Data Types:

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

When you want to define a variable in C# the structure of definition as follows:

dataType variableName = value;

For example:

int numberOfFeet: 4;

string name = “Amine”;

double myDoubleNum = 5.99D;

char myLetter = 'D';

bool myBool = true;

The general rules for naming variables are:

Names can contain letters, digits and the underscore character (_)

Names must begin with a letter or underscore

Names should start with a lowercase letter, and 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

Const

If you don't want others (or yourself) to overwrite existing values, you can add the const keyword in front of the variable type.

This will declare the variable as "constant", which means unchangeable and read-only:

```
const int myNum = 15;

myNum = 20; // error
```

But without const keyword:

Int myNumb = 25;

myNumb = 22; // no error

Identifier

When you want to identify some variable you have to use = sign.

If-Else

If else blocks are used to check states and conditions. For example, you want to compare two different variable that find which one is greater than. You can use if else blocks:

```
int firstNum = 5;
int secondNum = 6;

if(a > b) {
    Console.WriteLine("firstNum is greater than secondNum!!!");
}else{
    Console.WriteLine("secondNum is greater than firstNum!!!");
}
```

- 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`

NOTE: To identify we use = sign but to compare we use == sign. So, don't confuse these two operations with each other.

Explanation of If-Else Logic:

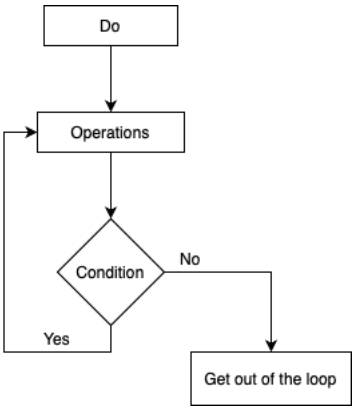
if (condition): If the expression within the if block is true, the code inside this block is executed. If it is not true, the expression of the else if block is checked. When the else if block is also not true, the code inside the else block is executed.

```
int x = 10;

if (x > 10)
{
    Console.WriteLine("x is greater than 10");
}
else if (x == 10)
{
    Console.WriteLine("x is equal to 10");
}
else
{
    Console.WriteLine("x is less than 10");
}
```

Do While Loop

The C# Do While Loop is a system that allows our programs to run at least once regardless of the conditional structure we define, and then aims to repeat the process by checking the conditional structure.



This means that the loop will run at least once even if the condition is not met.

For example:

```
do{
   Console.WriteLine("Do While Loop");
} while (condition);
```

If condition is not true, output will be: Do While Loop
If condition is true, output will be: (infinite times) Do While Loop

While Loop

The C# While loop first checks the (condition) expression. If the expression returns True, the commands in the loop blocks are executed. And after the last command, the While condition is checked again. The loop commands continue to be executed until this condition is False.

For example:

```
int say = 1;
while (say <= 10) {
 Console.WriteLine(say);
 say++;
}
```

The variable say is 1 in the first case. Since it is less than 10, 1 is printed on the screen and the value of the variable say is increased by 1. The new value of the variable say is 2 and since it is less than 10, 2 is printed on the screen. This continues until the value of the variable say is 11. Because the condition is true when count is less than or equal to 10.

For Loop

```
For(initial value; condition; increment){

        Code Blocks;

}
```

In programming language, loops are structures that allow specified command lines to be executed repeatedly. For loops are usually used with three statements. In the first of these statements, an initial value is given for the loop variable. In the second statement, a condition is set depending on the value of the loop variable, and as long as this condition is met, the loop repeats the specified operations in the loop. The third expression is usually used for the amount of increase or decrease of the loop variable.

For example:

```
for (int i = 1; i <= 5; i++) {

    Console.Write(i);

}
```

Output: 12345

In the first case, the value of variable i is 1. As long as i is less than 5, the value of i is printed on the screen and the value of variable i is incremented by one in each loop.

i++ (Post-increment):

This operator first uses the current value of i and then increments i by one.

```
int i = 0;

for (int j = 0; j < 5; j++){

    Console.Write(i++);

}
```

Output: 01234

++i (Pre-increment):

This operator first increments i by one and then uses the new value.

```
int i = 0;

 for (int j = 0; j < 5; j++){

     Console.Write(++i);

 }
```

Output: 12345

i-- (Post-decrement):

This operator first uses the current value of i and then decrements i by one.

```
int i = 5;

 for (int j = 0; j < 5; j++){

      Console.Write(i--);

}
```

Output: 54321

--i (Pre-decrement):

This operator first decrements i by one and then uses the new value.

```
int i = 5;

 for (int j = 0; j < 5; j++){

      Console.Write(--i);

}
```

Output: 43210

Switch Case:

The switch statement in C# is used to check a specific value and execute different code blocks depending on different cases. It's commonly used for multiple selections.

Here's how the switch statement works:The switch statement takes a key value and checks which case it matches. Each case statement checks for a specific value, and if the key value matches, it executes the code block of that case. If none of the case statements match, the default case (which is optional) is executed. The break statement is used at the end of each case block to exit the switch statement. If break is not used, the flow of code continues to the next case.

Here's an 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;
    default:
        Console.WriteLine("Invalid day number");
        break;
```

Output: Thursday

In this example, the switch statement checks the value of the day variable. If day is 1, "Monday" is printed; if it's 2, "Tuesday" is printed, and so on. If none of the cases match, the default block is executed.

Arrays

In C#, arrays are data structures used to store multiple data of the same type. Each element can be accessed by an index number. To define an array, we use the type name followed by [].

If the array size is specified, the size is specified; otherwise, the array size can be defined dynamically.
int[] numbers = new int[5]; // Creates an array of integers with 5 elements.

We use the index number to assign values to the array. The index number starts at 0.

numbers[0] = 10;

We can use a loop to print the array contents.

```
for (int i = 0; i < numbers.Length; i++)

{

    Console.WriteLine(numbers[i]);

}
```

Instead of specifying the size of the array, it can be defined by the initial elements of the array.

int[] dynamicNumbers = { 10, 20, 30, 40, 50 };

Multidimensional Arrays

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.

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

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

The single comma [,] specifies that the array is two-dimensional. A three-dimensional array would have two commas: int[,,].  ,
```

numbers is now an array with two arrays as its elements. The first array element contains three elements: 1, 4 and 2, while the second array element contains 3, 6 and 8. To visualize it, think of the array as a table with rows and columns:

	COLUMN 0	COLUMN 1	COLUMN 2
ROW 0	1	4	2
ROW 1	3	6	8

Access Elements of a 2D Array

To access an element of a two-dimensional array, you must specify two indexes: one for the array, and one for the element inside that array. Or better yet, with the table visualization in mind; one for the row and one for the column (see example below).

This statement accesses the value of the element in the first row (0) and third column (2) of the numbers array:

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

Console.WriteLine(numbers[0, 2]);  // Outputs 2
```

You can also change the value of an element. The following example will change the value of the element in the first row (0) and first column (0):

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

numbers[0, 0] = 5;  // Change value to 5

Console.WriteLine(numbers[0, 0]); // Outputs 5 instead of 1
```

Note that we have to use GetLength() instead of Length to specify how many times the loop should run:

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


for (int i = 0; i < numbers.GetLength(0); i++)

{
```

```
    for (int j = 0; j < numbers.GetLength(1); j++)
    {
        Console.WriteLine(numbers[i, j]);
    }
}
```

List

Lists in C# are dynamic-sized collections of elements. They are similar to arrays in that they can store multiple elements of the same type, but unlike arrays, the size of a list can dynamically grow or shrink.

List Definition: To define a list, we use the List<T> class, where T specifies the type of elements to be stored in the list.

List<int> numbers = new List<int>();

Adding Elements to the List: We can add elements to the list using the Add() method.

numbers.Add(30);

Getting the Number of Elements in the List: The Count property retrieves the number of elements in the list.

int count = numbers.Count; // Getting the number of elements in the list

Acessing Elements of the List: We can access elements of the list similar to arrays, using indexing.

int firstItem = numbers[0];

Printing the Content of the List: We can print the content of the list using a loop or the foreach loop.

foreach (int num in numbers){

    Console.WriteLine(num);

}

Foreach Loop

The foreach loop in C# is used to iterate over elements of a collection (such as an array, list, etc.) without needing to know the size or index of the collection. It simplifies the process of iterating through all elements of the collection one by one.

foreach (type variableName in collection){

```
    // Statements to be executed for each element

}
```

- type specifies the type of elements in the collection.
- variableName is the name of the variable to which each element of the collection will be assigned during each iteration.
- collection is the collection over which iteration will occur.
- The foreach loop applies the statements within its body to each element of the collection, one element at a time, until it has processed all elements in the collection.

Here's an example demonstrating how to use the foreach loop to iterate over the elements of an array and print each element:

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

// Printing array elements

      foreach (int num in numbers){

         Console.WriteLine(num);

      }
```

In this example, the foreach loop iterates over each element (num) in the numbers array, and for each iteration, it prints the value of num. The loop continues until all elements of the array have been processed.