Csharp for intermediate level:

C# Math

The C# Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.Max(*x,y*)

The `Math.Max(x,y)` method can be used to find the highest value of *x* and *y*:

Example

```
Math.Max(5, 10);
```

Math.Min(*x,y*)

The `Math.Min(x,y)` method can be used to find the lowest value of of *x* and *y*:

Example

```
Math.Min(5, 10);
```

Math.Sqrt(*x*)

The `Math.Sqrt(x)` method returns the square root of *x*:

Example

```
Math.Sqrt(64);
```

C# Strings

Strings are used for storing text.

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

Example

Create a variable of type `string` and assign it a value:

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

A string variable can contain many words, if you want:

Example

```
string greeting2 = "Nice to meet you!";
```

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 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
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"
```

C# String Concatenation

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);
```

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);
```

Adding Numbers and Strings

If you add two numbers, the result will be a number:

Example

```
int x = 10;

int y = 20;

int z = x + y;  // z will be 30 (an integer/number)
```

If you add two strings, the result will be a string concatenation:

Example

```
string x = "10";

string y = "20";

string z = x + y;  // z will be 1020 (a string)
```

C# String Interpolation

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);
```

C# Access Strings

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"
```

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

Example

```
string myString = "Hello";

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

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"
```

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);
```

C# Special Characters

Strings - 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.";
```

The sequence `\'` inserts a single quote in a string:

Example

```
string txt = "It\'s alright.";
```

The sequence `\\` inserts a single backslash in a string:

Example

```
string txt = "The character \\ is called backslash.";
```

Other useful escape characters in C# are:

Code	**Result**
\n	New Line
\t	Tab
\b	Backspace

C# Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, C# has a `bool` data type, which can take the values `true` or `false`.

Boolean Values

A boolean 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
```

However, it is more common to return boolean values from boolean expressions, for conditional testing (see below).

Boolean Expression

A Boolean expression returns a boolean value: `True` or `False`, by comparing values/variables.

This is useful to build logic, and find answers.

For example, you can use a comparison operator, such as the **greater than** (`>`) operator to find out if an expression (or a variable) is true:

Example

```
int x = 10;

int y = 9;

Console.WriteLine(x > y); // returns True, because 10 is higher than 9
```

Or even easier:

Example

```
Console.WriteLine(10 > 9); // returns True, because 10 is higher than 9
```

In the examples below, we use the **equal to** (`==`) operator to evaluate an expression:

Example

```
int x = 10;

Console.WriteLine(x == 10); // returns True, because the value of x is
equal to 10
```

Example

```
Console.WriteLine(10 == 15); // returns False, because 10 is not equal to
15
```

Real Life Example

Let's think of a "real life example" where we need to find out if a person is old enough to vote.

In the example below, we use the `>=` comparison operator to find out if the age (`25`) is **greater than** OR **equal to** the voting age limit, which is set to `18`:

Example

```
int myAge = 25;

int votingAge = 18;

Console.WriteLine(myAge >= votingAge);
```

Example

Output "Old enough to vote!" if `myAge` is **greater than or equal to** `18`. Otherwise output "Not old enough to vote.":

```
int myAge = 25;
int votingAge = 18;

if (myAge >= votingAge)
{
  Console.WriteLine("Old enough to vote!");
}
else
{
  Console.WriteLine("Not old enough to vote.");
}
```

C# If ... Else

C# Conditions and If Statements

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

---

The if Statement

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
}
```

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");
}
```

C# The else Statement

The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is `False`.

Syntax

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

Example

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

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

C# The else if Statement

The else if Statement

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."
```

C# Switch Statements

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.

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."
```

C# While Loop

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

C# While Loop

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++;

}
```

The Do/While Loop

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);
```

C# For Loop

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);

}
```

Example explained

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.

Another Example

This example will only print even values between 0 and 10:

Example

```
for (int i = 0; i <= 10; i = i + 2)

{

  Console.WriteLine(i);

}
```

Nested Loops

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

```
// Outer loop
for (int i = 1; i <= 2; ++i)
{
  Console.WriteLine("Outer: " + i);  // Executes 2 times

  // Inner loop
  for (int j = 1; j <= 3; j++)
  {
    Console.WriteLine(" Inner: " + j); // Executes 6 times (2 * 3)
  }
}
```

C# Foreach Loop

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);

}
```

C# Break and Continue

C# Break

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example jumps out of the loop when `i` is equal to `4`:

Example

```
for (int i = 0; i < 10; i++)

{

  if (i == 4)

  {

    break;
  }
  Console.WriteLine(i);
}
```

C# Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of `4`:

Example

```
for (int i = 0; i < 10; i++)
{
  if (i == 4)
  {
    continue;
  }
  Console.WriteLine(i);
}
```

Break and Continue in While Loop

You can also use `break` and `continue` in while loops:

Break Example

```
int i = 0;
while (i < 10)
```

```
{
  Console.WriteLine(i);
  i++;
  if (i == 4)
  {
    break;
  }
}
```

Continue Example

```
int i = 0;
while (i < 10)
{
  if (i == 4)
  {
    i++;
    continue;
  }
  Console.WriteLine(i);
  i++;
}
```

C# Arrays

Create an Array

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;
```

We have now declared a variable that holds an array of strings.

To insert values to 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, you could write:

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

Access the Elements of an Array

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
```

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);
}
```

Sort an Array

There are many array methods available, for example `Sort()`, which sorts an array alphabetically or in an ascending order:

Example

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

// Sort an int
int[] myNumbers = {5, 1, 8, 9};
Array.Sort(myNumbers);
foreach (int i in myNumbers)
{
  Console.WriteLine(i);
}
```

System.Linq Namespace

Other useful array methods, such as `Min`, `Max`, and `Sum`, can be found in the `System.Linq` namespace:

Example

```
using System;
using System.Linq;

namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] myNumbers = {5, 1, 8, 9};
      Console.WriteLine(myNumbers.Max());  // returns the largest value
      Console.WriteLine(myNumbers.Min());  // returns the smallest value
      Console.WriteLine(myNumbers.Sum());  // returns the sum of elements
    }
  }
}
```

C# Multidimensional Arrays

Multidimensional Arrays

In the previous chapter, you learned about arrays, which is also known as **single dimension arrays**. These are great, and something you will use a lot while programming in C#. However, if you want to store data as a tabular form, like a table with rows and columns, you need to get familiar with **multidimensional arrays**.

A multidimensional array is basically an array of arrays.

Arrays can have any number of dimensions. The most common are two-dimensional arrays (2D).

Two-Dimensional Arrays

To create a 2D array, add each array within its own set of curly braces, and insert a comma (,) inside the square brackets:

Example

```
int[,] numbers = { {1, 4, 2}, {3, 6, 8} };
```