

C# Variables (examples and tutorials)

Variables are containers for storing data values.

In C#, there are different **types** of variables (defined with different keywords), for example:

- `int` - stores integers (whole numbers), without decimals, such as 123 or -123
- `double` - stores floating point numbers, with decimals, such as 19.99 or -19.99
- `char` - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- `string` - stores text, such as "Hello World". String values are surrounded by double quotes
- `bool` - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, you must specify the type and assign it a value:

Syntax

```
type variableName = value;
```

Where *type* is a C# type (such as `int` or `string`), and *variableName* is the name of the variable (such as `x` or `name`). The **equal sign** is used to assign values to the variable.

To create a variable that should store text, look at the following example:

Example

Create a variable called **name** of type `string` and assign it the value "**John**":

```
string name = "John";  
Console.WriteLine(name);
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{
```

```
class Program
{
    static void Main(string[] args)
    {
        string name = "John";
        Console.WriteLine(name);
    }
}
```

To create a variable that should store a number, look at the following example:

Example

Create a variable called **myNum** of type **int** and assign it the value **15**:

```
int myNum = 15;

Console.WriteLine(myNum);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myNum = 15;
            Console.WriteLine(myNum);
        }
    }
}
```

You can also declare a variable without assigning the value, and assign the value later:

Example

```
int myNum;

myNum = 15;

Console.WriteLine(myNum);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myNum;
            myNum = 15;
            Console.WriteLine(myNum);
        }
    }
}
```

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

Example

Change the value of `myNum` to 20:

```
int myNum = 15;

myNum = 20; // myNum is now 20

Console.WriteLine(myNum);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myNum = 15;
            myNum = 20;
            Console.WriteLine(myNum);
        }
    }
}
```

Constants

However, you can add the `const` keyword if you don't want others (or yourself) to overwrite existing values (this will declare the variable as "constant", which means unchangeable and read-only):

Example

```
const int myNum = 15;  
  
myNum = 20; // error
```

The `const` keyword is useful when you want a variable to always store the same value, so that others (or yourself) won't mess up your code. An example that is often referred to as a constant, is PI (3.14159...).

Note: You cannot declare a constant variable without assigning the value. If you do, an error will occur: A const field requires a value to be provided.

Other Types

A demonstration of how to declare variables of other types:

Example

```
int myNum = 5;  
  
double myDoubleNum = 5.99D;  
  
char myLetter = 'D';  
  
bool myBool = true;  
  
string myText = "Hello";
```

Display Variables

The `WriteLine()` method is often used to display variable values to the console window.

To combine both text and a variable, use the `+` character:

Example

```
string name = "John";  
  
Console.WriteLine("Hello " + name);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = "John";
            Console.WriteLine("Hello " + name);
        }
    }
}
```

You can also use the `+` character to add a variable to another variable:

Example

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName + lastName;
Console.WriteLine(fullName);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstName = "John ";
            string lastName = "Doe";
            string fullName = firstName + lastName;
            Console.WriteLine(fullName);
        }
    }
}
```

For numeric values, the `+` character works as a mathematical operator (notice that we use `int` (integer) variables here):

Example

```
int x = 5;

int y = 6;

Console.WriteLine(x + y); // Print the value of x + y
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            int y = 6;
            Console.WriteLine(x + y);
        }
    }
}
```

From the example above, you can expect:

- x stores the value 5
- y stores the value 6
- Then we use the `WriteLine()` method to display the value of `x + y`, which is **11**

Declare Many Variables

To declare more than one variable of the **same type**, use a comma-separated list:

Example

```
int x = 5, y = 6, z = 50;

Console.WriteLine(x + y + z);
```

Exercise (try it)

```
using System;

namespace MyApplication
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5, y = 6, z = 50;
            Console.WriteLine(x + y + z);
        }
    }
}
```

C# Identifiers

All C# **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

Example

```
// Good

int minutesPerHour = 60;

// OK, but not so easy to understand what m actually is

int m = 60;
```

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits and the underscore character (_)
- Names must begin with a letter
- Names should start with a lowercase letter and it cannot contain whitespace
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like C# keywords, such as `int` or `double`) cannot be used as names

C# Data Types

As explained in the variables chapter, a variable in C# must be a specified data type:

Example

```
int myNum = 5;           // Integer (whole number)
double myDoubleNum = 5.99D; // Floating point number
char myLetter = 'D';     // Character
bool myBool = true;      // Boolean
string myText = "Hello"; // String
```

exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myNum = 5;           // integer (whole number)
            double myDoubleNum = 5.99D; // floating point number
            char myLetter = 'D';     // character
            bool myBool = true;      // boolean
            string myText = "Hello"; // string
            Console.WriteLine(myNum);
            Console.WriteLine(myDoubleNum);
            Console.WriteLine(myLetter);
            Console.WriteLine(myBool);
            Console.WriteLine(myText);
        }
    }
}
```

A data type specifies the size and type of variable values. It is important to use the correct data type for the corresponding variable; to avoid errors, to save time and memory, but it will also make your code more maintainable and readable. The most common data types are:

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
bool	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter, surrounded by single quotes
string	2 bytes per character	Stores a sequence of characters, surrounded by double quotes

Numbers

Number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are `int` and `long`. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. Valid types are `float` and `double`.

Even though there are many numeric types in C#, the most used for numbers are `int` (for whole numbers) and `double` (for floating point numbers). However, we will describe them all as you continue to read.

Integer Types

Int

The `int` data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the `int` data type is the preferred data type when we create variables with a numeric value.

Example

```
int myNum = 100000;  
  
Console.WriteLine(myNum);
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int myNum = 100000;  
            Console.WriteLine(myNum);  
        }  
    }  
}
```

Long

The `long` data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when `int` is not large enough to store the value. Note that you should end the value with an "L":

Example

```
long myNum = 15000000000L;  
  
Console.WriteLine(myNum);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            long myNum = 15000000000L;
            Console.WriteLine(myNum);
        }
    }
}
```

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

Float

The `float` data type can store fractional numbers from $3.4e-038$ to $3.4e+038$. Note that you should end the value with an "F":

Example

```
float myNum = 5.75F;

Console.WriteLine(myNum);
```

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            float myNum = 5.75F;
            Console.WriteLine(myNum);
        }
    }
}
```

Double

The `double` data type can store fractional numbers from $1.7e-308$ to $1.7e+308$. Note that you can end the value with a "D" (although not required):

Example

```
double myNum = 19.99D;  
Console.WriteLine(myNum);
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            double myNum = 19.99D;  
            Console.WriteLine(myNum);  
        }  
    }  
}
```

Use `float` or `double`?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of `float` is only six or seven decimal digits, while `double` variables have a precision of about 15 digits. Therefore it is safer to use `double` for most calculations.

Booleans

A boolean data type is declared with the `bool` keyword and can only take the values `true` or `false`:

Example

```
bool isCSharpFun = true;  
bool isFishTasty = false;  
Console.WriteLine(isCSharpFun); // Outputs True
```

```
Console.WriteLine(isFishTasty); // Outputs False
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            bool isCSharpFun = true;
            bool isFishTasty = false;
            Console.WriteLine(isCSharpFun); // Outputs True
            Console.WriteLine(isFishTasty); // Outputs False
        }
    }
}
```

Strings

The `string` data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example

```
string greeting = "Hello World";
```

```
Console.WriteLine(greeting);
```

```
using System;

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

C# Type Casting

Type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

- **Implicit Casting** (automatically) - converting a smaller type to a larger type size
`char -> int -> long -> float -> double`
- **Explicit Casting** (manually) - converting a larger type to a smaller size type
`double -> float -> long -> int -> char`

Implicit Casting

Implicit casting is done automatically when passing a smaller size type to a larger size type:

Example

```
int myInt = 9;

double myDouble = myInt;           // Automatic casting: int to double

Console.WriteLine(myInt);          // Outputs 9

Console.WriteLine(myDouble);       // Outputs 9
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myInt = 9;
            double myDouble = myInt; // Automatic casting: int to double

            Console.WriteLine(myInt);
            Console.WriteLine(myDouble);
        }
    }
}
```

Explicit Casting

Explicit casting must be done manually by placing the type in parentheses in front of the value:

Example

```
double myDouble = 9.78;

int myInt = (int) myDouble;    // Manual casting: double to int

Console.WriteLine(myDouble);  // Outputs 9.78

Console.WriteLine(myInt);     // Outputs 9
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            double myDouble = 9.78;
            int myInt = (int) myDouble;    // Manual casting: double to int

            Console.WriteLine(myDouble);
            Console.WriteLine(myInt);
        }
    }
}
```

Type Conversion Methods

It is also possible to convert data types explicitly by using built-in methods, such

as `Convert.ToBoolean`, `Convert.ToDouble`, `Convert.ToString`, `Convert.ToInt32` (`int`) and `Convert.ToInt64` (`long`):

Example

```
int myInt = 10;

double myDouble = 5.25;

bool myBool = true;
```

```
Console.WriteLine(Convert.ToString(myInt));    // convert int to string
Console.WriteLine(Convert.ToDouble(myInt));    // convert int to double
Console.WriteLine(Convert.ToInt32(myDouble));  // convert double to int
Console.WriteLine(Convert.ToString(myBool));   // convert bool to
string
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myInt = 10;
            double myDouble = 5.25;
            bool myBool = true;

            Console.WriteLine(Convert.ToString(myInt));    // Convert int to
string
            Console.WriteLine(Convert.ToDouble(myInt));    // Convert int to
double
            Console.WriteLine(Convert.ToInt32(myDouble));  // Convert double
to int
            Console.WriteLine(Convert.ToString(myBool));   // Convert bool to
string
        }
    }
}
```

Why Conversion?

Many times, there's no need for type conversion. But sometimes you have to. Take a look at the next chapter, when working with user input, to see an example of this.

Get User Input

You have already learned that `Console.WriteLine()` is used to output (print) values. Now we will use `Console.ReadLine()` to get user input.

In the following example, the user can input his or hers username, which is stored in the variable `userName`. Then we print the value of `userName`:

Example

```
// Type your username and press enter
Console.WriteLine("Enter username:");

// Create a string variable and get user input from the keyboard and
store it in the variable

string userName = Console.ReadLine();

// Print the value of the variable (userName), which will display the
input value

Console.WriteLine("Username is: " + userName);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Type your username and press enter
            Console.WriteLine("Enter username:");

            // Create a string variable and get user input from the keyboard
            and store it in the variable
            string userName = Console.ReadLine();

            // Print the value of the variable (userName), which will display
            the input value
            Console.WriteLine("Username is: " + userName);
        }
    }
}
```

User Input and Numbers

The `Console.ReadLine()` method returns a `string`. Therefore, you cannot get information from another data type, such as `int`. The following program will cause an error:

Example

```
Console.WriteLine("Enter your age:");  
  
int age = Console.ReadLine();  
  
Console.WriteLine("Your age is: " + age);
```

The error message will be something like this:

```
Cannot implicitly convert type 'string' to 'int'
```

Like the error message says, you cannot implicitly convert type 'string' to 'int'.

Luckily, for you, you just learned from the previous chapter (Type Casting), that you can convert any type explicitly, by using one of the `Convert.To` methods:

Example

```
Console.WriteLine("Enter your age:");  
  
int age = Convert.ToInt32(Console.ReadLine());  
  
Console.WriteLine("Your age is: " + age);
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Enter your age:");  
            int age = Convert.ToInt32(Console.ReadLine());  
            Console.WriteLine("Your age is: " + age);  
        }  
    }  
}
```

Note: If you enter wrong input (e.g. text in a numerical input), you will get an exception/error message (like `System.FormatException: 'Input string was not in a correct format.'`).

C# Operators

Operators are used to perform operations on variables and values.

In the example below, we use the **+** **operator** to add together two values:

Example

```
int x = 100 + 50;
```

Although the **+** operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example

```
int sum1 = 100 + 50;           // 150 (100 + 50)
int sum2 = sum1 + 250;         // 400 (150 + 250)
int sum3 = sum2 + sum2;        // 800 (400 + 400)
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int sum1 = 100 + 50;           // 150 (100 + 50)
            int sum2 = sum1 + 250;         // 400 (150 + 250)
            int sum3 = sum2 + sum2;        // 800 (400 + 400)
            Console.WriteLine(sum1);
            Console.WriteLine(sum2);
            Console.WriteLine(sum3);
        }
    }
}
```

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations:

Operator	Name	Description	Example
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	$x++$
--	Decrement	Decreases the value of a variable by 1	$x--$

C# Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

```
int x = 10;
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            Console.WriteLine(x);
        }
    }
}
```

The **addition assignment** operator (**+=**) adds a value to a variable:

Example

```
int x = 10;
```

```
x += 5;
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            x += 5;
            Console.WriteLine(x);
        }
    }
}
```

A list of all assignment operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Exercise (try all these operators)

```
using System;
```

```
namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            x <= 3;
            Console.WriteLine(x);
        }
    }
}
```

C# Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            int y = 3;
            Console.WriteLine(x <= y); // returns False because 5 is neither
less than or equal to 3
        }
    }
}
```

C# Logical Operators

Logical operators are used to determine the logic between variables or values:

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	$x < 5 \ \&\& \ x < 10$
	Logical or	Returns true if one of the statements is true	$x < 5 \ \ x < 4$
!	Logical not	Reverse the result, returns false if the result is true	$!(x < 5 \ \&\& \ x < 10)$

Exercise (try above operators)

```
using System;

namespace MyApplication
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            Console.WriteLine(!(x > 3 && x < 10)); // returns False because !
(not) is used to reverse the result
        }
    }
}
```

C# Strings

Strings are used for storing text.

A **string** variable contains a collection of characters surrounded by double quotes:

String Length

A string in C# is actually an object, which contain properties and methods that can perform certain operations on strings. For example, the length of a string can be found with the **Length** property:

Example

```
string txt = "ABCDEFGHJKLMNOPQRSTUVWXYZ";

Console.WriteLine("The length of the txt string is: " + txt.Length);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string txt = "ABCDEFGHJKLMNOPQRSTUVWXYZ";
            Console.WriteLine("The length of the txt string is: " +
txt.Length);
        }
    }
}
```

Other Methods

There are many string methods available, for example `ToUpper()` and `ToLower()`, which returns a copy of the string converted to uppercase or lowercase:

Example

```
string txt = "Hello World";  
Console.WriteLine(txt.ToUpper()); // Outputs "HELLO WORLD"  
Console.WriteLine(txt.ToLower()); // Outputs "hello world"
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string txt = "Hello World";  
            Console.WriteLine(txt.ToUpper()); // Outputs "HELLO WORLD"  
            Console.WriteLine(txt.ToLower()); // Outputs "hello world"  
        }  
    }  
}
```

String Concatenation

The `+` operator can be used between strings to combine them. This is called **concatenation**:

Example

```
string firstName = "John ";  
string lastName = "Doe";  
string name = firstName + lastName;  
Console.WriteLine(name);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstName = "John ";
            string lastName = "Doe";
            string name = firstName + lastName;
            Console.WriteLine(name);
        }
    }
}
```

Note that we have added a space after "John" to create a space between firstName and lastName on print.

You can also use the `string.Concat()` method to concatenate two strings:

Example

```
string firstName = "John ";
string lastName = "Doe";
string name = string.Concat(firstName, lastName);
Console.WriteLine(name);
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstName = "John ";
            string lastName = "Doe";
            string name = string.Concat(firstName, lastName);
            Console.WriteLine(name);
        }
    }
}
```

String Interpolation

Another option of string concatenation, is **string interpolation**, which substitutes values of variables into placeholders in a string. Note that you do not have to worry about spaces, like with concatenation:

Example

```
string firstName = "John";  
string lastName = "Doe";  
string name = $"My full name is: {firstName} {lastName}";  
Console.WriteLine(name);
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string firstName = "John";  
            string lastName = "Doe";  
            string name = $"My full name is: {firstName} {lastName}";  
            Console.WriteLine(name);  
        }  
    }  
}
```

Also note that you have to use the dollar sign (\$) when using the string interpolation method.

String interpolation was introduced in C# version 6.

Access Strings

You can access the characters in a string by referring to its index number inside square brackets `[]`.

This example prints the **first character** in **myString**:

Example

```
string myString = "Hello";  
Console.WriteLine(myString[0]); // Outputs "H"
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string myString = "Hello";  
            Console.WriteLine(myString[0]);  
        }  
    }  
}
```

Note: String indexes start with 0: [0] is the first character. [1] is the second character, etc.

This example prints the **second character** (1) in **myString**:

Example

```
string myString = "Hello";  
Console.WriteLine(myString[1]); // Outputs "e"
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string myString = "Hello";  
            Console.WriteLine(myString[1]);  
        }  
    }  
}
```

You can also find the index position of a specific character in a string, by using the `IndexOf()` method:

Example

```
string myString = "Hello";  
Console.WriteLine(myString.IndexOf("e")); // Outputs "1"
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string myString = "Hello";  
            Console.WriteLine(myString.IndexOf("e"));  
        }  
    }  
}
```

Another useful method is `Substring()`, which extracts the characters from a string, starting from the specified character position/index, and returns a new string. This method is often used together with `IndexOf()` to get the specific character position:

Example

```
// Full name  
string name = "John Doe";  
  
// Location of the letter D  
int charPos = name.IndexOf("D");  
  
// Get last name  
string lastName = name.Substring(charPos);
```

```
// Print the result  
Console.WriteLine(lastName);
```

Exercise (try it)

```
using System;  
  
namespace GetLastName  
{  
    class Program  
    {  
        static void Main()  
        {  
            // Full name  
            string name = "John Doe";  
  
            // Location of the letter D  
            int charPos = name.IndexOf("D");  
  
            // Get last name  
            string lastName = name.Substring(charPos);  
  
            // Print the result  
            Console.WriteLine(lastName);  
        }  
    }  
}
```

Special Characters

Because strings must be written within quotes, C# will misunderstand this string, and generate an error:

```
string txt = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

Escape character	Result	Description
------------------	--------	-------------

\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

The sequence `\"` inserts a double quote in a string:

Example

```
string txt = "We are the so-called \"Vikings\" from the north.";

using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string txt = "We are the so-called \"Vikings\" from the north.";
            Console.WriteLine(txt);
        }
    }
}
```

Please try all the exercises..

References

www.w3schools.com