C# - Crash Course

1. Hello World

```
// Hello World! program
namespace HelloWorld
{
    class Hello {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello World!");
        }
    }
}
```

Let's break down the program line by line.

1. `// Hello World! Program`

   `//` indicates the beginning of a comment in C#. Comments are not executed by the C# compiler.

   They are intended for the developers to better understand a piece of code. To learn more about comments in C#, visit *C# comments*.

2. `namespace HelloWorld{...}`

   The namespace keyword is used to define our own namespace. Here we are creating a namespace called `HelloWorld`.

Just think of namespace as a container which consists of classes, methods and other namespaces.

3. `class Hello{...}`

   The above statement creates a class named – `Hello` in C#. Since, C# is an object-oriented programming language, creating a class is mandatory for the program’s execution.

4. `static void Main(string[] args){...}`

   `Main()` is a method of class Hello. The execution of every C# program starts from the `Main()` method. So it is mandatory for a C# program to have a `Main()` method.

5. `System.Console.WriteLine("Hello World!");`

   For now, just remember that this is the piece of code that prints **Hello World!** to the output screen

2. Keywords and Identifiers

C# Keywords

Keywords are predefined sets of reserved words that have special meaning in a program. The meaning of keywords can not be changed, neither can they be directly used as identifiers in a program.

```
int @void;
```

The above statement will create a variable `@void` of type `int`.

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	in (generic modifier)	int	interface
internal	is	lock	long
namespace	new	null	object
operator	out	out (generic modifier)	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	string	struct	switch
this	throw	true	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	using static
void	volatile	while	

C# Identifiers

Identifiers are the name given to entities such as variables, methods, classes, etc. They are tokens in a program which uniquely identify an element.

```
int value;
```

Here, `value` is the name of variable.

3. Variables

A variable is a symbolic name given to a memory location. Variables are used to store data in a computer program.

```
int age = 24;
```

In this example, a variable `age` of type `int` (integer) is declared and initialized to some value during declaration.
Since, it’s a variable, we can change the value of variables as well. For example,

```
int age = 24;

age = 35;
```

Implicitly typed variables:

Alternatively in C#, we can declare a variable without knowing its type using `var` keyword.

```
var value = 5;
```

The compiler determines the type of variable from the value that is assigned to the variable. In the above example, `value` is of type `int.`

Primitive Data Types:

Variables in C# are broadly classified into two types: **Value types** and **Reference types**. In this tutorial we will be discussing about primitive (simple) data types which is a subclass of Value types.

1. Boolean (bool)

- Boolean data type has two possible values: `true` or `false`
- **Default value**: `false`
- Boolean variables are generally used to check conditions such as in *if statements*, *loops*, etc.

For Example:

```
using System;
namespace DataType
{
    class BooleanExample
    {
        public static void Main(string[] args)
        {
            bool isValid = true;
            Console.WriteLine(isValid);
        }
    }
}
```

When we run the program, the output will be:

```
True
```

2. short

- **Size**: 16 bits
- **Range**: -32,768 to 32,767
- **Default value**: 0

For example:

```
using System;
namespace DataType
{
    class ShortExample
    {
        public static void Main(string[] args)
        {
            short value = -1109;
            Console.WriteLine(value);
        }
    }
}
```

When we run the program, the output will be:

```
-1109
```

3. int

- **Size**: 32 bits
- **Range**: -231 to 231-1
- **Default value**: 0

For example:

```
using System;
namespace DataType
{
    class IntExample
    {
        public static void Main(string[] args)
        {
            int score = 51092;
            Console.WriteLine(score);
        }
    }
}
```

When we run the program, the output will be:

```
51092
```

4. long

- **Size**: 64 bits
- **Range**: -263 to 263-1
- **Default value**: `0L` [L at the end represent the value is of long type]

For example:

```
using System;
namespace DataType
{
    class LongExample
    {
        public static void Main(string[] args)
        {
            long range = -7091821871L;
            Console.WriteLine(range);
        }
    }
}
```

When we run the program, the output will be:

```
-7091821871
```

5. byte

- **Size**: 8 bits
- **Range**: 0 to 255.
- **Default value**: 0

For example:

```
using System;
namespace DataType
{
    class ByteExample
    {
        public static void Main(string[] args)
        {
            byte age = 62;
            Console.WriteLine(level);
        }
    }
}
```

When we run the program, the output will be:

```
62
```

6. float

- Single-precision floating point type
- **Size**: 32 bits
- **Range**: 1.5 × 10–45 to 3.4 × 1038
- **Default value**: 0.0F [F at the end represent the value is of float type]

For example:

```
using System;
namespace DataType
{
    class FloatExample
    {
        public static void Main(string[] args)
        {
            float number = 43.27F;
            Console.WriteLine(number);
        }
    }
}
```

When we run the program, the output will be:

```
43.27
```

7. double

- Double-precision floating point type. What is the difference between single and double precision floating point?
- **Size**: 64 bits
- **Range**: 5.0 × 10–324 to 1.7 × 10308
- **Default value**: 0.0D [D at the end represent the value is of double type]

For example:

```
using System;
namespace DataType
{
    class DoubleExample
    {
        public static void Main(string[] args)
        {
            double value = -11092.53D;
            Console.WriteLine(value);
        }
    }
}
```

When we run the program, the output will be:

```
-11092.53
```

8. Character (char)

- It represents a 16 bit unicode character.
- **Size**: 16 bits
- **Default value**: '\0'

```
using System;
namespace DataType
{
    class CharExample
    {
        public static void Main(string[] args)
        {
            char ch1 ='\u0042';
            char ch2 = 'x';
            Console.WriteLine(ch1);
            Console.WriteLine(ch2);
        }
    }
}
```

When we run the program, the output will be:

```
B
x
```

The unicode value of `'B'` is `'\u0042'`, hence printing `ch1` will print `'B'`.

Literals

Let's look at the following statement:

```
int number = 41;
```

Here,

- `int` is a data type
- `number` is a variable and
- `41` is a literal

4. Operators

Operators are symbols that are used to perform operations on operands. Operands may be variables and/or constants.

1. Basic Assignment Operator

Basic assignment operator (=) is used to assign values to variables. For example,

```
double x;

x = 50.05;
```

Here, 50.05 is assigned to x

2. Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, multiplication, division, etc.

For example,

```
int x = 5;

int y = 10;

int z = x + y;// z = 15
```

C# Arithmetic Operators

Operator	Operator Name	Example
+	Addition Operator	6 + 3 evaluates to 9
-	Subtraction Operator	10 - 6 evaluates to 4
*	Multiplication Operator	4 * 2 evaluates to 8
/	Division Operator	10 / 5 evaluates to 2
%	Modulo Operator (Remainder)	16 % 3 evaluates to 1

3. Relational Operators

Relational operators are used to check the relationship between two operands. If the relationship is true the result will be `true`, otherwise it will result in `false`.
Relational operators are used in decision making and loops.

C# Relational Operators

Operator	Operator Name	Example
==	Equal to	6 == 4 evaluates to false
>	Greater than	3 > -1 evaluates to true
<	Less than	5 < 3 evaluates to false
>=	Greater than or equal to	4 >= 4 evaluates to true
<=	Less than or equal to	5 <= 3 evaluates to false
!=	Not equal to	10 != 2 evaluates to true

4. Logical Operators

Logical operators are used to perform logical operation such as `and`, `or`. Logical operators operates on boolean expressions (`true` and `false`) and returns boolean values. Logical operators are used in decision making and loops.

Here is how the result is evaluated for logical `AND` and `OR` operators.

C# Logical operators

Operand 1	Operand 2	OR (||)	AND (&&)
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

In simple words, the table can be summarized as:

- If one of the operand is true, the `OR` operator will evaluate it to `true`.
- If one of the operand is false, the `AND` operator will evaluate it to `false`.

5. Unary Operators

Unlike other operators, the unary operators operates on a single operand.

C# unary operators

Operator	Operator Name	Description
+	Unary Plus	Leaves the sign of operand as it is
-	Unary Minus	Inverts the sign of operand
++	Increment	Increment value by 1
--	Decrement	Decrement value by 1
!	Logical Negation (Not)	Inverts the value of a boolean

6. Ternary Operator

The ternary operator `? :` operates on three operands. It is a shorthand for `if-then-else` statement. Ternary operator can be used as follows:

```
variable = Condition? Expression1 : Expression2;
```

The ternary operator works as follows:

- If the expression stated by Condition is `true`, the result of `Expression1` is assigned to variable.
- If it is `false`, the result of `Expression2` is assigned to variable.

7. Bitwise and Bit Shift Operators

Bitwise and bit shift operators are used to perform bit manipulation operations.

C# Bitwise and Bit Shift operators

Operator	Operator Name
~	Bitwise Complement
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
<<	Bitwise Left Shift
>>	Bitwise Right Shift

8. Compound Assignment Operators

C# Compound Assignment Operators

Operator	Operator Name	Example	Equivalent To
+=	Addition Assignment	`x += 5`	`x = x + 5`

C# Compound Assignment Operators

Operator	Operator Name	Example	Equivalent To
-=	Subtraction Assignment	`x -= 5`	`x = x - 5`
*=	Multiplication Assignment	`x *= 5`	`x = x * 5`
/=	Division Assignment	`x /= 5`	`x = x / 5`
%=	Modulo Assignment	`x %= 5`	`x = x % 5`
&=	Bitwise AND Assignment	`x &= 5`	`x = x & 5`
|=	Bitwise OR Assignment	`x |= 5`	`x = x | 5`
^=	Bitwise XOR Assignment	`x ^= 5`	`x = x ^ 5`
<<=	Left Shift Assignment	`x <<= 5`	`x = x << 5`
>>=	Right Shift Assignment	`x >>= 5`	`x = x >> 5`
=>	Lambda Operator	`x => x*x`	`Returns` `x*x`

5. Basic Input and Output

C# Output

In order to output something in C#, we can use

```
System.Console.WriteLine() // prints on next line

System.Console.Write()    // prints on same line
```

Here, `System` is a namespace, `Console` is a class within namespace `System` and `WriteLine` and `Write` are methods of class `Console`.

C# Input

In C#, the simplest method to get input from the user is by using the `ReadLine()` method of the `Console` class.
However, `Read()` and `ReadKey()` are also available for getting input from the user.

```
Console.Write("Enter integer value: ");
userInput = Console.ReadLine();
/* Converts to integer type */
intVal = Convert.ToInt32(userInput);
Console.WriteLine("You entered {0}",intVal)
```

```
Enter integer value: 101
You entered 101
```

6. Expressions, Statements and Blocks

Expressions, statements and blocks are the building block of a C# program.

Expressions

An expression in C# is a combination of operands (variables, literals, method calls) and operators that can be evaluated to a single value. To be precise, an expression must have at least one operand but may not have any operator.

Let's look at the example below:

```
double temperature;

temperature = 42.05;
```

Here, `42.05` is an expression. Also, `temperature = 42.05` is an expression too.

Statements

A statement is a basic unit of execution of a program. A program consists of multiple statements.

For example:

```
int age = 21;

Int marks = 90;
```

In the above example, both lines above are statements.

Blocks

A block is a combination of zero or more statements that is enclosed inside curly brackets { }.

7. Comments

Comments are used in a program to help us understand a piece of code. They are human readable words intended to make the code readable. Comments are completely ignored by the compiler.

Single Line Comments

Single line comments start with a double slash `//`. The compiler ignores everything after `//` to the end of the line.

```
int a = 5 + 7; // Adding 5 and 7
```

Here, `Adding 5 and 7` is the comment.

Multi Line Comments

Multi line comments start with `/*` and ends with `*/`. Multi line comments can span over multiple lines.

XML Documentation Comments

XML documentation comment is a special feature in C#. It starts with a triple slash `///` and is used to categorically describe a piece of code. This is done using XML tags within a comment. These comments are then, used to create a separate XML documentation file.

}

8. if...else Statement

The if statement in C# may have an optional else statement. The block of code inside the else statement will be executed if the expression is evaluated to `false`.

The syntax of if...else statement in C# is:

```
if (boolean-expression)
{
       // statements executed if boolean-expression is true
}
else
{
       // statements executed if boolean-expression is false
}
```

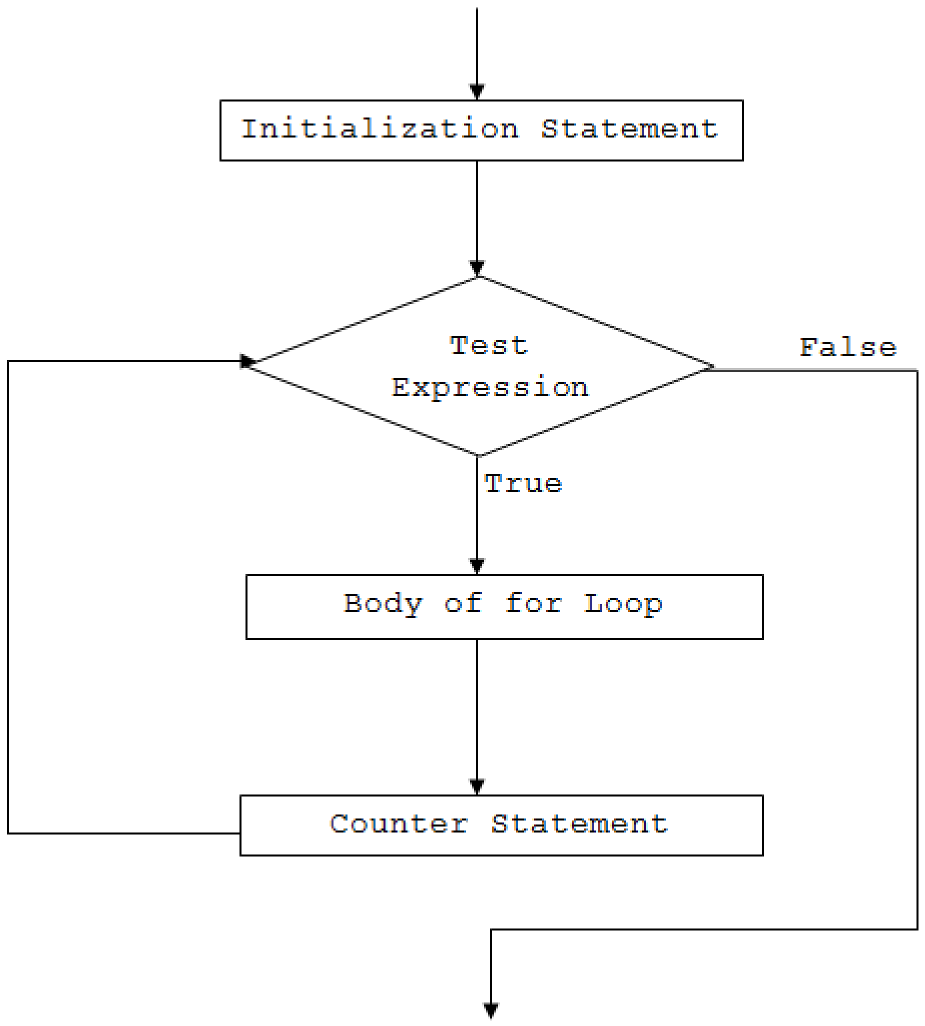
9. for loop

The **for** keyword is used to create for loop in C#.

The syntax of **for loop** is:

```
for (initialization; condition; iterator)
{
       // body of for loop
}
```

Flowchart:



10. while loop

The **while** keyword is used to create while loop in C#.

The syntax for while loop is:

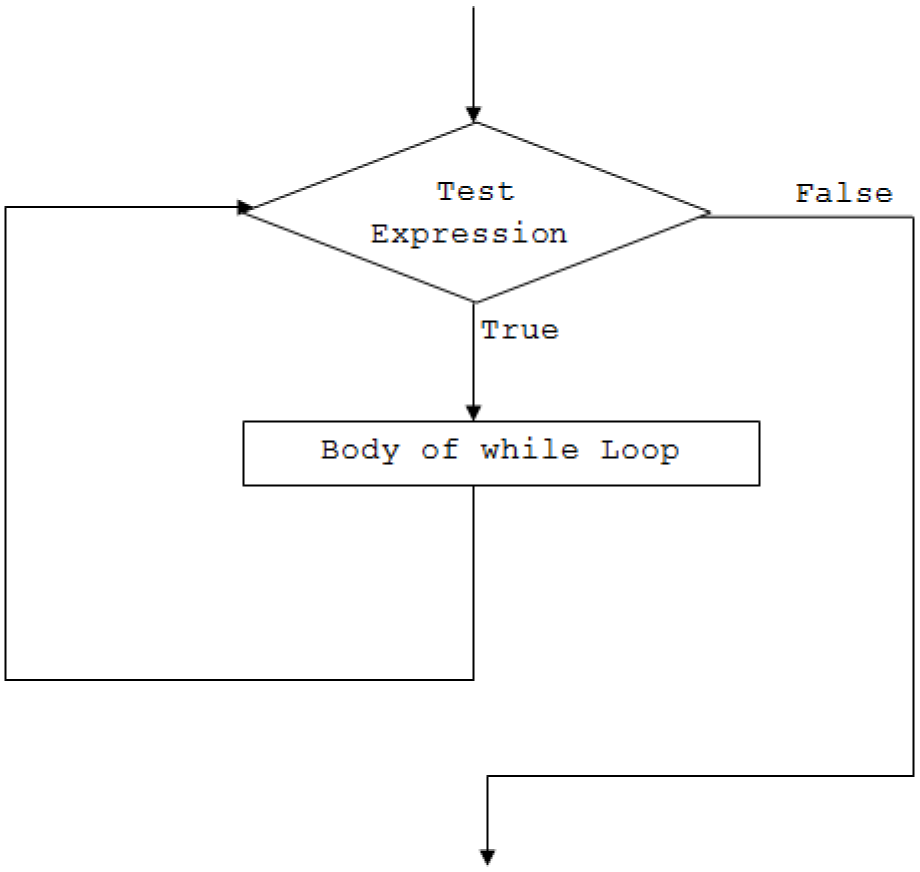
```
while (test-expression)

{

        // body of while

}
```

Flowchart:



11. do...while loop

The **do** and **while** keyword is used to create a do...while loop. It is similar to a while loop, however there is a major difference between them.
In while loop, the condition is checked before the body is executed. It is the exact opposite in do...while loop, i.e. condition is checked after the body is executed.

This is why, the body of do...while loop will execute at least once irrespective to the test-expression.

The syntax for do...while loop is:

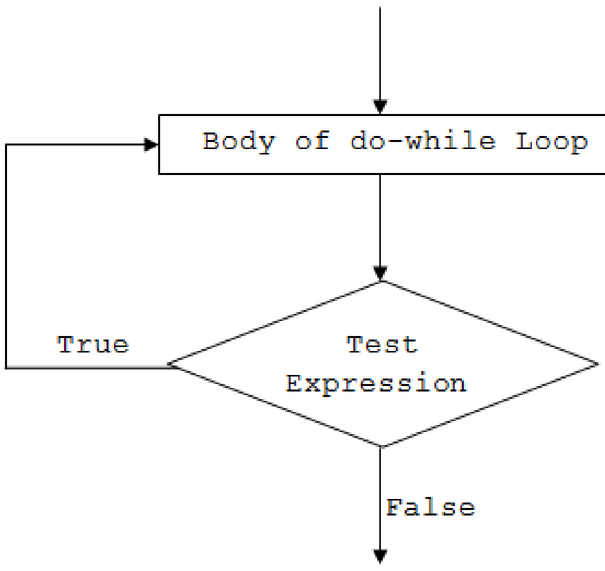
```
do

{

       // body of do while loop

} while (test-expression);
```

Flowchart:



12. For-Each Loop

The foreach loop iterates through each item, hence called foreach loop.

Syntax of foreach loop:

```
foreach (element in iterable-item)

{

    // body of foreach loop

}
```

13. Switch Statement

Switch statement can be used to replace the if...else if statement in C#. The advantage of using switch over if...else if statement is the codes will look much cleaner and readable with switch.
The syntax of switch statement is:

```
switch (variable/expression)

{

    case value1:

        // Statements executed if expression(or variable) = value1

        break;

    case value2:

        // Statements executed if expression(or variable) = value1

        break;

    ... ... ...

    ... ... ...

    default:

        // Statements executed if no case matches

}
```

The switch statement evaluates the `expression` (or `variable`) and compare its value with the values (or expression) of each case (`value1`, `value2`, …). When it finds the matching value, the statements inside that case are executed.

But, if none of the above cases matches the expression, the statements inside `default` block is executed.

However a problem with the switch statement is, when the matching value is found, it executes all statements after it until the end of switch block.

To avoid this, we use `break` statement at the end of each case. The break statement stops the program from executing non-matching statements by terminating the execution of switch statement.

14. Ternary Operator

Ternary operator are a substitute for if...else statement. So before you move any further in this tutorial, go through C# if...else statement (if you haven't).

The syntax of ternary operator is:

```
Condition ? Expression1 : Expression2;
```

The ternary operator works as follows:

- If the expression stated by `Condition` is `true`, the result of `Expression1` is returned by the ternary operator.
- If it is `false`, the result of `Expression2` is returned.

15. BitWise Operator

Bitwise and bit shift operators are used to perform bit level operations on integer (int, long, etc) and boolean data. These operators are not commonly used in real life situations.

Bitwise OR

Bitwise OR operator is represented by `|`. It performs bitwise OR operation on the corresponding bits of two operands. If either of the bits is `1`, the result is `1`. Otherwise the result is `0`.
If the operands are of type `bool`, the bitwise OR operation is equivalent to logical OR operation between them.

Bitwise AND

Bitwise AND operator is represented by `&`. It performs bitwise AND operation on the corresponding bits of two operands. If either of the bits is `0`, the result is `0`. Otherwise the result is `1`.
If the operands are of type `bool`, the bitwise AND operation is equivalent to logical AND operation between them.

Bitwise XOR

Bitwise XOR operator is represented by `^`. It performs bitwise XOR operation on the corresponding bits of two operands. If the corresponding bits are **same**, the result is `0`. If the corresponding bits are **different**, the result is `1`.
If the operands are of type `bool`, the bitwise XOR operation is equivalent to logical XOR operation between them.

Bitwise Complement

Bitwise Complement operator is represented by `~`. It is a unary operator, i.e. operates on only one operand.
The `~` operator **inverts** each bits i.e. changes 1 to 0 and 0 to 1.

Bitwise Left Shift

Bitwise left shift operator is represented by `<<`. The `<<` operator shifts a number to the left by a specified number of bits. Zeroes are added to the least significant bits.
In decimal, it is equivalent to

```
num * 2bits
```

Bitwise Right Shift

Bitwise left shift operator is represented by `>>`. The `>>` operator shifts a number to the right by a specified number of bits. The first operand is shifted to right by the number of bits specified by second operand.
In decimal, it is equivalent to

```
floor(num / 2bits)
```

16. Pre-Processor Directive

Preprocessor directives are a block of statements that gets processed before the actual compilation starts. C# preprocessor directives are the commands for the compiler that affects the compilation process.

These commands specifies which sections of the code to compile or how to handle specific errors and warnings.

C# preprocessor directive begins with a `# (hash)` symbol and all preprocessor directives last for one line. Preprocessor directives are terminated by `new line` rather than `semicolon`.

#define directive

- The `#define` directive allows us to define a symbol.
- Symbols that are defined when used along with `#if` directive will evaluate to true.
- These symbols can be used to specify conditions for compilation.
- **Syntax:**

```
#define SYMBOL
```

#undef directive

- The `#undef` directive allows us to undefine a symbol.
- Undefined symbols when used along with `#if` directive will evaluate to false.
- **Syntax:**

```
#undef SYMBOL
```

#if directive

- The `#if` directive are used to test the preprocessor expression.
- `#if` directive is followed by an `#endif` directive.
- The codes inside the `#if` directive is compiled only if the expression tested with `#if` evaluates to true.

- **Syntax:**

```
#if preprocessor-expression
        code to compile<
#endif
```

#elif directive

- The `#elif` directive is used along with #if directive that lets us create a compound conditional directive.
- It is used when testing multiple preprocessor expression.
- The codes inside the `#elif` directive is compiled only if the expression tested with that `#elif` evaluates to true.
- **Syntax:**

```
#if preprocessor-expression-1

        code to compile

#elif preprocessor-expression-2

        code-to-compile

#endif
```

#else directive

- The `#else` directive is used along with `#if` directive.
- If none of the expression in the preceding `#if` and `#elif` (if present) directives are true, the codes inside the `#else` directive will be compiled.

- **Syntax:**

```
#if preprocessor-expression-1

        code to compile

#elif preprocessor-expression-2

        code-to-compile

#else

        code-to-compile

#endif
```

#endif directive

- The `#endif` directive is used along with `#if` directive to indicate the end of `#if` directive.
- **Syntax:**

```
#if preprocessor-expression-1

        code to compile

#endif
```

#warning directive

- The `#warning` directive allows us to generate a user-defined level one warning from our code.

- **Syntax:**

```
#warning warning-message
```

#error directive

- The `#error` directive allows us to generate a user-defined error from our code.
- **Syntax:**

```
#error error-message
```

#line directive

- The `#line` directive allows us to modify the line number and the filename for errors and warnings.
- **Syntax:**

```
#line line-number file-name
```

#region and #endregion directive

- The `#region` directive allows us to create a region that can be expanded or collapsed when using a Visual Studio Code Editor.
- This directive is simply used to organize the code.
- The #region block can not overlap with a `#if` block. However, a `#region` block can be included within a `#if` block and a `#if` block can overlap with a `#region` block.
- `#endregion` directive indicates the end of a `#region` block.

- **Syntax:**

```
#region region-description

        codes

#endregion
```

#pragma directive

- The `#pragma` directive is used to give the compiler some special instructions for the compilation of the file in which it appears.
- The instruction may include disabling or enabling some warnings.
- C# supports two `#pragma` instructions:
  - `#pragma warning`: Used for disabling or enabling warnings
  - `#pragma checksum`: It generates checksums for source files which will be used for debugging.
- **Syntax:**

```
#pragma pragma-name pragma-arguments
```

17. Namespaces

We can define a namespace in C# using the *namespace* keyword as:

```
namespace Namespace-Name

{

    //Body of namespace

}
```