

# C# Basic Documentation

## Types

Type	Description
<b>byte</b>	8-bit unsigned integer
<b>sbyte</b>	8-bit signed integer
<b>short</b>	16-bit signed integer
<b>ushort</b>	16-bit unsigned integer
<b>int</b>	32-bit signed integer
<b>uint</b>	32-bit unsigned integer
<b>long</b>	64-bit signed integer
<b>ulong</b>	64-bit unsigned integer
<b>float</b>	32-bit Single-precision floating point type
<b>double</b>	64-bit double-precision floating point type
<b>decimal</b>	128-bit decimal type for financial and monetary calculations
<b>char</b>	16-bit single Unicode character
<b>bool</b>	8-bit logical true/false value
<b>object</b>	Base type of all other types.
<b>string</b>	A sequence of Unicode characters

## Console Reading/Writing

### String Console.ReadLine()

Writes the specified string value, followed by the current line terminator, to the standard output stream.

#### Returns

String

The next line of characters from the input stream, or **null** if no more lines are available.

### Console.WriteLine(String text)

Writes the specified string value, followed by the current line terminator, to the standard output stream.

### Console.WriteLine(String text, Object[] objects)

Writes the text representation of the specified array of objects, followed by the current line terminator, to the standard output stream using the specified format information.

```
static void Main(string[] args)
{
    string world = "World";
    Console.WriteLine("Hello World!");
    Console.WriteLine("Hello " + world + "!");
    Console.WriteLine("Hello {0}!", world);
    Console.WriteLine($"Hello {world}!");
    string sentence = Console.ReadLine();
}
```

## Random Class

Represents a pseudo-random number generator. The Random class methods can be used to obtain random numbers.

### int Next()

Returns a non-negative random integer.

*Returns*

Integer

### int Next(int maxValue)

Returns a non-negative random integer that is less than the specified maximum.

*Returns*

Integer

### int Next(int minValue, int maxValue)

Returns a random integer that is between the specified range, excluding the maximum number.

*Returns*

Integer

```
static void Main(string[] args)
{
    Random randomObject = new Random(); //create the random class object
    int number1 = randomObject.Next(); //non-negative number
    int number2 = randomObject.Next(10); //number from 0 to 9
    int number3 = randomObject.Next(1, 101); //number from 1 to 100
}
```

## Conversion from String

Every built-in type has a parsing method that converts a string to the corresponding type.

### T.Parse(String text)

Parses the provided string to the indicated type (T).

#### Returns

T

Returns the parsed string.

```
static void Main(string[] args)
{
    string text = "0";
    int number = int.Parse(text);
    float numberFloat = float.Parse(text);

    string text2 = "true";
    bool booleanValue = bool.Parse(text2);
}
```

## Selection Structures

### The if statement

Selects a statement to execute based on the value of a Boolean expression.

An if statement without an else part executes its body only if a Boolean expression evaluates to true. An if statement with else/else if parts selects one of the statements to execute based on the Boolean expression. Several if statements can be nested to check multiple conditions.

```
int value = 0;
if (value > 50)
{
    Console.WriteLine("Larger than 50");
}
else if (value > 10)
{
    Console.WriteLine("Larger than 10");
}
else
{
    Console.WriteLine("Something else...");
}
```

## The **switch** statement

Selects a statement list to execute based on a pattern match with an expression.

The **switch** statement selects a statement list to execute based on a pattern match with a match expression. One or more constant cases can be defined to which the expression is compared.

A **default** case can be defined, which specifies statements to execute when a match expression doesn't match any other case pattern. If a match expression doesn't match any case pattern and there is no **default** case, control falls through a **switch** statement.

The statement lists associated to each case must end with a **break** statement.

```
int number = 2;
switch (number)
{
    case 1:
        Console.WriteLine("Value is one.");
        break;

    case 2:
        Console.WriteLine("Value is two.");
        break;

    default:
        Console.WriteLine($"Value is {number}.");
        break;
}
```

## Iteration (repetition) Structures

### The **while** statement

The **while** statement executes a statement or a block of statements while a specified Boolean expression evaluates to **true**. Because that expression is evaluated before each execution of the loop, a **while** loop executes zero or more times.

```
int n = 0;
while (n < 5)
{
    Console.Write(n);
    n++;
}
```

## The **do** statement

The **do** statement executes a statement or a block of statements while a specified Boolean expression evaluates to **true**. Because that expression is evaluated after each execution of the loop, a **do** loop executes one or more times.

```
int n = 0;
do
{
    Console.Write(n);
    n++;
} while (n < 5);
```

## The **for** statement

The **for** statement executes a statement or a block of statements while a specified Boolean expression evaluates to **true**.

It is composed of three statements:

- the first statement is executed (one time) before the execution of the code block.
- the second statement defines the condition for executing the code block.
- The third statement is executed (every time) after the code block has been executed.

The following example shows the **for** statement that executes its body while an integer counter is less than three:

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
```

## The **foreach** statement

The **foreach** statement executes a statement or a block of statements for each element within a specific collection.

```
int[] array3 = { 1, 2, 3, 4, 5, 6 };

foreach(int number in array3)
{
    Console.WriteLine(number);
}
```

## Jump Structures

### The **break** statement

The **break** statement terminates the closest enclosing loop or **switch** statement in which it appears. Control is passed to the statement that follows the terminated statement, if any.

```
for (int i = 0; i < 3; i++)
{
    if(i == 2)
        break;
    Console.Write(i);
}
```

### The **continue** statement

The **continue** statement passes control to the next iteration of the enclosing iteration statement in which it appears. If there is no following iteration, control is passed to the statement that follows iteration statement, if any.

```
for (int i = 0; i < 3; i++)
{
    if(i == 2)
        break;
    Console.Write(i);
}
```

### The **return** statement

The **return** statement terminates execution of the method in which it appears and returns control to the calling method. It can also return an optional value. If the method is a **void** type, the **return** statement can be omitted.

```
static double CalculateArea(int r)
{
    double area = r * r * Math.PI;
    return area;
}

static void Main()
{
    int radius = 5;
    double result = CalculateArea(radius);
    Console.WriteLine("The area is {0:0.00}", result);
}
```

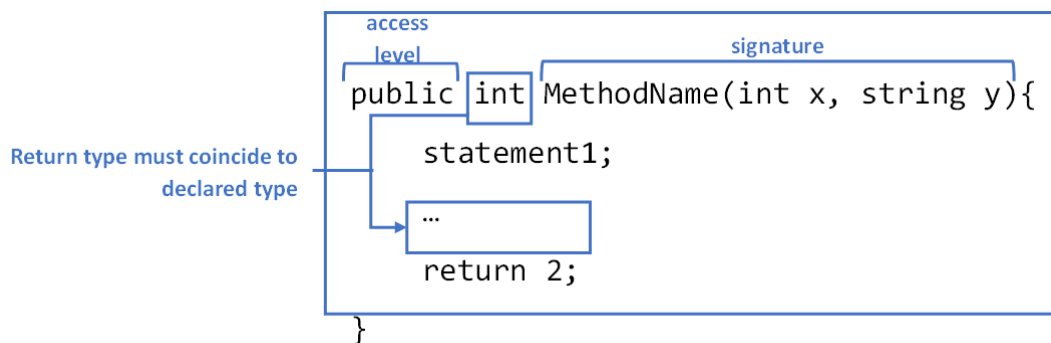
## Methods

Methods are code blocks that correspond to a group of statements. The contained statements are executed when the method is called.

One or more parameters may be defined in a method's declaration. Parameters are the list of variables provided externally to a method. When calling a method, the values that are passed to those variables are called arguments.

Methods have a signature, which is their unique identification in the program. In C#, it is composed by the method's name and the parameter types

A method's declaration includes characteristics that are not part of the signature: access level and return type.



## Strings

A string is an object of type `String` whose value is text. Internally, the text is stored as a sequential read-only collection of `Char` objects.

In C#, the `string` keyword is an alias for `String`. Therefore, `String` and `string` are equivalent.

Strings can be initialized with `null` value, `string literals` or from an `array` of chars. They are immutable: when a string variable is modified, a new object is created. The `Length` property of a string represents the number of `Char` objects it contains.

```
static void Main(string[] args)
{
    string a = "Hello", b = "World";
    string c = "Hello World";
    string d = "He";
    d += "llo";
    string e = a + " " + b;
}
```

```

Console.WriteLine(a == d);
Console.WriteLine(object.ReferenceEquals(a, d));

char letter = 'H';
char letter2 = a[0];
Console.WriteLine(letter == letter2);
}

```

Methods	Definitions	Example
<b>Clone()</b>	Make clone of string.	str2 = str1.Clone();
<b>CompareTo()</b>	Compare two strings and returns integer value as output. It returns 0 for true and 1 for false.	int compare = str2.CompareTo(str1);
<b>Contains()</b>	checks whether specified character or string is exists or not in the string value.	bool check = str2.Contains("hack");
<b>EndsWith()</b>	checks whether specified character(s) is/are the last character(s) of string or not.	bool check = str2.EndsWith("io");
<b>Equals()</b>	compares two string and returns Boolean value true as output if they are equal, false if not	bool check = str2.Equals(str1);
<b>IndexOf()</b>	Returns the index position of first occurrence of specified character.	int index= str1.IndexOf(":")
<b>ToLower()</b>	Converts String into lower case based on rules of the current culture.	str1 = str1.ToLower();
<b>ToUpper()</b>	Converts String into Upper case based on rules of the current culture.	str1 = str1.ToUpper();
<b>Insert()</b>	Insert the string or character in the string at the specified position.	str1.Insert(0, "Welcome"); str1.Insert(i, "Thank You");
<b>LastIndexOf()</b>	Returns the index position of last occurrence of specified character.	int index = str1.LastIndexOf("T");



<b>Length</b>	returns length of string.	<code>int length = str1.Length;</code>
<b>Remove()</b>	deletes all the characters from beginning to specified index position.	<code>string str2 = str1.Remove(i);</code>
<b>Replace()</b>	replaces the specified character with another	<code>string str2 = str1.Replace('a', 'e');</code>
<b>Split()</b>	This method splits the string based on specified value.	<code>str1 = "Hello and welcome"; string[] arr = str1.Split("and");</code>
<b>StartsWith()</b>	Checks whether the first character(s) of string is/are same as specified character(s).	<code>bool check = str1.StartsWith("H");</code>
<b>Substring()</b>	This method returns substring.	<code>string str2 = str1.Substring(1, 7);</code>
<b>ToCharArray()</b>	Converts string into char array.	<code>String[] arr = str1.ToCharArray();</code>
<b>Trim()</b>	It removes extra whitespaces from beginning and ending of string.	<code>string str2 = str1.Trim();</code>

## Arrays

An array is a data structure that contains a number of variables which are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

The length of an array cannot be changed after declaration, so arrays need to be initialized when declared.

Similar to the `string` keyword, `array` is an alias for the `Array` class. Values in an `array` can be accessed and changed by using the indexer operator.

```
static void Main(string[] args)
{
    // Declare a single-dimensional array of 5 integers.
    int[] array1 = new int[5];
}
```

```
// Declare and set array element values.
int[] array2 = new int[] { 1, 3, 5, 7, 9 };

// Alternative syntax.
int[] array3 = { 1, 2, 3, 4, 5, 6 };
array3[0] = 0;

foreach (int number in array3)
{
    Console.WriteLine(number);
}
}
```

Methods	Definitions	Example
<b>Clear()</b>	Clears the contents of an array.	int [] numbers = new int{1,2,3}; numbers.Clear();
<b>IndexOf()</b>	Searches for the specified object and returns the index of its first occurrence in a one-dimensional array.	int [] numbers = new int{1,2,3}; int index=numbers.IndexOf(2);
<b>LastIndexOf()</b>	Searches for the specified object and returns the index of the last occurrence within the entire one-dimensional Array.	int [] numbers = new int{1,2,3}; int index = numbers.LastIndexOf(2);
<b>Reverse()</b>	Reverses the sequence of the elements in the entire one-dimensional Array.	int [] numbers = new int{1,2,3}; numbers.Reverse();
<b>SetValue()</b>	Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.	int [] numbers = new int{1,2,3}; numbers.SetValue(0, 10);
<b>Sort()</b>	Sorts the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array.	int [] numbers = new int{1,2,3}; numbers.Sort();

## Lists

The List class represents strongly-typed lists of elements. Lists have **dynamic** sizing, which is controlled by its capacity. **Capacity** refers to the number of elements the internal data structure can hold without resizing.

```
static void Main(string[] args)
{
    // Declare an empty list of integers, and a list of strings with 10 of capacity.
    List<int> numbers = new List<int>();
    List<string> words = new List<string>(10);
    // Declare a list of numbers, pre-filled with 5 integers (defined in an array)
    List<int> existingNumbers = new List<int>(new int[] { 1, 2, 3, 4, 5 });
}
```

Methods	Definitions	Example
<b>Add(T)</b>	Adds an object to the end of the List<T>.	List<int> numbers = new List<int>(); numbers.Add(5);
<b>AddRange()</b>	Adds the elements of the specified collection to the end of the List<T>.	List<int> numbers = new List<int>(); numbers.Add(new int[]{1,2,3});
<b>Contains()</b>	Determines whether an element is in the List<T>.	List<int> numbers = new List<int>(); numbers.Add(new int[]{1,2,3}); if(numbers.Contains(1)){...}
<b>IndexOf()</b>	Searches for the specified object and returns the zero-based index of the first occurrence within the entire List<T>.	List<int> numbers = new List<int>(); numbers.Add(new int[]{1,2,3}); int index = numbers.IndexOf(1);
<b>Insert()</b>	Inserts an element into the List<T> at the specified index.	List<string> lt = new List<string>(); lt.Add(new string[]{"Hello", "!"}); lt.Insert(1, "World");
<b>Remove()</b>	Removes the first occurrence of a specific object from the List<T>.	List<int> numbers = new List<int>(); numbers.Add(new int[]{1,2,3}); numbers.Remove(1);
<b>RemoveAt()</b>	Removes the element at the specified index of the List<T>.	List<string> lt = new List<string>(); lt.Add(new string[]{"Hello", "!"}); lt.RemoveAt(1);
<b>Reverse()</b>	Reverses the order of the elements in the specified range.	List<int> numbers = new List<int>(); numbers.Add(new int[]{1,2,3}); numbers.Reverse();
<b>Sort()</b>	Sorts the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array.	List<int> numbers = new List<int>(); numbers.Add(new int[]{2,3,1}); numbers.Sort();
<b>ToArray()</b>	Copies the elements of the List<T> to a new array.	List<int> numbers = new List<int>(); numbers.Add(new int[]{2,3,1}); int[] nArray = numbers.ToArray();

## Queues

The Queue<T> class represents a **strongly-typed collection** of elements. In queues, elements are inserted at the end and removed from the front. This implies a **First-In-First-Out behavior (FIFO)**.

```
static void Main(string[] args)
{
    // Declare an empty queue of integers, and a queue of strings with 10 of capacity.
    Queue<int> numbers = new Queue<int>();
    Queue<string> words = new Queue<string>(10);
}
```

Methods	Definitions	Example
<b>Contains(T)</b>	Determines whether an element is in the Queue<T>.	Queue<int> numbers = new Queue<int>(); numbers.Enqueue(1); if(numbers.Contains(1)){...}
<b>Dequeue()</b>	Removes and returns the object at the beginning of the Queue<T>.	Queue<int> numbers = new Queue<int>(); numbers.Enqueue(1); int number = numbers.Dequeue();
<b>Enqueue(T)</b>	Adds an object to the end of the Queue<T>.	Queue<int> numbers = new Queue<int>(); numbers.Enqueue(1);
<b>Peek()</b>	Returns the object at the beginning of the Queue<T> without removing it.	Queue<int> numbers = new Queue<int>(); numbers.Enqueue(1); int number = numbers.Peek();
<b>ToArray()</b>	Copies the Queue<T> elements to a new array.	Queue<int> numbers = new Queue<int>(); numbers.Enqueue(1); numbers.Enqueue(3); int[] nArray = numbers.ToArray();

## Stacks

The Stack<T> class represents a strongly-typed collection of elements. In stacks, elements are **inserted and removed from the top**. This implies a **Last-In-First-Out behavior (LIFO)**.

```
static void Main(string[] args)
{
    // Declare an empty stack of integers, and a stack of strings with 10 of capacity.
    Stack<int> numbers = new Stack<int>();
    Stack<string> words = new Stack<string>(10);
}
```

Methods	Definitions	Example
<b>Contains(T)</b>	Determines whether an element is in the Stack<T>.	Stack<int> numbers = new Stack<int>(); numbers.Push(1); if(numbers.Contains(1)){...}
<b>Pop()</b>	Removes and returns the object at the top of the Stack<T>.	Stack<int> numbers = new Stack<int>(); numbers.Push(1); int number = numbers.Pop();
<b>Push(T)</b>	Inserts an object at the top of the Stack<T>.	Stack<int> numbers = new Stack<int>(); numbers.Push(1);
<b>Peek()</b>	Returns the object at the top of the Stack<T> without removing it.	Stack<int> numbers = new Stack<int>(); numbers.Push(1); int number = numbers.Peek();
<b>ToArray()</b>	Copies the Stack<T> to a new array.	Stack<int> numbers = new Stack<int>(); numbers.Push(1); int[] nArray = numbers.ToArray();

## Dictionaries

The Dictionary class is a **collection of key-value pairs**. These elements are stored without any order in specific. The values can be quickly obtained using the associated key.

```
static void Main(string[] args)
{
    // Declare an empty dictionary of key-value pairs (integer-string)
    Dictionary<int, string> names = new Dictionary<int, string>();
    // Declare an empty dictionary of key-value pairs (integer-integer), capacity of 10
    Dictionary<int, int> phoneNumbers = new Dictionary<int, int>(10);
    // Declare a dictionary of key-value pairs (integer-integer), pre-filled with 2 pairs
    Dictionary<int, int> existingNumbers = new Dictionary<int, int> {{0,20}, {1,54}};
}
```

Methods	Definitions	Example
<b>Add(TK,TV)</b>	Adds the specified key and value to the dictionary.	Dictionary<int, string> names = new Dictionary<int, string>(); names.Add(0, "John");
<b>ContainsKey(TK)</b>	Determines whether the dictionary contains the specified key.	Dictionary<int, string> names = new Dictionary<int, string>(); names.Add(0, "John"); if(names.ContainsKey(0)){...}
<b>ContainsValue(TV)</b>	Determines whether the dictionary contains the specified value.	Dictionary<int, string> names = new Dictionary<int, string>(); names.Add(0, "John"); if(names.ContainsValue("John")){...}
<b>Remove(TK)</b>	Removes the value with the specified key from the dictionary.	Dictionary<int, string> names = new Dictionary<int, string>(); names.Add(0, "John"); names.Remove(0);
<b>TryGetValue(TK)</b>	Gets the value associated with the specified key.	Dictionary<int, string> names = new Dictionary<int, string>(); names.Add(0, "John");  string val = ""; if (names.TryGetValue(0, out val)) { Console.WriteLine("Value:" + val); }