Contents

Inside a C# program

1/11/2019 • 2 minutes to read • Edit Online

The section discusses the general structure of a C# program, and includes the standard "Hello, World!" example.

In this section

- Hello World -- Your First Program
- General Structure of a C# Program
- Identifier names
- C# Coding Conventions

Related sections

- Getting Started with C#
- C# Programming Guide
- C# Reference
- Samples and tutorials

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

- C# Programming Guide

Hello World -- Your first program (C# Programming Guide)

1/11/2019 • 4 minutes to read • Edit Online

The following procedure creates a C# version of the traditional "Hello World!" program. The program displays the string `Hello World!`

For more examples of introductory concepts, see Getting Started with Visual C# and Visual Basic.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To create and run a console application

1. Start Visual Studio.
2. On the menu bar, choose **File**, **New**, **Project**.

   The **New Project** dialog box opens.
3. Expand **Installed**, expand **Templates**, expand **Visual C#**, and then choose **Console Application**.
4. In the **Name** box, specify a name for your project, and then choose the **OK** button.

   The new project appears in **Solution Explorer**.
5. If Program.cs isn't open in the **Code Editor**, open the shortcut menu for **Program.cs** in **Solution Explorer**, and then choose **View Code**.
6. Replace the contents of Program.cs with the following code.

```
// A Hello World! program in C#.
using System;
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
```

7. Choose the F5 key to run the project. A Command Prompt window appears that contains the line `Hello World!`

Next, the important parts of this program are examined.

Comments

The first line contains a comment. The characters `//` convert the rest of the line to a comment.

```
// A Hello World! program in C#.
```

You can also comment out a block of text by enclosing it between the `/*` and `*/` characters. This is shown in the following example.

```
/* A "Hello World!" program in C#.
This program displays the string "Hello World!" on the screen. */
```

Main method

A C# console application must contain a `Main` method, in which control starts and ends. The `Main` method is where you create objects and execute other methods.

The `Main` method is a static method that resides inside a class or a struct. In the previous "Hello World!" example, it resides in a class named `Hello`. You can declare the `Main` method in one of the following ways:

- It can return `void`.

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

- It can also return an integer.

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

- With either of the return types, it can take arguments.

  ```
  static void Main(string[] args)
  {
      //...
  }
  ```

  -or-

  ```
  static int Main(string[] args)
  {
      //...
      return 0;
  }
  ```

The parameter of the `Main` method, `args`, is a `string` array that contains the command-line arguments used to

invoke the program. Unlike in C++, the array does not include the name of the executable (exe) file.

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

The call to ReadKey at the end of the `Main` method prevents the console window from closing before you have a chance to read the output when you run your program in debug mode, by pressing F5.

Input and output

C# programs generally use the input/output services provided by the run-time library of the .NET Framework. The statement `System.Console.WriteLine("Hello World!");` uses the WriteLine method. This is one of the output methods of the Console class in the run-time library. It displays its string parameter on the standard output stream followed by a new line. Other Console methods are available for different input and output operations. If you include the `using System;` directive at the beginning of the program, you can directly use the System classes and methods without fully qualifying them. For example, you can call `Console.WriteLine` instead of `System.Console.WriteLine`:

```
using System;
```

```
Console.WriteLine("Hello World!");
```

For more information about input/output methods, see System.IO.

Command-line compilation and execution

You can compile the "Hello World!" program by using the command line instead of the Visual Studio Integrated Development Environment (IDE).

To compile and run from a command prompt

1. Paste the code from the preceding procedure into any text editor, and then save the file as a text file. Name the file `Hello.cs`. C# source code files use the extension `.cs`.
2. Perform one of the following steps to open a command-prompt window:
   - In Windows 10, on the **Start** menu, search for `Developer Command Prompt`, and then tap or choose **Developer Command Prompt for VS 2017**.

     A Developer Command Prompt window appears.
   - In Windows 7, open the **Start** menu, expand the folder for the current version of Visual Studio, open the shortcut menu for **Visual Studio Tools**, and then choose **Developer Command Prompt for VS 2017**.

     A Developer Command Prompt window appears.
   - Enable command-line builds from a standard Command Prompt window.

     See How to: Set Environment Variables for the Visual Studio Command Line.
3. In the command-prompt window, navigate to the folder that contains your `Hello.cs` file.
4. Enter the following command to compile `Hello.cs`.

   ```
   csc Hello.cs
   ```

   If your program has no compilation errors, an executable file that is named `Hello.exe` is created.

5. In the command-prompt window, enter the following command to run the program:

```
Hello
```

For more information about the C# compiler and its options, see C# Compiler Options.

See also

- C# Programming Guide
- Inside a C# Program
- Strings
- Samples and tutorials
- C# Reference
- Main() and Command-Line Arguments
- Getting Started with Visual C# and Visual Basic

General Structure of a C# Program (C# Programming Guide)

2/3/2019 • 2 minutes to read • Edit Online

C# programs can consist of one or more files. Each file can contain zero or more namespaces. A namespace can contain types such as classes, structs, interfaces, enumerations, and delegates, in addition to other namespaces. The following is the skeleton of a C# program that contains all of these elements.

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

Related Sections

For more information:

- Classes
- Structs
- Namespaces
- Interfaces

- Delegates

C# Language Specification

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

See also

- C# Programming Guide
- Inside a C# Program
- C# Reference

Identifier names

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

An **identifier** is the name you assign to a type (class, interface, struct, delegate, or enum), member, variable, or namespace. Valid identifiers must follow these rules:

- Identifiers must start with a letter, or `_`.
- Identifiers may contain Unicode letter characters, decimal digit characters, Unicode connecting characters, Unicode combining characters, or Unicode formatting characters. For more information on Unicode categories, see the Unicode Category Database. You can declare identifiers that match C# keywords by using the `@` prefix on the identifier. The `@` is not part of the identifier name. For example, `@if` declares an identifier named `if`. These verbatim identifiers are primarily for interoperability with identifiers declared in other languages.

For a complete definition of valid identifiers, see the Identifiers topic in the C# Language Specification.

Naming conventions

In addition to the rules, there are a number of identifier naming conventions used throughout the .NET APIs. By convention, C# programs use `PascalCase` for type names, namespaces, and all public members. In addition, the following conventions are common:

- Interface names start with a capital `I`.
- Attribute types end with the word `Attribute`.
- Enum types use a singular noun for non-flags, and a plural noun for flags.
- Identifiers should not contain two consecutive `_` characters. Those names are reserved for compiler generated identifiers.

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

- C# Programming Guide
- Inside a C# Program
- C# Reference
- Classes
- Structs
- Namespaces
- Interfaces
- Delegates

C# Coding Conventions (C# Programming Guide)

1/23/2019 • 8 minutes to read • Edit Online

Coding conventions serve the following purposes:

- They create a consistent look to the code, so that readers can focus on content, not layout.
- They enable readers to understand the code more quickly by making assumptions based on previous experience.
- They facilitate copying, changing, and maintaining the code.
- They demonstrate C# best practices.

The guidelines in this topic are used by Microsoft to develop samples and documentation.

Naming Conventions

- In short examples that do not include using directives, use namespace qualifications. If you know that a namespace is imported by default in a project, you do not have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they are too long for a single line, as shown in the following example.

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- You do not have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.

Layout Conventions

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

- Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see Options, Text Editor, C#, Formatting.
- Write only one statement per line.
- Write only one declaration per line.
- If continuation lines are not indented automatically, indent them one tab stop (four spaces).
- Add at least one blank line between method definitions and property definitions.
- Use parentheses to make clauses in an expression apparent, as shown in the following code.

```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

Commenting Conventions

- Place the comment on a separate line, not at the end of a line of code.
- Begin comment text with an uppercase letter.
- End comment text with a period.
- Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.

```
// The following declaration creates a query. It does not run
// the query.
```

- Do not create formatted blocks of asterisks around comments.

Language Guidelines

The following sections describe practices that the C# team follows to prepare code examples and samples.

String Data Type

- Use string interpolation to concatenate short strings, as shown in the following code.

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- To append strings in loops, especially when you are working with large amounts of text, use a StringBuilder object.

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

Implicitly Typed Local Variables

- Use implicit typing for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.

```
// When the type of a variable is clear from the context, use var
// in the declaration.
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```

- Do not use var when the type is not apparent from the right side of the assignment.

```
// When the type of a variable is not clear from the context, use an
// explicit type.
int var4 = ExampleClass.ResultSoFar();
```

- Do not rely on the variable name to specify the type of the variable. It might not be correct.

```
// Naming the following variable inputInt is misleading.
// It is a string.
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Avoid the use of `var` in place of dynamic.
- Use implicit typing to determine the type of the loop variable in for and foreach loops.

  The following example uses implicit typing in a `for` statement.

```
var syllable = "ha";
var laugh = "";
for (var i = 0; i < 10; i++)
{
    laugh += syllable;
    Console.WriteLine(laugh);
}
```

  The following example uses implicit typing in a `foreach` statement.

```
foreach (var ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

Unsigned Data Type

- In general, use `int` rather than unsigned types. The use of `int` is common throughout C#, and it is easier to interact with other libraries when you use `int`.

Arrays

- Use the concise syntax when you initialize arrays on the declaration line.

```
// Preferred syntax. Note that you cannot use var here instead of string[].
string[] vowels1 = { "a", "e", "i", "o", "u" };


// If you use explicit instantiation, you can use var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a time.
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

Delegates

- Use the concise syntax to create instances of a delegate type.

```
// First, in class Program, define the delegate type and a method that
// has a matching signature.

// Define the type.
public delegate void Del(string message);

// Define a method that has a matching signature.
public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

```
// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.
Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.
Del exampleDel1 = new Del(DelMethod);
```

try-catch and using Statements in Exception Handling

- Use a try-catch statement for most exception handling.

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        throw;
    }
}
```

- Simplify your code by using the C# using statement. If you have a try-finally statement in which the only code in the `finally` block is a call to the Dispose method, use a `using` statement instead.

```
// This try-finally statement only calls Dispose in the finally block.
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}


// You can do the same thing with a using statement.
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset = font2.GdiCharSet;
}
```

&& and || Operators

- To avoid exceptions and increase performance by skipping unnecessary comparisons, use && instead of & and || instead of | when you perform comparisons, as shown in the following example.

```
Console.Write("Enter a dividend: ");
var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the following condition
// causes a run-time error. The && operator short circuits when the
// first expression is false. That is, it does not evaluate the
// second expression. The & operator evaluates both, and causes
// a run-time error when divisor is 0.
if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

New Operator

- Use the concise form of object instantiation, with implicit typing, as shown in the following declaration.

```
var instance1 = new ExampleClass();
```

The previous line is equivalent to the following declaration.

```
ExampleClass instance2 = new ExampleClass();
```

- Use object initializers to simplify object creation.

```
// Object initializer.
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

Event Handling

- If you are defining an event handler that you do not need to remove later, use a lambda expression.

```
public Form2()
{
    // You can use a lambda expression to define an event handler.
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

```
// Using a lambda expression shortens the following traditional definition.
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

Static Members

- Call static members by using the class name: *ClassName.StaticMember*. This practice makes code more readable by making static access clear. Do not qualify a static member defined in a base class with the name of a derived class. While that code compiles, the code readability is misleading, and the code may break in the future if you add a static member with the same name to the derived class.

LINQ Queries

- Use meaningful names for query variables. The following example uses `seattleCustomers` for customers who are located in Seattle.

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Use aliases to make sure that property names of anonymous types are correctly capitalized, using Pascal casing.

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and a distributor ID, instead of leaving them as `Name` and `ID` in the result, rename them to clarify that `Name` is the name of a customer, and `ID` is the ID of a distributor.

```
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { CustomerName = customer.Name, DistributorID = distributor.ID };
```

- Use implicit typing in the declaration of query variables and range variables.

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Align query clauses under the from clause, as shown in the previous examples.
- Use where clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

```
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Use multiple `from` clauses instead of a join clause to access inner collections. For example, a collection of `Student` objects might each contain a collection of test scores. When the following query is executed, it returns each score that is over 90, along with the last name of the student who received the score.

```
// Use a compound from to access the inner sequence within each element.
var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```

Security

Follow the guidelines in Secure Coding Guidelines.

See also

- Visual Basic Coding Conventions
- Secure Coding Guidelines

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