Contents

Main() and command-line arguments (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The `Main` method is the entry point of a C# application. (Libraries and services do not require a `Main` method as an entry point.) When the application is started, the `Main` method is the first method that is invoked.

There can only be one entry point in a C# program. If you have more than one class that has a `Main` method, you must compile your program with the **/main** compiler option to specify which `Main` method to use as the entry point. For more information, see /main (C# Compiler Options).

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments:
        System.Console.WriteLine(args.Length);
    }
}
```

Overview

- The `Main` method is the entry point of an executable program; it is where the program control starts and ends.
- `Main` is declared inside a class or struct. `Main` must be static and it need not be public. (In the earlier example, it receives the default access of private.) The enclosing class or struct is not required to be static.
- `Main` can either have a `void`, `int`, or, starting with C# 7.1, `Task`, or `Task<int>` return type.
- If and only if `Main` returns a `Task` or `Task<int>`, the declaration of `Main` may include the `async` modifier. Note that this specifically excludes an `async void Main` method.
- The `Main` method can be declared with or without a `string[]` parameter that contains command-line arguments. When using Visual Studio to create Windows applications, you can add the parameter manually or else use the Environment class to obtain the command-line arguments. Parameters are read as zero-indexed command-line arguments. Unlike C and C++, the name of the program is not treated as the first command-line argument.

The addition of `async` and `Task`, `Task<int>` return types simplifies program code when console applications need to start and `await` asynchronous operations in `Main`.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Command-line Building With csc.exe
- C# Programming Guide
- Methods
- Inside a C# Program

Command-Line Arguments (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

You can send arguments to the `Main` method by defining the method in one of the following ways:

```
static int Main(string[] args)
```

```
static void Main(string[] args)
```

> **NOTE**
> To enable command-line arguments in the `Main` method in a Windows Forms application, you must manually modify the signature of `Main` in program.cs. The code generated by the Windows Forms designer creates a `Main` without an input parameter. You can also use Environment.CommandLine or Environment.GetCommandLineArgs to access the command-line arguments from any point in a console or Windows application.

The parameter of the `Main` method is a String array that represents the command-line arguments. Usually you determine whether arguments exist by testing the `Length` property, for example:

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

You can also convert the string arguments to numeric types by using the Convert class or the `Parse` method. For example, the following statement converts the `string` to a `long` number by using the Parse method:

```
long num = Int64.Parse(args[0]);
```

It is also possible to use the C# type `long`, which aliases `Int64`:

```
long num = long.Parse(args[0]);
```

You can also use the `Convert` class method `ToInt64` to do the same thing:

```
long num = Convert.ToInt64(s);
```

For more information, see Parse and Convert.

Example

The following example shows how to use command-line arguments in a console application. The application takes one argument at run time, converts the argument to an integer, and calculates the factorial of the number. If no arguments are supplied, the application issues a message that explains the correct usage of the program.

To compile and run the application from a command prompt, follow these steps:

1. Paste the following code into any text editor, and then save the file as a text file with the name `Factorial.cs`.

```
//Add a using directive for System if the directive isn't already present.

public class Functions
{
    public static long Factorial(int n)
    {
        // Test for invalid input
        if ((n < 0) || (n > 20))
        {
            return -1;
        }

        // Calculate the factorial iteratively rather than recursively:
        long tempResult = 1;
        for (int i = 1; i <= n; i++)
        {
            tempResult *= i;
        }
        return tempResult;
    }
}

class MainClass
{
    static int Main(string[] args)
    {
        // Test if input arguments were supplied:
        if (args.Length == 0)
        {
            System.Console.WriteLine("Please enter a numeric argument.");
            System.Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Try to convert the input arguments to numbers. This will throw
        // an exception if the argument is not a number.
        // num = int.Parse(args[0]);
        int num;
        bool test = int.TryParse(args[0], out num);
        if (test == false)
        {
            System.Console.WriteLine("Please enter a numeric argument.");
            System.Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Calculate factorial.
        long result = Functions.Factorial(num);

        // Print result.
        if (result == -1)
            System.Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            System.Console.WriteLine("The Factorial of {0} is {1}.", num, result);

        return 0;
    }
}
// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.
```

2. From the **Start** screen or **Start** menu, open a Visual Studio **Developer Command Prompt** window, and

then navigate to the folder that contains the file that you just created.

3. Enter the following command to compile the application.

   ```
   csc Factorial.cs
   ```

   If your application has no compilation errors, an executable file that's named `Factorial.exe` is created.

4. Enter the following command to calculate the factorial of 3:

   ```
   Factorial 3
   ```

5. The command produces this output: `The factorial of 3 is 6.`

> **NOTE**
>
> When running an application in Visual Studio, you can specify command-line arguments in the Debug Page, Project Designer.

For more examples about how to use command-line arguments, see How to: Create and Use Assemblies Using the Command Line.

See also

- System.Environment
- C# Programming Guide
- Main() and Command-Line Arguments
- How to: Display Command Line Arguments
- How to: Access Command-Line Arguments Using foreach
- Main() Return Values
- Classes

How to: Display Command Line Arguments (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Arguments provided to an executable on the command-line are accessible through an optional parameter to `Main`. The arguments are provided in the form of an array of strings. Each element of the array contains one argument. White-space between arguments is removed. For example, consider these command-line invocations of a fictitious executable:

INPUT ON COMMAND-LINE	ARRAY OF STRINGS PASSED TO MAIN
executable.exe a b c	"a" "b" "c"
executable.exe one two	"one" "two"
executable.exe "one two" three	"one two" "three"

NOTE

When you are running an application in Visual Studio, you can specify command-line arguments in the Debug Page, Project Designer.

Example

This example displays the command line arguments passed to a command-line application. The output shown is for the first entry in the table above.

```
class CommandLine
{
    static void Main(string[] args)
    {
        // The Length property provides the number of array elements
        System.Console.WriteLine("parameter count = {0}", args.Length);

        for (int i = 0; i < args.Length; i++)
        {
            System.Console.WriteLine("Arg[{0}] = [{1}]", i, args[i]);
        }
    }
}
/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

See also

- C# Programming Guide
- Command-line Building With csc.exe
- Main() and Command-Line Arguments
- How to: Access Command-Line Arguments Using foreach
- Main() Return Values

How to: Access Command-Line Arguments Using foreach (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Another approach to iterating over the array is to use the foreach statement as shown in this example. The `foreach` statement can be used to iterate over an array, a .NET Framework collection class, or any class or struct that implements the IEnumerable interface.

> **NOTE**
> When running an application in Visual Studio, you can specify command-line arguments in the Debug Page, Project Designer.

Example

This example demonstrates how to print out the command line arguments using `foreach`.

```
// arguments: John Paul Mary
```

```
class CommandLine2
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Number of command line parameters = {0}", args.Length);

        foreach (string s in args)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Number of command line parameters = 3
    John
    Paul
    Mary
*/
```

See also

- Array
- System.Collections
- Command-line Building With csc.exe
- C# Programming Guide
- foreach, in
- Main() and Command-Line Arguments
- How to: Display Command Line Arguments
- Main() Return Values

Main() return values (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The `Main` method can return `void`:

```
static void Main()
{
    //...
}
```

It can also return an `int`:

```
static int Main()
{
    //...
    return 0;
}
```

If the return value from `Main` is not used, returning `void` allows for slightly simpler code. However, returning an integer enables the program to communicate status information to other programs or scripts that invoke the executable file. The return value from `Main` is treated as the exit code for the process. The following example shows how the return value from `Main` can be accessed.

Example

This example uses .NET Core command line tools. If you are unfamiliar with .NET Core command line tools, you can learn about them in this Get started topic.

Modify the `Main` method in *program.cs* as follows:

```
// Save this program as MainReturnValTest.cs.
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

When a program is executed in Windows, any value returned from the `Main` function is stored in an environment variable. This environment variable can be retrieved using `ERRORLEVEL` from a batch file, or `$LastExitCode` from powershell.

You can build the application using the dotnet CLI `dotnet build` command.

Next, create a Powershell script to run the application and display the result. Paste the following code into a text file and save it as `test.ps1` in the folder that contains the project. Run the powershell script by typing `test.ps1` at the powershell prompt.

Because the code returns zero, the batch file will report success. However, if you change MainReturnValTest.cs to return a non-zero value and then re-compile the program, subsequent execution of the powershell script will

report failure.

```
dotnet run
if ($LastExitCode -eq 0) {
    Write-Host "Execution succeeded"
} else
{
    Write-Host "Execution Failed"
}
Write-Host "Return value = " $LastExitCode
```

Sample output

```
Execution succeeded
Return value = 0
```

Async Main return values

Async Main return values move the boilerplate code necessary for calling asynchronous methods in `Main` to code generated by the compiler. Previously, you would need to write this construct to call asynchronous code and ensure your program ran until the asynchronous operation completed:

```
public static void Main()
{
    AsyncConsoleWork().GetAwaiter().GetResult();
}

private static async Task<int> AsyncConsoleWork()
{
    // Main body here
    return 0;
}
```

Now, this can be replaced by:

```
static async Task<int> Main(string[] args)
{
    return await AsyncConsoleWork();
}
```

The advantage of the new syntax is that the compiler always generates the correct code.

Compiler generated code

When the application entry point returns a `Task` or `Task<int>`, the compiler generates a new entry point that calls the entry point method declared in the application code. Assuming that this entry point is called `$GeneratedMain`, the compiler generates the following code for these entry points:

- `static Task Main()` results in the compiler emitting the equivalent of `private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task Main(string[])` results in the compiler emitting the equivalent of `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` results in the compiler emitting the equivalent of `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`

- `static Task<int> Main(string[])` results in the compiler emitting the equivalent of `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

> **NOTE**
>
> If the examples used `async` modifier on the `Main` method, the compiler would generate the same code.

See also

- C# Programming Guide
- C# Reference
- Main() and Command-Line Arguments
- How to: Display Command Line Arguments
- How to: Access Command-Line Arguments Using foreach