Tutorial: Explore ideas using top-level statements to build code as you learn

Article • 01/18/2025

In this tutorial, you learn how to:

- ✓ Learn the rules governing your use of top-level statements.
- ✓ Use top-level statements to explore algorithms.
- ✓ Refactor explorations into reusable components.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Start exploring

Top-level statements enable you to avoid the extra ceremony required by placing your program's entry point in a static method in a class. The typical starting point for a new console application looks like the following code:

```csharp
using System;

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

The preceding code is the result of running the `dotnet new console` command and creating a new console application. Those 11 lines contain only one line of executable code. You can simplify that program with the new top-level statements feature. That enables you to remove all but two of the lines in this program:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Important

The C# templates for .NET 6 use *top level statements*. Your application may not match the code in this article, if you've already upgraded to the .NET 6. For more information see the article on New C# templates generate top level statements

The .NET 6 SDK also adds a set of *implicit* `global using` directives for projects that use the following SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

These implicit `global using` directives include the most common namespaces for the project type.

For more information, see the article on Implicit using directives

This feature simplifies your exploration of new ideas. You can use top-level statements for scripting scenarios, or to explore. Once you've got the basics working, you can start refactoring the code and create methods, classes, or other assemblies for reusable components you built. Top-level statements do enable quick experimentation and beginner tutorials. They also provide a smooth path from experimentation to full programs.

Top-level statements are executed in the order they appear in the file. Top-level statements can only be used in one source file in your application. The compiler generates an error if you use them in more than one file.

Build a magic .NET answer machine

For this tutorial, let's build a console application that answers a "yes" or "no" question with a random answer. You build out the functionality step by step. You can focus on your task rather than ceremony needed for the structure of a typical program. Then, once you're happy with the functionality, you can refactor the application as you see fit.

A good starting point is to write the question back to the console. You can start by writing the following code:

C#

```
Console.WriteLine(args);
```

You don't declare an `args` variable. For the single source file that contains your top-level statements, the compiler recognizes `args` to mean the command-line arguments. The type of args is a `string[]`, as in all C# programs.

You can test your code by running the following `dotnet run` command:

.NET CLI

```
dotnet run -- Should I use top level statements in all my programs?
```

The arguments after the `--` on the command line are passed to the program. You can see the type of the `args` variable printed to the console:

Console

```
System.String[]
```

To write the question to the console, you need to enumerate the arguments and separate them with a space. Replace the `WriteLine` call with the following code:

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

Now, when you run the program, it correctly displays the question as a string of arguments.

Respond with a random answer

After echoing the question, you can add the code to generate the random answer. Start by adding an array of possible answers:

C#

```
string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don’t count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];
```

This array has 10 answers that are affirmative, five that are noncommittal, and five that are negative. Next, add the following code to generate and display a random answer from the array:

C#

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

You can run the application again to see the results. You should see something like the following output:

.NET CLI

```
dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.
```

The code to generate an answer includes a variable declaration in your top level statements. The compiler includes that declaration in the compiler generated `Main` method. Because these variable declarations are local variables, you can't include the `static` modifier.

This code answers the questions, but let's add one more feature. You'd like your question app to simulate thinking about the answer. You can do that by adding a bit of ASCII animation, and pausing while working. Add the following code after the line that echoes the question:

C#

```csharp
for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();
```

You also need to add a `using` directive to the top of the source file:

C#

```csharp
using System.Threading.Tasks;
```

The `using` directives must be before any other statements in the file. Otherwise, it's a compiler error. You can run the program again and see the animation. That makes a better experience. Experiment with the length of the delay to match your taste.

The preceding code creates a set of spinning lines separated by a space. Adding the `await` keyword instructs the compiler to generate the program entry point as a method that has the `async` modifier, and returns a System.Threading.Tasks.Task. This program doesn't return a value, so the program entry point returns a `Task`. If your program returns an integer value, you would add a return statement to the end of your top-level statements. That return statement would specify the integer value to return. If your top-level statements include an `await` expression, the return type becomes System.Threading.Tasks.Task<TResult>.

Refactoring for the future

Your program should look like the following code:

```
C#

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
    "It is decidedly so.",  "Ask again later.",            "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",    "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",         "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

The preceding code is reasonable. It works. But it isn't reusable. Now that you have the application working, it's time to pull out reusable parts.

One candidate is the code that displays the waiting animation. That snippet can become a method:

You can start by creating a local function in your file. Replace the current animation with the following code:

C#

```csharp
await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
```

The preceding code creates a local function inside your main method. That code still isn't reusable. So, extract that code into a class. Create a new file named *utilities.cs* and add the following code:

C#

```csharp
namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
```

```
                    Console.Write("/ \\");
                    await Task.Delay(50);
                    Console.Write("\b\b\b");
                    Console.Write("- |");
                    await Task.Delay(50);
                    Console.Write("\b\b\b");
                    Console.Write("\\ /");
                    await Task.Delay(50);
                    Console.Write("\b\b\b");
                }
                Console.WriteLine();
            }
        }
    }
```

A file that has top-level statements can also contain namespaces and types at the end of the file, after the top-level statements. But for this tutorial you put the animation method in a separate file to make it more readily reusable.

Finally, you can clean the animation code to remove some duplication, by using `foreach` loop to iterate through set of animations elements defined in `animations` array. The full `ShowConsoleAnimation` method after refactor should look similar to the following code:

```C#
public static async Task ShowConsoleAnimation()
{
    string[] animations = ["| -", "/ \\", "- |", "\\ /"];
    for (int i = 0; i < 20; i++)
    {
        foreach (string s in animations)
        {
            Console.Write(s);
            await Task.Delay(50);
            Console.Write("\b\b\b");
        }
    }
    Console.WriteLine();
}
```

Now you have a complete application, and you refactored the reusable parts for later use. You can call the new utility method from your top-level statements, as shown in the finished version of the main program:

```C#
using MyNamespace;
```

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don’t count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

The preceding example adds the call to `Utilities.ShowConsoleAnimation`, and adds another `using` directive.

Summary

Top-level statements make it easier to create simple programs for use to explore new algorithms. You can experiment with algorithms by trying different snippets of code. Once you learned what works, you can refactor the code to be more maintainable.

Top-level statements simplify programs that are based on console apps. These apps include Azure functions, GitHub actions, and other small utilities. For more information, see Top-level statements (C# Programming Guide).