

---

# Introduction to C#

# What is C# ?

---

- C # is a simple, modern, component-based, object-oriented programming language that resembles C ++ and Java.
- C # is the core language of Microsoft's .NET .
- It is included in the Visual Studio .NET integrated development environment as well as Visual Basic.NET, Visual C ++. Net, Visual J #, and JScript.
- All of these languages provide access to all of the components of the .NET architecture, including the runtime called the Common Language Runtime (CLR) and a comprehensive class library.

# C# & .Net framework

---

- ❑ **C # is the main language of the .NET Framework and it has the following advantages:**
  - C # datatypes exactly match common language runtime (CLR) with no middle layer, so C # may have a small performance advantage;
  - in the Microsoft documentation, many examples are given in C # only;
  - C # has a stricter and more secure syntax, and excludes the use of implicit typing;
  - C # has XML documentation functions built into the language;
  - C # has a more familiar syntax for C, C ++, Java, and Perl programmers.

# Visual Studio.Net environment

---

- Visual Studio .Net is a development environment rich in tools containing all the functionalities necessary for the creation of C #, VB.Net, C ++. Net, J #, JScript projects.

# Visual Studio 2019

## Open recent

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

## Get started



### Clone or check out code

Get code from an online repository like GitHub or Azure DevOps



### Open a project or solution

Open a local Visual Studio project or .sln file



### Open a local folder

Navigate and edit code within any folder



### Create a new project

Choose a project template with code scaffolding to get started

[Continue without code →](#)

# Create a new project

## Recent project templates

A list of your recently accessed templates will be displayed here.

All Languages

All Platforms

All Project Types



### Console App (.NET Core)

New

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

C#

Linux

macOS

Windows

Console



### Console App (.NET Core)

New

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

Visual Basic

Windows

Linux

macOS

Console



### Class Library (.NET Standard)

New

A project for creating a class library that targets .NET Standard.

C#

Android

iOS

Linux

macOS

Windows

Library



### Class Library (.NET Standard)

New

A project for creating a class library that targets .NET Standard.

Visual Basic

Android

iOS

Linux

macOS

Windows

Library



### MSTest Test Project (.NET Core)

New

A project that contains MSTest unit tests that can run on .NET Core on Windows, Linux and MacOS.

C#

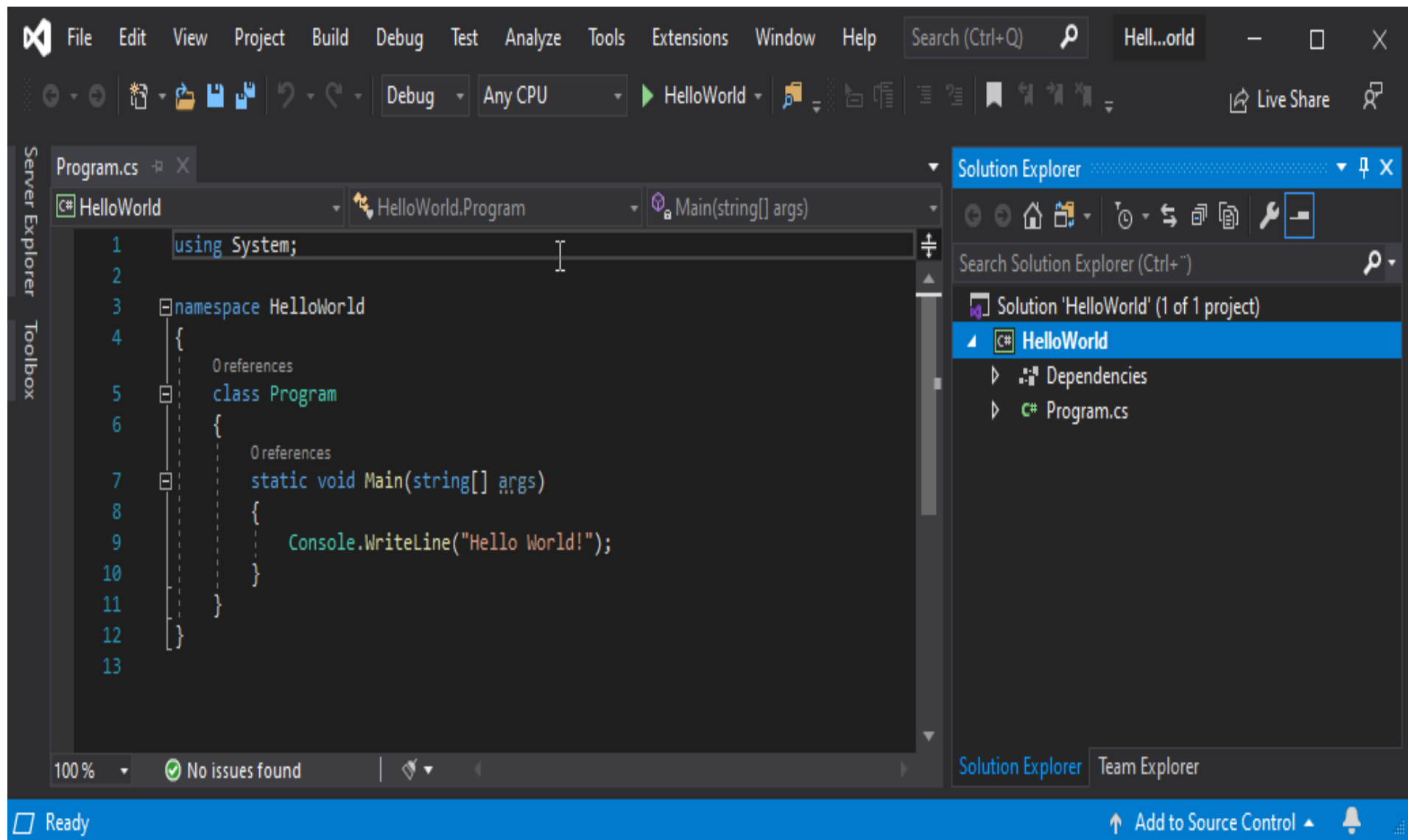
Linux

macOS

Windows

Test

Next



# First application in C#

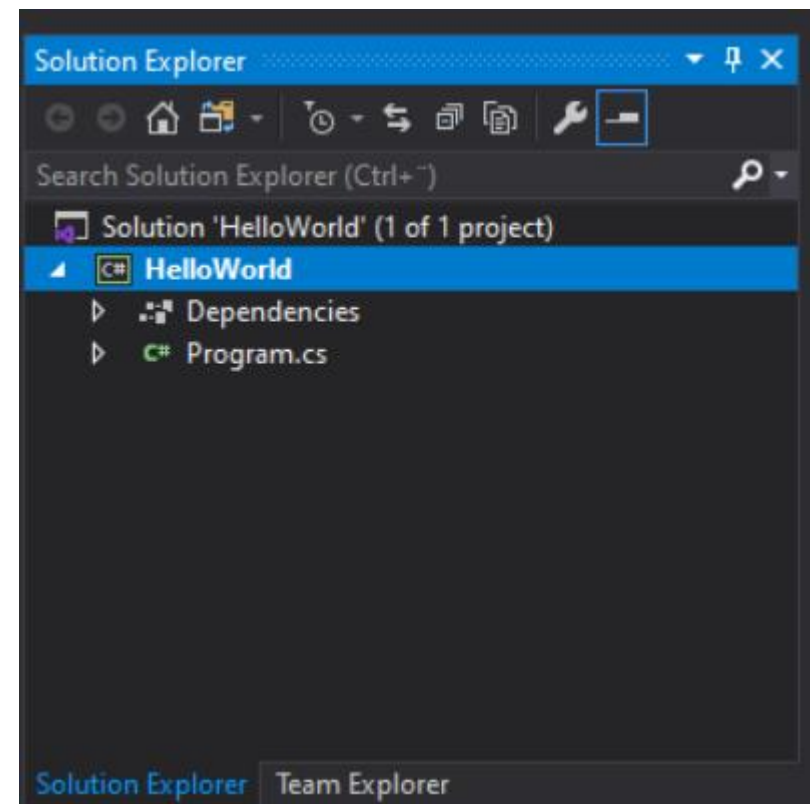
```
1  using System;
2
3
4  namespace HelloWorld
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             Console.WriteLine("Hello World");
11             Console.ReadKey();
12         }
13     }
14 }
15
```



# Solution explorer

---

- The Solution Explorer is a window that shows the files that make up the solution in a tree structure.



# Solution files (1):

---

## ❑ HelloWorld.sln

- This is the first file in the solution.
- There is one per application. All solution file contains one or more project files.
- In the file system, all solution file has the extension .sln.
- In Solution Explorer, it appears a simple name to read, such as "Solution' HelloWolrd "in this project

## Solution files (2):

---

### ❑ HelloWorld.csproj

- This is a C # project file.
- All project file contains one or more source files.
- Source files for the same project must be written in the same programming language.
- This type of file is displayed in Solution Explorer only using the project name but is stored in the system with the extension .csproj.

# Solution files (3):

---

## □ Program.cs

- This is a C # source file, in which you will write your code.
- It contains some code that is automatically generated by Visual Studio.Net and that we will review shortly.

# Solution files (4):

---


## ❑ **AssemblyInfo.cs**

- This is another C # source file.
- This file allows you to add attributes to your program, such as the name of the author, the date the program was written, etc.
- There are other more advanced attributes that can affect the way the program runs.

# Creation of documentation using XML and comments

---

- In Visual C #, it is possible to create documentation for code by including XML tags. in comment fields directly before the block of code to which they refer.



The screenshot shows a C# code file in Visual Studio. On the left, a Solution Explorer pane displays a tree view with a yellow bar next to the 'Program' class and a green bar next to the 'Main' method. The code editor shows the following content:

```
3 namespace HelloWorld
4 {
5     0 references
6     class Program
7     {
8         /// <summary>
9         ///
10        /// </summary>
11        /// <param name="args"></param>
12        0 references
13        static void Main(string[] args)
14        {
15            Console.WriteLine("Hello World");
16            Console.ReadKey();
17        }
18    }
19 }
```

# Creation of documentation using XML and comments

---

- ❑ Here are some examples of tags:
  - **<summary> description</summary>** : used to describe a type.
  - **<remarks>description</remarks>** : Adding the additional information to a type description.
  - **<para>contenu</para>** : being used inside another tag, such as <summary>, <remarks> or <returns>, and allows you to structure the text.

# Main () method

---

- Each executable program has at least one static Main () method in one of its classes.
- This method represents the entry point of the program, that is, the main thread created automatically when the process starts will start by executing the code of this method.
- There can optionally be multiple Main () methods (each in a different class) per program. If so, you must tell the compiler which Main () method is the entry point of the program.



# The comments

---

- ❑ There are three ways to comment out text in a C # source file:
  - Text placed between the `/ *` tags followed by `* /` is commented out. These tags can possibly be on two different lines.
  - If a line contains the tag `//` then the text of this line which follows this tag is commented out.
  - If a line contains the `///` tag then the text of that line which follows this tag is commented out. In addition, this text will be part of the automatic documentation.

# Data types in C# (1)

---

- ❑ There are 6 categories of C # predefined types:
  - **Reference:** object, string
  - **Signed:** sbyte, short, int, long
  - **Unsigned:** byte, ushort, uint, ulong
  - **Character:** char
  - **Floating:** float, double, decimal
  - **Logical:** bool

# Data types in C# (2)

---

- ❑ The predefined types are aliases of the CTS types defined in the .Net framework.
- **Example:** Type `int` is an alias of `System.Int32`
- `Int32` is a structure that belongs to the `System` namespace

# Integer types

| Name   | CTS Type      | Description             | Rang de Valeurs              |
|--------|---------------|-------------------------|------------------------------|
| sbyte  | System.SByte  | 8-bit signed integer    | -128 à 127                   |
| short  | System.Int16  | 16-bit signed integer   | -32768 à 32767               |
| int    | System.Int32  | 32-bit signed integer   | -2147483648 à 2147483647     |
| long   | System.Int64  | 64-bit signed integer   | ( $-2^{63}$ à $2^{63} - 1$ ) |
| byte   | System.Byte   | 8-bit unsigned integer  | 0 à 255                      |
| ushort | System.UInt16 | 16-bit unsigned integer | 0 à 65535                    |
| uint   | System.UInt32 | 32-bit unsigned integer | 0 à 4294967295               |
| ulong  | System.UInt64 | 64-bit unsigned integer | 0 à 18446744073709551615     |

# Date type

| C# Type | .Net Framework (System) type | Signed? | Bytes | Possible values  |
|---------|------------------------------|---------|-------|--|
| sbyte   | System.Sbyte                 | OUI     | 1     | -128 à 127   |
| short   | System.Int16                 | OUI     | 2     | -32768 à 32767   |
| int     | System.Int32                 | OUI     | 4     | -2147483648 à 2147483647   |
| long    | System.Int64                 | OUI     | 8     | -9223372036854775808 à 9223372036854775807                           |
| byte    | System.Byte                  | Non     | 1     | 0 à 255  |
| ushort  | System.UInt16                | Non     | 2     | 0 à 65535  |
| uint    | System.UInt32                | Non     | 4     | 0 à 4294967295   |
| ulong   | System.UInt64                | Non     | 8     | 0 à 18446744073709551615   |
| float   | System.Single                | OUI     | 4     | Approximately $\pm 1.5 \times 10^{-45}$ à $\pm 3.4 \times 10^{38}$   |
| double  | System.Double                | OUI     | 8     | Approximately $\pm 5.0 \times 10^{-324}$ à $\pm 1.7 \times 10^{308}$ |
| decimal | System.Decimal               | OUI     | 12    | Approximately $\pm 1.0 \times 10^{-28}$ à $\pm 7.9 \times 10^{28}$   |
| char    | System.Char                  | –       | 2     | Unicode (16 bit)   |
| bool    | System.Boolean               | –       | 1 / 2 | true or false  |

# Variables & Data types

---

## ❑ Declaration of Variables:

- **Step 1:** Determining the Type of data to use.
- **Step 2:** Creating the name of the Variable.
- **Step 3:** Ending with a semicolon.
- **Example:** int counter; double rate;

## ❑ Initialization of Variables:

- **Step 1:** Using the assignment operator.
- **Step 2:** Determining the value to assign.
- **Step 3:** Ending with a semicolon
- 
- **Example :** int compteur = 10; double taux = 150.59;

# Strings

---

❑ A string can be declared and initialized in several ways:

- `string s = "Hello World";`
- **Result** : Hello World
  
- `string s = "\"Hello\"";`
- **Result**: "Hello"
  
- `string s = "Hello\nWorld";`
- **Result** :   Hello  
                  World
  
- `string s = @"C:\Test\SubDirectory";`
- Exact String

# Constants

---

- ❑ Constants in C # are declared with the keyword `const`
- Example :
- **const** int MAX\_EMPLOYEES = 3000;
- **const** string SQL\_CONNECTION\_STRING = “database=Northwind;user\_id=moi; password = secret”;



# Conversions between numbers and strings

---

- String -> int :  
*int.Parse(string) or Int32.Parse(string)*
- String -> long :  
*long.Parse(string) or Int64.Parse(string)*
- String -> double :  
*double.Parse(string) or Double.Parse(string)*
- String -> float :  
*float.Parse(string) or Float.Parse(string)*

# Error of conversions

---

- Converting a string to a number may fail if the string does not represent a valid number. A fatal error will be generated.
- This error is called an exception in C #.

# Error of conversions

---

- This error can be handled by the following try / catch clause:

```
try{
```

```
//calling the function likely to generate the exception  
}
```

```
catch (Exception e){
```

```
//handling exception e
```

```
}
```

- If the function does not generate an exception, then we go to the next statement, otherwise we go to the body of the catch clause and then to the next statement.

# Type changes

---

- In an expression, it is possible to change the type of a value.
- This is called type casting.
- The syntax for changing the type of a value in an expression is as follows:  
(type) value
- The value then takes the indicated type.

# Arrays

---

- ❑ An array in C # is an object allowing to gather under the same identifier data of the same type

- It's declaration :

*Type[ ] Tableau=new Type[n]*

- n is the number of data that the array can contain.

The syntax Table [I] designates data  $n^{\circ} i$  where i belongs to the interval [0, n-1].

- An array can be initialized at the same time as declared :

`int[ ] integers=new int[ ] {0,10,20,30};`

# Arrays

---

- The Arrays have a Length property which is the number of elements in the array.

- A two-dimensional array can be declared as follows:

`Type[ , ] Tableau=new Type[n,m];`

- where n is the number of rows, m the number of columns.
- The syntax `[i, j]` designates the element j of line i of Table.
- The two-dimensional array can also be initialized at the same time as it is declared:

`double[ , ] réels=new double[ , ] { {0.5, 1.7}, {8.4, -6}};`

# Arrays

---

- The number of elements in each of the dimensions can be obtained by the GetLenth (i) method where i = 0 represents the dimension corresponding to the 1st index, i = 1 the dimension corresponding to the 2nd index,...

An array of arrays is declared as follows:

```
Type[ ][ ]Tableau=new Type[n][ ];
```

- The above declaration creates an array of n rows.
- Each Array [ i ] element is a one-dimensional array reference.

# The operators

---

- ❑ The operators of arithmetic expressions are as follows:
  - **+ Addition**
  - **- Subtraction**
  - **\* Multiplication**
  - **/ division:** The result is the exact quotient if at least one of the operands is real. If both operands are integers the result is the integer quotient.  
So  $5/2 \rightarrow 2$  and  $5.0 / 2 \rightarrow 2.5$
  - **% division:** The result is the remainder whatever the nature of the operands, the quotient being itself integer. It is therefore the modulo operation.



# The operators

---

□ There are various mathematical functions. Here are some of them :

- *double* **Sqrt(double x)**
- *double* **Cos(double x)**
- *double* **Sin(double x)**
- *double* **Tan(double x)**
- *double* **Pow(double x,double y)**
- *double* **Exp(double x)**
- *double* **Log(double x)**
- *double* **Abs(double x)**

# The operators

---

- All of these functions are defined in a C # class called Math. When used, they must be prefixed with the name of the class where they are defined.

So we will write:

```
double x, y=4;
```

```
x=Math.Sqrt(y);
```

# The operators

---

- Priorities in the evaluation of arithmetic expressions:
- The priority of the operators when evaluating an arithmetic expression is as follows (from highest priority to least priority):

*[functions], [ ( )], [ \*, /, %], [ +, - ]*

—————→ **priority**

- Remark : Operators of the same block [ ] have the same priority.

# The operators

---

- Relational operators:  
<, <=, ==, !=, >, > =
- The result of a relational expression is the Boolean.

false if expression false, true otherwise

# The operators

---

## ❑ Comparison of two characters:

- It is possible to compare two characters with relational operators.
- It is then their ASCII codes, which are numbers, which are then compared.
- It is recalled that according to the ASCII order we have the following relations:

Space < .. < '0' < '1' < .. < '9' < .. < 'A' < 'B' < .. < 'Z' < .. < 'a' < 'b' < .. < 'z'

# The operators

---

- ❑ Comparison of strings:
  - They are compared character by character. The first inequality encountered between two characters induces an inequality of the same direction on the strings.

# The operators

---

- Boolean operators:

NOT (!)

AND (&&)

OR (||)

- Relational operators take precedence over && and || operators.

# The operators

---

- ❑ Combination of operators:

$a=a+b$  can be written  $a+=b$

$a=a-b$  can be written  $a-=b$

- ❑ Increment and decrement operators:

The variable notation  $++$  means

$\text{variable}=\text{variable}+1$  or  $\text{variable}+=1$

The variable notation  $--$  means

$\text{variable}=\text{variable}-1$  or  $\text{variable}-=1$ .



# The operators

---

## ❑ The ternary operator « ? »

- The expression `expr_cond? expr1: expr2` is evaluated as follows:
- The `expr_cond` is evaluated. It is a conditional expression with value `true` or `false`
- If `true`, the value of the expression is that of `expr1`. So, `expr2` is not evaluated.
- If it is `false`, the reverse occurs: the value of the expression is that of `expr2`. So, `expr1` is not evaluated.

# The operators

---

## ❑ Example:

- Operation `i=(j>4 ? j+1:j-1);` will assign to variable `i` : `j+1` if `j>4`, `j-1` otherwise.
- It's the same as writing  
    `if(j>4) i=j+1;`  
    `else i=j-1;`  
    but it's more concise

# Control structures

---

- A control structure is a part of the program that changes the default behavior.
- Remember that this default behavior is to execute the instructions one after the other.

# Control structures: The conditions

---

- ❑ There are three types of conditions:
  - if / else
  - switch
  - The ternary operator ?:

# Control structures: The conditions

---

❑ C # has the following conditional statements:

- Use if to specify a block of code to execute, if a specified condition is true
- Use else to specify a block of code to execute, 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 alternate code blocks to run

# if-else

---

```
if (boolean)
{ ... }
else if (boolean)
{ }
else
{ ...}
```

# Switch

---

- The switch expression is evaluated once
- The value of the expression is compared to the values of each.
- If there is a match, the associated code block is executed
- When C # hits a keyword break, it exits the switch block.
- This will stop the execution of other code and case tests inside the block.
- When a match is found and the job is done, it's time to take a break. There is no need for further testing.

# Switch

---

```
switch(expression) {  
    case constant-expression1 :  
        statement(s) ;  
        break;  
    case constant-expression2 :  
    case constant-expression3 :  
        statement(s) ;  
        break;  
  
    /* you can have any number of case statements */  
    default : /* Optional */  
        statement(s) ;  
}
```



# Control structures: Loops

---

□ There are four types of loops:

- while
- do / while
- for
- foreach

# For

---

```
for ( initialisation; condition; modification) {  
    Instruction1 ...  
    Instruction 2 ...  
}
```

# while

---

```
while (condition:boolean)
{... // code to be executed in the loop
}
//The code is executed as long as the boolean is true
```

# do-while

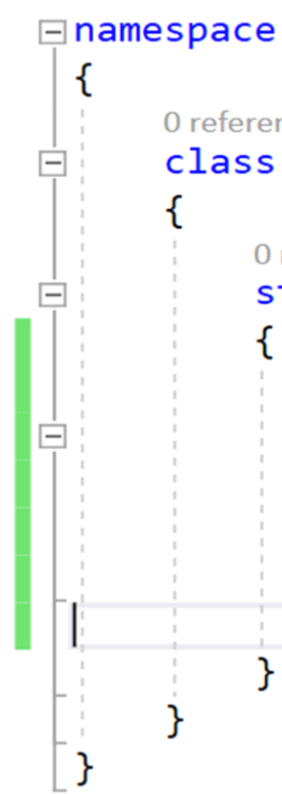
---

```
do
{... // code to be executed in the loop
} while (condition: boolean);

//Executed at least once regardless of the value of the boolean
```

# Example

```
1  using System;
2
3  namespace Foreach
4  {
5      0 references
6      class Program
7      {
8          0 references
9          static void Main(string[] args)
10         {
11             int i = 1;
12             do
13             {
14                 Console.WriteLine(i);
15             } while (i > 1);
16         }
17     }
18
```



# foreach

---


```
string[] tab = new string[5];
tab[0] = "string 1";
...
tab[4] = "string 5";

// Iteration over all the data in container
foreach (string s in tab)
{
    // Code executed for each value contained in array(tab)
    // The value is stored in the variable s
    // declared in the foreach statement

    Console.WriteLine(s);
}
```

# Example

```
1      using System;
2
3      namespace Foreach
4      {
5          0 references
6          class Program
7          {
8              0 references
9              static void Main(string[] args)
10             {
11                 int[] arr = { 1, 3, 4, 8, 2 };
12
13                 // The sum of elements in array.
14                 int sum = 0;
15                 foreach (int i in arr)
16                     sum += i;
17                 // Sum is : 1+3+4+8+2 = 18
18                 Console.WriteLine(sum);
19             }
20         }
21     }
```



# Exceptions

---

- .NET error handling is based on exceptions. Exceptions must all derive from the base `System.Exception`.
- `Exceptions` contain information about the error and its context
- The syntax used is similar to C++.
- Availability of a `finally` block for the release of resources



# Exceptions

---

```
try
{
    // Launching the exception :
    throw new IOException("not found");
}
// Retrieving the exception :
catch (IOException e)
{
    Console.WriteLine(e.Message);
    // Relaunching the exception : throw;
}
catch
{
    // Treatment of all other exceptions
}
```

# Exceptions

---

```
try
{
    // Code executed under the control of the try.
}
catch
{
    // Retrieve exceptions
}
finally
{
    // End of try block processing.
    // Executed whether there was an exception or not.
    // Attention, is not executed if the exception is
    // not processed by any catch block!
}
```