Introduction to C# basics

Table of contents

Main types in C#

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

Declaring data types

```
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
```

The general rules for naming variables 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

Getting values from the console

```
string tmp = Console.ReadLine();
```

- declaration of a string which is a type of text
- it gets the value that the Console.ReadLine(); reads from the console

```
int num = 0;
```
//declaration of an integer which is a type of number with the value of 0

```
num = Convert.ToInt32(tmp);
```

- the num variable gets the value from the console
- the text read from the console needs to be converted into the type of the I variable

This process can be shortened, and squashed into one line as the following example shows:

```
int num2 = Convert.ToInt32(Console.ReadLine());
```

Printing values back to the console

- WriteLine prints a whole line and puts the cursor into the next line:
  ```
  Console.WriteLine("This is printed onto the console");
  ```
- Write leaves the cursor in the same line:
  ```
  Console.Write("This is printed onto the console");
  ```
- If we use \n between the "" we get the equivalent of WriteLine:
  ```
  Console.Write("This is printed onto the console\n");
  ```
- Examples of printing the values from variables:
  - `int x = 2; int y = 3;`
  - `Console.WriteLine(x);`
    - result on the console: 2
  - `Console.WriteLine("x = {0}", x);`
    - result on the console: `x = 2`
  - `Console.WriteLine("{0} * {1} = {2}", x, y, x*y);`
    - result on the console: `2 * 3 = 6`
    - {0} points to the first variable (x)

- {1} points to the second variable (y)
- {2} is printing the result of the equations (x*y)

- `Console.WriteLine($"Longest word {words.Split(' ')[index]}");`

Comments

These lines are going. to be ignored when running the code

```
// This is a single line comment


/*
  This is a multiple line comment.
*/
```

Operators

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--

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

Logical Operators

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)

Conditions

If

C# supports the usual logical conditions from mathematics:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`
- You can use these conditions to perform different actions for different decisions.

C# has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

Use the `if` statement to specify a block of C# code to be executed if a condition is `True`.

Syntax

```
if (condition)
{
  // block of code to be executed if the condition is True
}
```

Note that `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.
In the example below, we test two values to find out if 20 is greater than 18. If the condition is `True`, print some text:

Example

```
if (20 > 18)
{
  Console.WriteLine("20 is greater than 18");
}
```

We can also test variables:

Example

```
int x = 20;
int y = 18;
if (x > y)
{
  Console.WriteLine("x is greater than y");
}
```

Example explained

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the `>` operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

If else

```
int time = 20;
if (time < 18)
```

```
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

Example explained

In the example above, time (20) is greater than 18, so the condition is `False`. Because of this, we move on to the `else` condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

Else If

Use the else if statement to specify a new condition if the first condition is False.

Syntax

```
if (condition1)
{
  // block of code to be executed if condition1 is True
}
else if (condition2)
{
  // block of code to be executed if the condition1 is false and condition2 is True
}
else
{
  // block of code to be executed if the condition1 is false and condition2 is
False
}
```

Example

```
int time = 22;
if (time < 10)
{
  Console.WriteLine("Good morning.");
}
else if (time < 20)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

Example explained

In the example above, time (22) is greater than 10, so the **first condition** is False. The next condition, in the else if statement, is also False, so we move on to the else condition since **condition1** and **condition2** is both False - and print to the screen "Good evening". However, if the time was 14, our program would print "Good day."

Short Hand If Else

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

Syntax

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Instead of writing:

Example

```
int time = 20;
if (time < 18)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
```

Instead you can simply write:

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
Console.WriteLine(result);
```

Switch

Use the switch statement to select one of many code blocks to be executed.

Syntax

```
switch(expression)
{
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
    break;
}
```

This is how it works:

- The switch expression is evaluated once
- The value of the expression is compared with the values of each case
- If there is a match, the associated block of code is executed
- The break and default keywords will be described later in this chapter
- The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;
switch (day)
```

```
{
  case 1:
    Console.WriteLine("Monday");
    break;
  case 2:
    Console.WriteLine("Tuesday");
    break;
  case 3:
    Console.WriteLine("Wednesday");
    break;
  case 4:
    Console.WriteLine("Thursday");
    break;
  case 5:
    Console.WriteLine("Friday");
    break;
  case 6:
    Console.WriteLine("Saturday");
    break;
  case 7:
    Console.WriteLine("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

The break Keyword

When C# reaches a break keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The default Keyword

The default keyword is optional and specifies some code to run if there is no case match:

Example

```
int day = 4;
switch (day)
{
  case 6:
    Console.WriteLine("Today is Saturday.");
    break;
  case 7:
    Console.WriteLine("Today is Sunday.");
    break;
  default:
    Console.WriteLine("Looking forward to the Weekend.");
    break;
}
// Outputs "Looking forward to the Weekend."
```

Loops

For

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

Syntax

```
for (statement 1; statement 2; statement 3)
{
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.
Statement 2 defines the condition for executing the code block.
Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example

```
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

Statement 1 sets a variable before the loop starts (int i = 0).
Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.
Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Foreach

There is also a foreach loop, which is used exclusively to loop through elements in an **array**:

Syntax

```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

The following example outputs all elements in the **cars** array, using a foreach loop:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
foreach (string i in cars)
{
  Console.WriteLine(i);
}
```

While

The while loop loops through a block of code as long as a specified condition is True:

Syntax

```
while (condition)
{
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0;
while (i < 5)
{
  Console.WriteLine(i);
  i++;
}
```

Do while

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do
{
  // code block to be executed
}
while (condition);
```

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

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

Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value. To declare an array, define the variable type with **square brackets**:

```
string[] cars; // This variable holds an array of strings.
```

To manually insert values into it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, we could write:

```
int[] myNumArr = {10, 20, 30, 40};
```

You access an array element by referring to the index number.
This statement accesses the value of the first element in **cars**:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
Console.WriteLine(cars[0]);
// Outputs Volvo
```

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
Console.WriteLine(cars[0]);
// Now outputs Opel instead of Volvo
```

Array Length

To find out how many elements an array has, use the Length property:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
Console.WriteLine(cars.Length);
// Outputs 4
```

Loop Through an Array

You can loop through the array elements with the for loop, and use the Length property to specify how many times the loop should run.
The following example outputs all elements in the **cars** array:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.Length; i++)
{
  Console.WriteLine(cars[i]);
}
```

The foreach Loop

There is also a foreach loop, which is used exclusively to loop through elements in an **array**:

Syntax

```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

The following example outputs all elements in the **cars** array, using a foreach loop:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
foreach (string i in cars)
{
  Console.WriteLine(i);
}
```

The example above can be read like this: **for each** string element (called **i** - as in **i**ndex) in **cars**, print out the value of **i**.

If you compare the for loop and foreach loop, you will see that the foreach method is easier to write, it does not require a counter (using the Length property), and it is more readable.

Other Ways to Create an Array

If you are familiar with C#, you might have seen arrays created with the new keyword, and perhaps you have seen arrays with a specified size as well. In C#, there are different ways to create an array:

```
// Create an array of four elements, and add values later
string[] cars = new string[4];

// Create an array of four elements and add values right away
string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements without specifying the size
string[] cars = new string[] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements, omitting the new keyword, and
without specifying the size
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

It is up to you which option you choose. In our tutorial, we will often use the last option, as it is faster and easier to read.

However, you should note that if you declare an array and initialize it later, you have to use the new keyword:

```
// Declare an array
string[] cars;

// Add values, using new
cars = new string[] {"Volvo", "BMW", "Ford"};

// Add values without using new (this will cause an error)
cars = {"Volvo", "BMW", "Ford"};
```