriteLine($"We are looking for the {n}th number");
```

At the end of our `for` loop, we could print out every value. We could also use a conditional print statement by using `WriteIf` or `WriteLineIf` to add a print line only when `sum` is 1 at the end of the for loop:

C#

Copy

```csharp
for (int i = 2; i <= n; i++)
{
    sum = n1 + n2;
    n1 = n2;
    n2 = sum;
    Debug.WriteLineIf(sum == 1, $"sum is 1, n1 is {n1}, n2 is {n2}");
}
```

Debug the application and you should get the following output:

Output

Copy

```
Entering Fibonacci method
We are looking for the 5th number
sum is 1, n1 is 1, n2 is 1
```

Check for conditions with Assert

In some situations, you might want to stop the entire running application when a certain condition isn't met. By using `Debug.Assert`, you can check for a condition and output additional information about the state of the application. Let's add a check right before the return statement to ensure n2 is 5.

C#

Copy

```csharp
// If n2 is 5 continue, else break.
Debug.Assert(n2 == 5, "The return value is not 5 and it should be.");
return n == 0 ? n1 : n2;
```

Our application logic is already correct, so let's update our `Fibonacci(5);` to `Fibonacci(6);`, which will have a different result.

Debug the application. When `Debug.Assert` is run in the code, the debugger stops the application so you can inspect variables, watch window, call stack, and more. It also outputs the message to the debug console.

Output

Copy

```
---- DEBUG ASSERTION FAILED ----
---- Assert Short Message ----
The return value is not 5 and it should be.
```

```
---- Assert Long Message ----

   at Program.<<Main>$>g__Fibonacci|0_0(Int32 n) in C:\Users\Jon\Desktop\DotNetDebugging\Program.cs:line 23
   at Program.<Main>$(String[] args) in C:\Users\Jon\Desktop\DotNetDebugging\Program.cs:line 3
```

Stop debugging, and then run the application without debug by entering the following command in the terminal.

Bash

```bash
dotnet run
```

The application is terminated after the assertion has failed and information has been logged to the application output.

Output

```
Process terminated. Assertion failed.
The return value is not 5 and it should be.
   at Program.<<Main>$>g__Fibonacci|0_0(Int32 n) in C:\Users\Jon\Desktop\DotNetDebugging\Program.cs:line 23
   at Program.<Main>$(String[] args) in C:\Users\Jon\Desktop\DotNetDebugging\Program.cs:line 3
```

Now, let's run the application in `Release` configuration with the following command in the terminal.

Bash

```bash
dotnet run --configuration Release
```

The application successfully runs to completion because we're no longer in the `Debug` configuration.

Congratulations, you've successfully and efficiently debugged code by using features of .NET, which include `Debug.WriteLine` and `Debug.Assert`. Well done!