

Chapter 1. C# Introduction

What is C#?

C# is pronounced "C-Sharp".

It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.

C# has roots from the C family, and the language is close to other popular languages like [C++](#) and [Java](#).

The first version was released in year 2002. The latest version, **C# 11**, was released in November 2022.

C# is used for:

- Mobile applications
- Desktop applications
- Web applications
- Web services
- Web sites
- Games
- VR
- Database applications
- And much, much more!

Why Use C#?

- It is one of the most popular programming language in the world
- It is easy to learn and simple to use
- It has a huge community support
- C# is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
- As C# is close to [C](#), [C++](#) and [Java](#), it makes it easy for programmers to switch to C# or vice versa

C# Get Started

C# IDE

The easiest way to get started with C#, is to use an IDE.

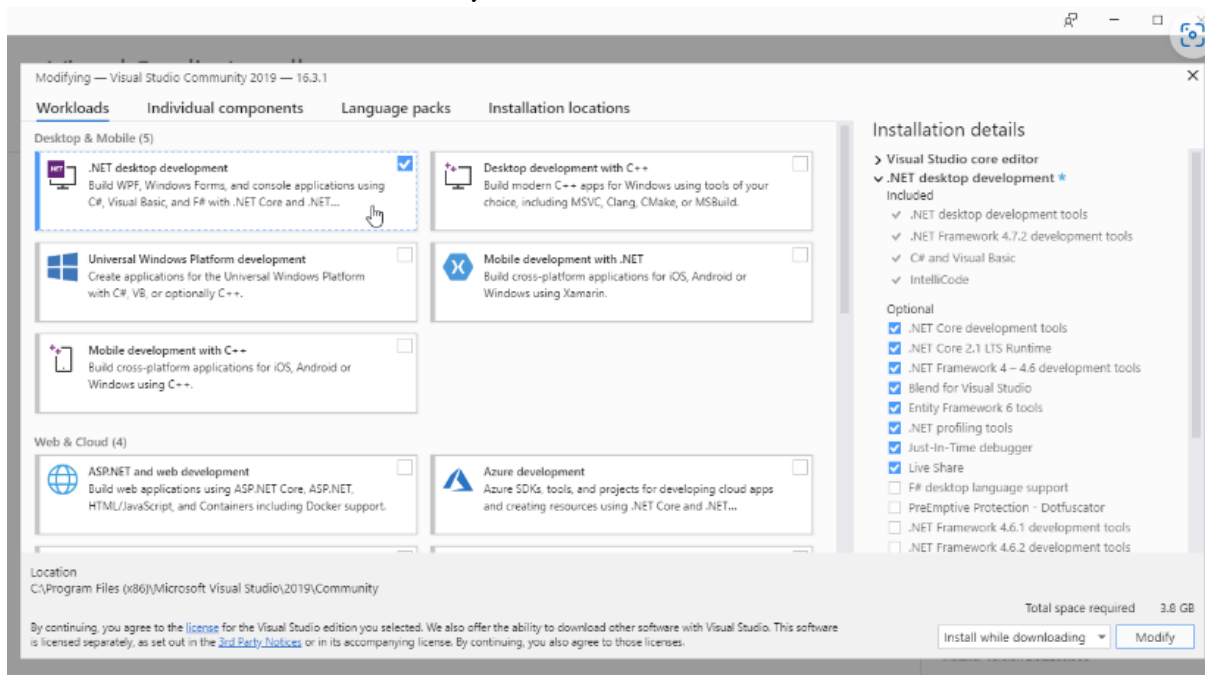
An IDE (Integrated Development Environment) is used to edit and compile code.

In our tutorial, we will use Visual Studio Community, which is free to download from <https://visualstudio.microsoft.com/vs/community/>.

Applications written in C# use the .NET Framework, so it makes sense to use Visual Studio, as the program, the framework, and the language, are all created by Microsoft.

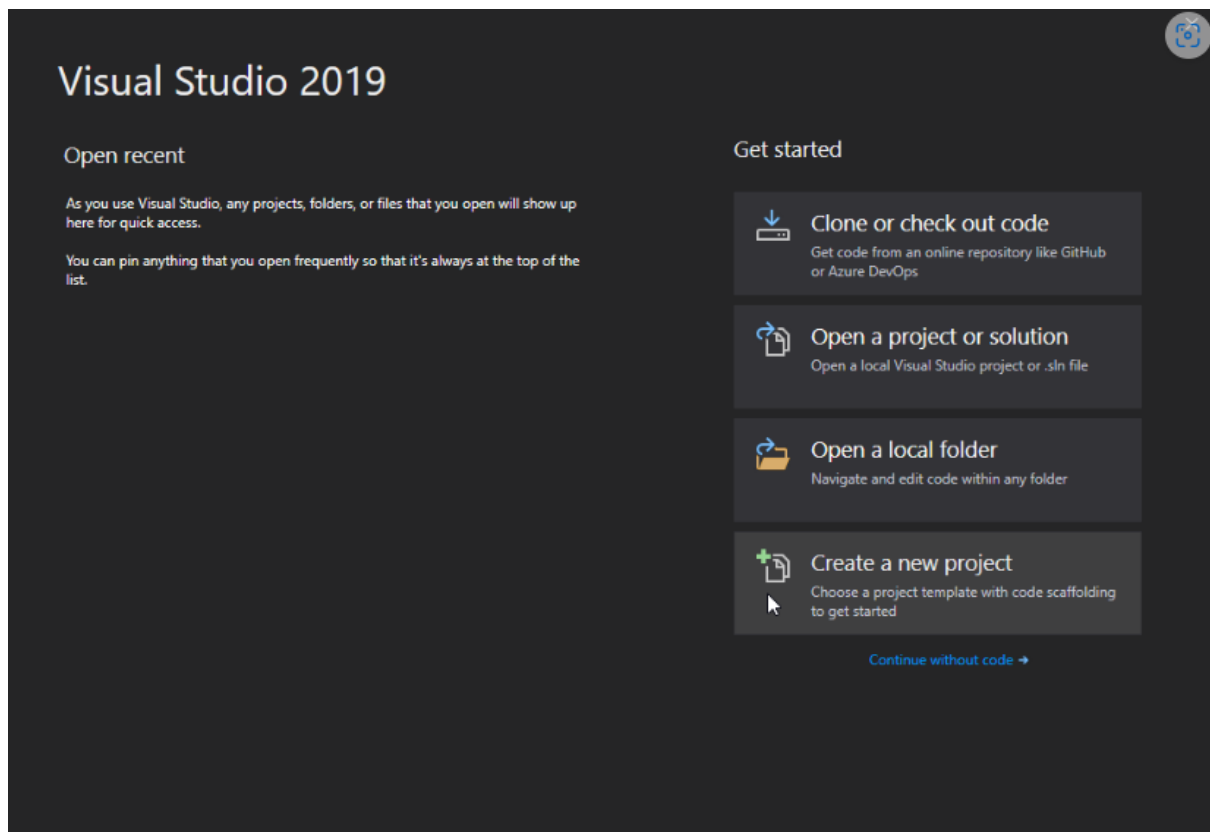
C# Install

Once the Visual Studio Installer is downloaded and installed, choose the .NET workload and click on the **Modify/Install** button:

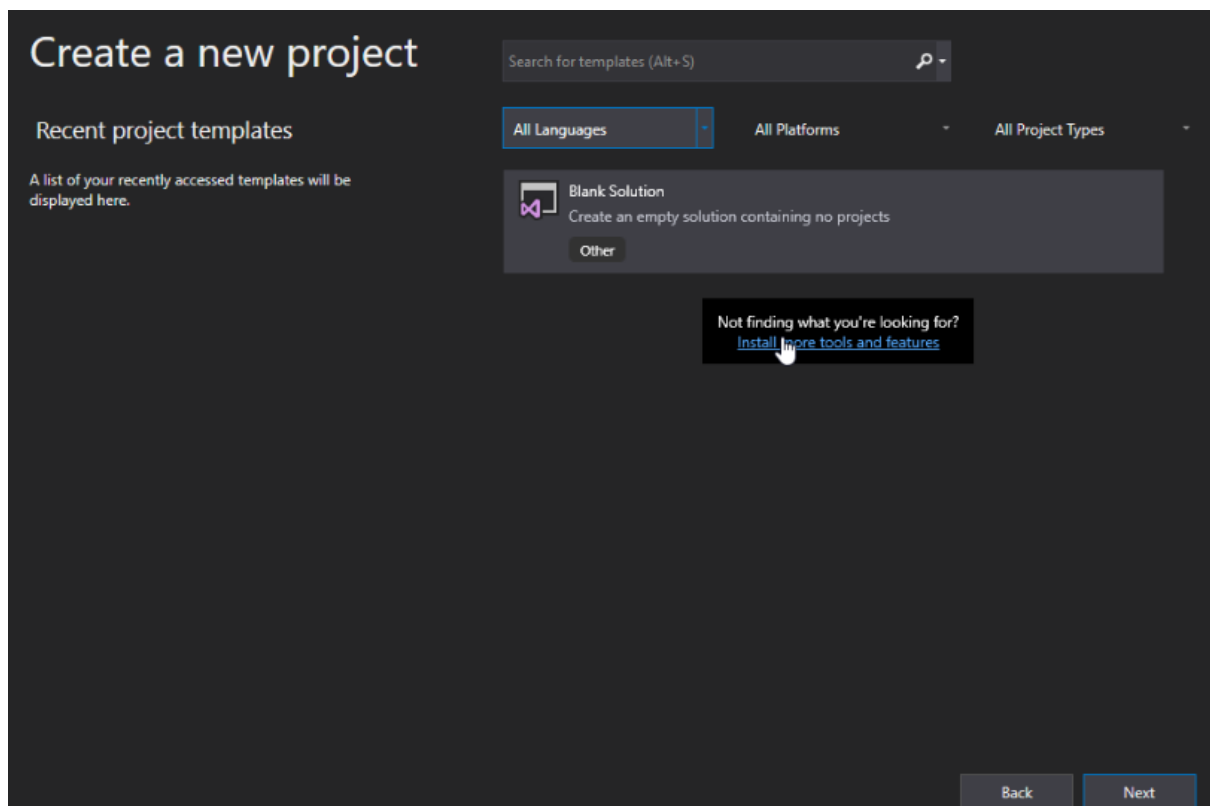


After the installation is complete, click on the **Launch** button to get started with Visual Studio.

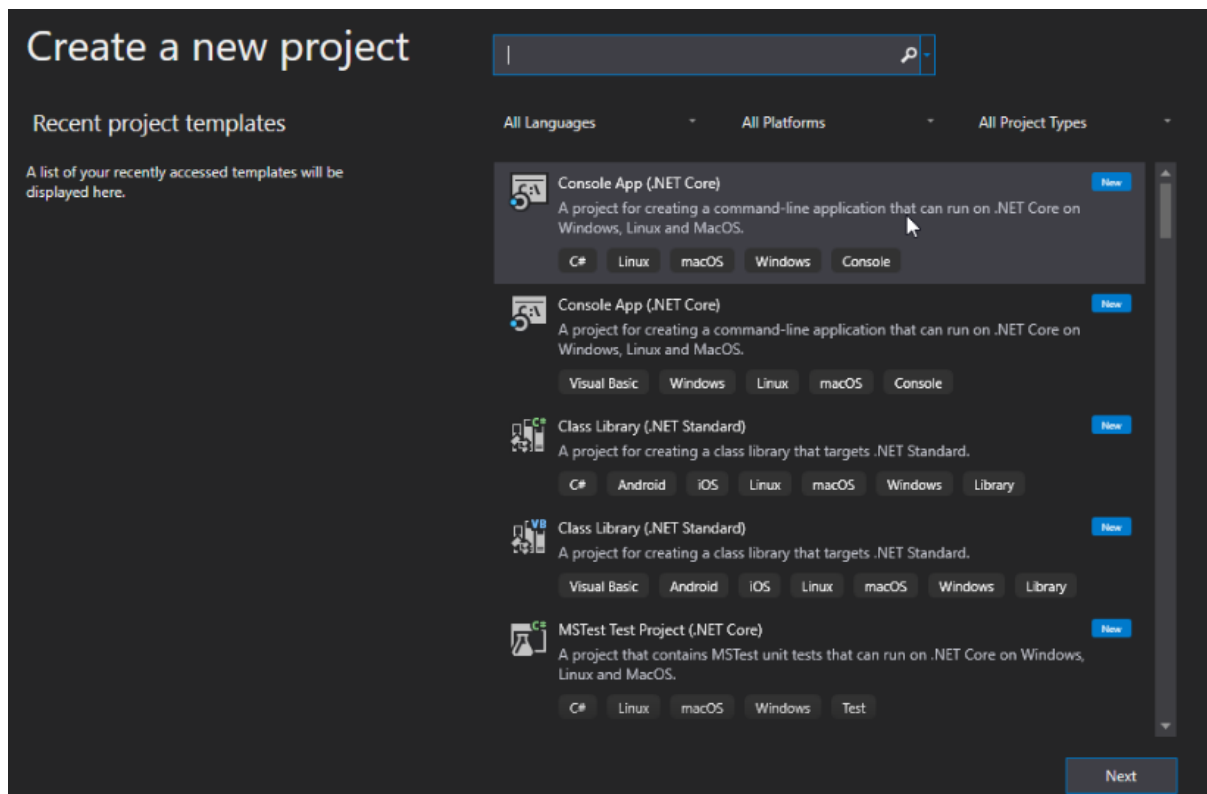
On the start window, choose **Create a new project**:



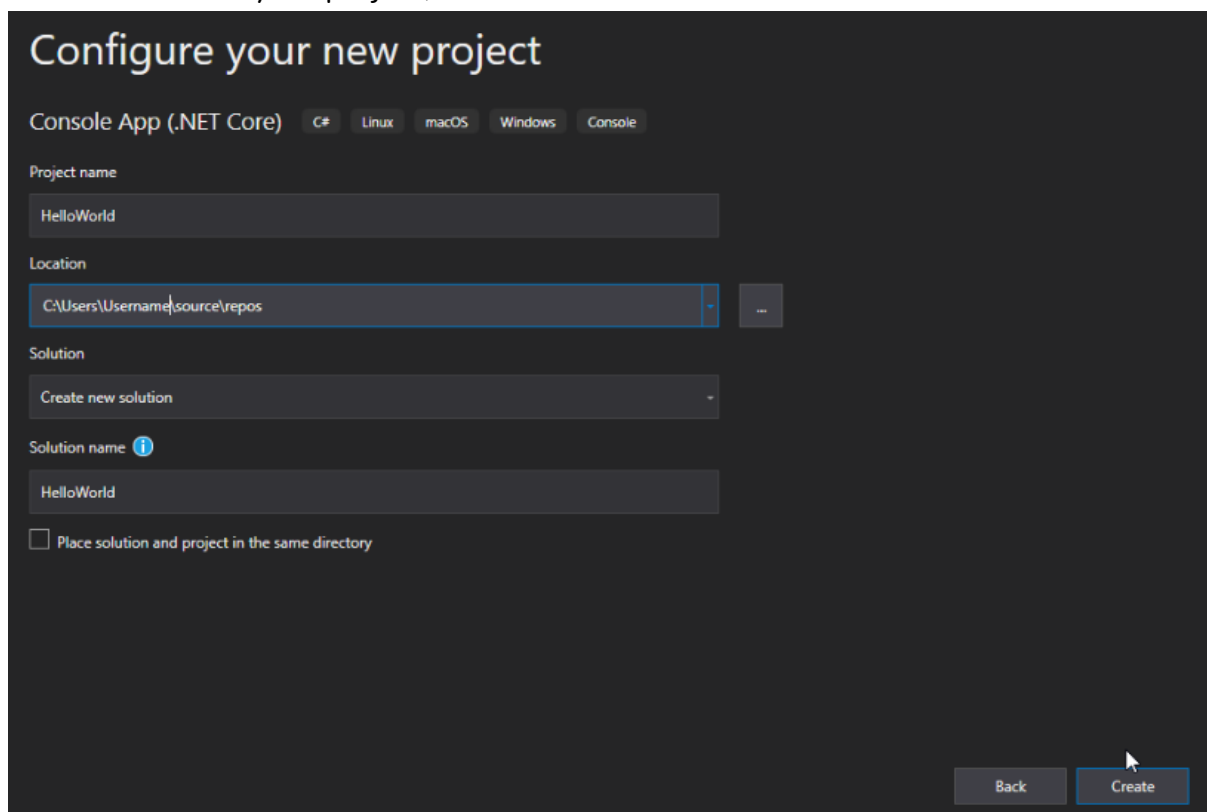
Then click on the "Install more tools and features" button:



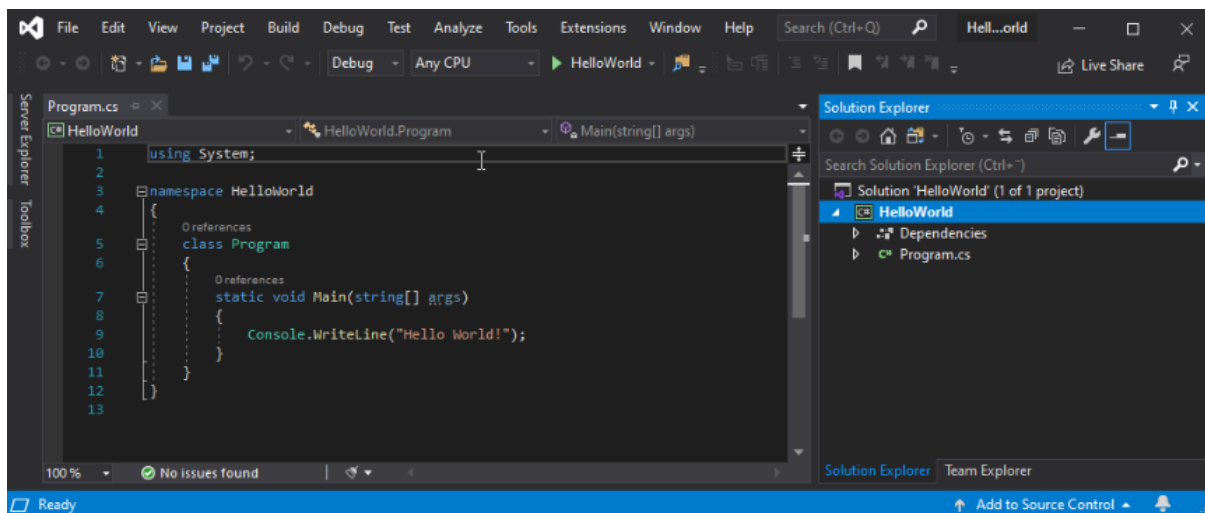
Choose "Console App (.NET Core)" from the list and click on the Next button:



Enter a name for your project, and click on the Create button:



Visual Studio will automatically generate some code for your project:



The code should look something like this:

Program.cs

```
using System;

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

Result:

```
Hello World!
```

Example explained

Line 1: `using System` means that we can use classes from the `System` namespace.

Line 2: A blank line. C# ignores white space. However, multiple lines makes the code more readable.

Line 3: `namespace` is used to organize your code, and it is a container for classes and other namespaces.

Line 4: The curly braces `{}` marks the beginning and the end of a block of code.

Line 5: `class` is a container for data and methods, which brings functionality to your program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.

Line 7: Another thing that always appear in a C# program, is the `Main` method. Any code inside its curly brackets `{ }` will be executed. You don't have to understand the keywords before and after Main. You will get to know them bit by bit while reading this tutorial.

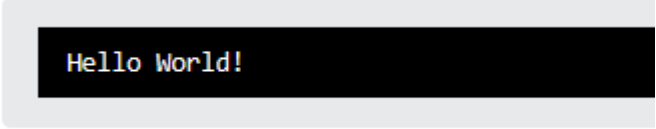
Line 9: `Console` is a class of the `System` namespace, which has a `WriteLine()` method that is used to output/print text. In our example it will output "Hello World!".

If you omit the `using System` line, you would have to write `System.Console.WriteLine()` to print/output text.

Note: Every C# statement ends with a semicolon `;`.

Note: C# is case-sensitive: "MyClass" and "myclass" has different meaning.

Note: Unlike [Java](#), the name of the C# file does not have to match the class name, but they often do (for better organization). When saving the file, save it using a proper name and add ".cs" to the end of the filename. To run the example above on your computer, make sure that C# is properly installed: Go to the [Get Started Chapter](#) for how to install C#. The output should be:



```
Hello World!
```

CHAPTER 2. DATA TYPES

As explained in the variables chapter, a variable in C# must be a specified data type:

Example

```
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
```

```
using System;
namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            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
            Console.WriteLine(myNum);
            Console.WriteLine(myDoubleNum);
            Console.WriteLine(myLetter);
            Console.WriteLine(myBool);
            Console.WriteLine(myText);
        }
    }
}
```

5
5.99
D
true
Hello

A data type specifies the size and type of variable values.

It is important to use the correct data type for the corresponding variable; to avoid errors, to save time and memory, but it will also make your code more maintainable and readable. The most common data types are:

| 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 |

Numbers

Number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are **int** and **long**. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. Valid types are **float** and **double**.

Even though there are many numeric types in C#, the most used for numbers are **int** (for whole numbers) and **double** (for floating point numbers). However, we will describe them all as you continue to read.

Integer Types

Int

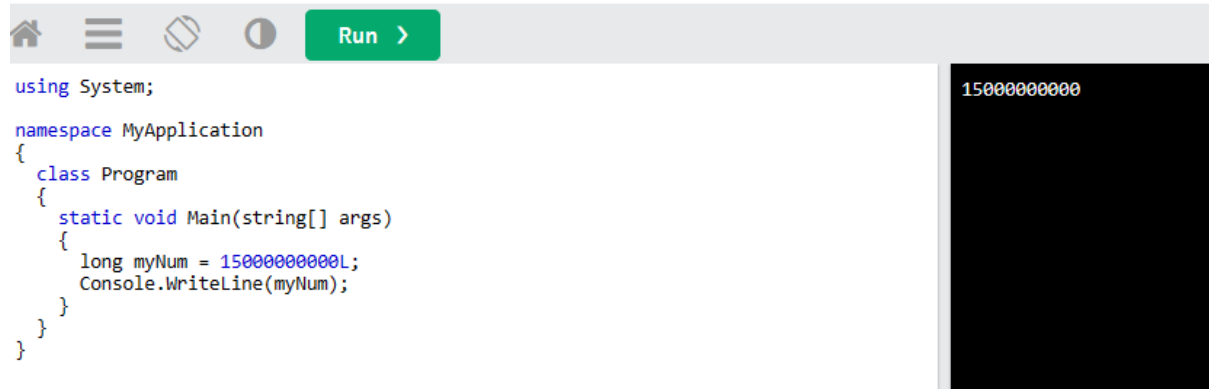
The **int** data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the **int** data type is the preferred data type when we create variables with a numeric value.

Example

```
int myNum = 100000;  
Console.WriteLine(myNum);
```


Long

The **long** data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when **int** is not large enough to store the value. Note that you should end the value with an "L":



```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            long myNum = 15000000000L;
            Console.WriteLine(myNum);
        }
    }
}
```

15000000000

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

The **float** and **double** data types can store fractional numbers. Note that you should end the value with an "F" for floats and "D" for doubles:

Float Example

```
float myNum = 5.75F;
Console.WriteLine(myNum);
```

Double Example

```
double myNum = 19.99D;
Console.WriteLine(myNum);
```

Use **float** or **double**?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of **float** is only six or seven decimal digits, while **double** variables have a precision of about 15 digits. Therefore it is safer to use **double** for most calculations.

Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

Example

```
float f1 = 35e3F;  
double d1 = 12E4D;  
Console.WriteLine(f1);  
Console.WriteLine(d1);
```

Booleans

A boolean data 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
```

Boolean values are mostly used for conditional testing, which you will learn more about in a later chapter.

Characters

The **char** data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

Example

```
char myGrade = 'B';  
Console.WriteLine(myGrade);
```

Strings

The **string** data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example

```
string greeting = "Hello World";  
Console.WriteLine(greeting);
```

Exercise:

Add the correct data type for the following variables:

```
 myNum = 9;  
 myDoubleNum = 8.99;  
 myLetter = 'A';  
 myBoolean = false;  
 myText = "Hello World";
```

Operators

Operators are used to perform operations on variables and values.

In the example below, we use the **+** **operator** to add together two values:

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int x = 100 + 50;  
            Console.WriteLine(x);  
        }  
    }  
}
```

150

Although the **+** operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example

```
int sum1 = 100 + 50;      // 150 (100 + 50)
int sum2 = sum1 + 250;    // 400 (150 + 250)
int sum3 = sum2 + sum2;    // 800 (400 + 400)
```

```
using System;
namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int sum1 = 100 + 50;      // 150 (100 + 50)
            int sum2 = sum1 + 250;    // 400 (150 + 250)
            int sum3 = sum2 + sum2;    // 800 (400 + 400)
            Console.WriteLine(sum1);
            Console.WriteLine(sum2);
            Console.WriteLine(sum3);
        }
    }
}
```

```
150
400
800
```

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 | $x--$ |

```
using System;
namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            int y = 3;
            Console.WriteLine(x + y);
        }
    }
}
```

```
8
```

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            x++;
            Console.WriteLine(x);
        }
    }
}
```

6

Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (**=**) to assign the value **10** to a variable called **x**:

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            Console.WriteLine(x);
        }
    }
}
```

10

The **addition assignment** operator (**+=**) adds a value to a variable:

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            x += 5;
            Console.WriteLine(x);
        }
    }
}
```

15

A list of all assignment operators:

| Operator | Example | Same As |
|----------|---------|------------|
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| *= | x *= 3 | x = x * 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |
| &= | x &= 3 | x = x & 3 |
| = | x = 3 | x = x 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |
| <<= | x <<= 3 | x = x << 3 |

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            x %= 3;
            Console.WriteLine(x);
        }
    }
}
```

2

Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either **True** or **False**. These values are known as *Boolean values*, and you will learn more about them in the [Booleans](#) and [If..Else](#) chapter.

In the following example, we use the **greater than operator** (>) to find out if 5 is greater than 3:

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            int y = 3;
            Console.WriteLine(x > y); // returns True because 5 is greater than 3
        }
    }
}
```

True

A list of all comparison operators:

| 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 |

Chapter 3. IF Condition

In this chapter, we'll discuss about:

- a. IF Statement
- b. Nexted IF

3.1 IF Statement

If statement is a fundamental control structure used to make decisions in your code. It allows you to execute a block of code if a specified condition is true. The code inside the if block is skipped, and your program continues with the next statement if the condition is false.

Here is the basic syntax of an if statement in C#:

```
if (condition)
{
    // Code to execute if the condition is true
}
```

The if keyword is followed by an expression enclosed in parentheses. This expression is the condition that you want to check. It can be any valid boolean expression that evaluates to either true or false.

If the condition inside the parentheses evaluates to true, the code block enclosed in curly braces { } following the if statement is executed. This code block can contain one or more statements.

The code inside the if block is skipped, and the program continues with the next statement after the if block if the condition is false.

Here is a simple example:

```
using System;

namespace IfConditionExample
{
    class Program
    {
        static void Main(string[] args)
        {
            int number = 10;
```

```
        if (number % 2 == 0)
        {
            Console.WriteLine("The number is even.");
        }
        else
        {
            Console.WriteLine("The number is odd.");
        }
    }
}
```

You can also use an else statement to specify a block of code that should be executed when the if condition is false. Here is an example:

```
int y = 3;
if (y > 5)
{
    Console.WriteLine("y is greater than 5");
}
else
{
    Console.WriteLine("y is not greater than 5");
}
```

In this case, because y is not greater than 5, the message "y is not greater than 5" will be printed.

You can also use else if clauses to test multiple conditions in sequence:

```
int z = 7;
if (z > 10)
{
    Console.WriteLine("z is greater than 10");
}
else if (z > 5)
{
    Console.WriteLine("z is greater than 5 but not greater than 10");
}
else
```

```
{  
    Console.WriteLine("z is not greater than 5");  
}
```

This allows you to handle different cases based on various conditions.

3.2 Nexted IF

Nested `if` statements in C# are used when you need to create more complex conditional logic by placing one `if` statement inside another. This allows you to test multiple conditions in a hierarchical manner. Each `if` statement is associated with its own block of code, and the inner `if` statements are only evaluated if the outer `if` conditions are true.

Here is the basic syntax of a nested `if` statement:

```
if (outerCondition)  
{  
    // Code to execute if the outer condition is true  
  
    if (innerCondition)  
    {  
        // Code to execute if both the outer and inner conditions are true  
    }  
    else  
    {  
        // Code to execute if the outer condition is true, but the inner condition is false  
    }  
}  
else  
{  
    // Code to execute if the outer condition is false  
}
```

Here is an example to illustrate the concept of nested `if` statements:

```
int age = 25;  
bool hasLicense = true;  
if (age >= 18)  
{
```

```
    Console.WriteLine("You are eligible to apply for a driving license.");

    if (hasLicense)
    {
        Console.WriteLine("You already have a driving license.");
    }
    else
    {
        Console.WriteLine("You can apply for a driving license.");
    }
}
else
{
    Console.WriteLine("You are not eligible to apply for a driving license.");
}
```

Explanation:

- The outer `if` statement checks if the `age` is greater than or equal to 18. If it is true, it displays a message about eligibility for a driving license.
- Within the outer `if` block, there is a nested `if` statement that checks if the `hasLicense` variable is `true`. If it is true, it displays a message indicating that the person already has a driving license; otherwise, it indicates that they can apply for one.
- If the outer `if` condition is `false`, it indicates that the person is not eligible for a driving license.

Nested `if` statements can be nested further, allowing you to create more complex conditional structures based on your specific requirements. However, it is important to keep the code organized and clear to avoid confusion, and you should consider using alternative control structures like `switch` statements or refactoring your code if the nesting becomes too deep and complex.

References:

1. [If Statement in JavaScript \(tutorjoes.in\)](http://tutorjoes.in)

Chapter 4. Looping

Looping is a fundamental concept in programming that allows you to repeatedly execute a block of code as long as a specific condition is met or for a predetermined number of times. In C#, there are several looping constructs to achieve this:

4.1 For Loop

The `for` loop is commonly used when you know in advance how many times you want to execute a block of code. It consists of three parts: initialization, condition, and iterator.

```
for (int i = 0; i < 5; i++)  
{  
    // Code to be executed repeatedly  
}
```

In this example, the loop initializes `i` to 0, executes the code block while `i` is less than 5, and increments `i` by 1 in each iteration.

4.2 While Loop

The while loop repeatedly executes a block of code as long as a specified condition is `true`. It is suitable when you don't know in advance how many iterations are needed.

```
int count = 0;  
while (count < 5)  
{  
    // Code to be executed repeatedly  
    count++;  
}
```

This loop will execute the code block as long as `count` is less than 5.

4.3 Do-while Loop

The `do-while` loop is similar to the `while` loop, but it ensures that the code block is executed at least once before checking the condition.

```
int count = 0;  
do  
{  
    // Code to be executed repeatedly  
    count++;  
}  
while (count < 5);
```

In this example, the code block will be executed at least once, even if `count` is initially not less than 5.

4.4 Foreach Loop

The `foreach` loop is used for iterating over collections (arrays, lists, etc.) without worrying about indexing. It iterates through each element in the collection.

```
int[] numbers = { 1, 2, 3, 4, 5 };  
foreach (int num in numbers)  
{  
    // Code to be executed for each element in the array  
}
```

This loop will iterate through each element in the `numbers` array and execute the code block for each element.

4.5 Break and continue Statements

Inside loops, you can use the `break` statement to exit the loop prematurely when a certain condition is met, and the `continue` statement to skip the rest of the current iteration and move to the next iteration.

```
for (int i = 0; i < 10; i++)  
{  
    if (i == 5)  
    {  
        break; // Exit the loop when i is 5  
    }  
    if (i % 2 == 0)  
    {  
        continue; // Skip even numbers  
    }  
    // Code to be executed  
}
```

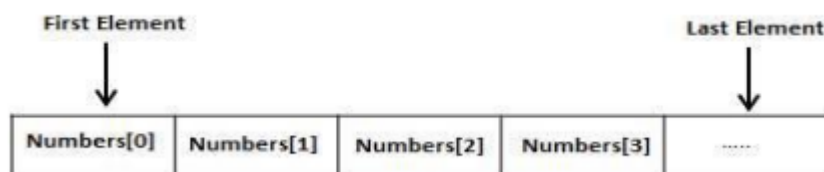
Looping is a powerful way to perform repetitive tasks in your programs, and choosing the appropriate loop construct depends on your specific requirements and the structure of your data. It's important to ensure that the loop's conditions are properly managed to avoid infinite loops or unexpected behaviour.

Chapter 5. Array

5.1 C# Arrays

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type stored at contiguous memory locations.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables. A specific element in an array is accessed by an index. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



5.2 Declaring Array Types

To declare an array in C#, you can use the following syntax –

```
datatype[] arrayName;
```

where,

- *datatype* is used to specify the type of elements in the array.
- `[]` specifies the rank of the array. The rank specifies the size of the array.
- *arrayName* specifies the name of the array.

For example,

```
double[] balance;
```

5.3 Initializing an Array

Declaring an array does not initialize the array in the memory. When the array variable is initialized, you can assign values to the array. Array is a reference type, so you need to use the **new** keyword to create an instance of the array. For example,

```
double[] balance = new double[10];
```

5.4 Using the foreach Loop

In the previous example, we used a for loop for accessing each array element. You can also use a **foreach** statement to iterate through an array.

```
using System;

namespace ArrayApplication {
    class MyArray {
        static void Main(string[] args) {
            int[] n = new int[10]; /* n is an array of 10 integers */

            /* initialize elements of array n */
            for ( int i = 0; i < 10; i++ ) {
                n[i] = i + 100;
            }

            /* output each array element's value */
            foreach (int j in n) {
                int i = j-100;
                Console.WriteLine("Element[{0}] = {1}", i, j);
            }
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```


References:

2. [If Statement in JavaScript \(tutorialjoes.in\)](https://www.tutorialjoes.in/2018/05/20/javascript-if-statement/)
3. <https://www.w3schools.com/>
4. chatgpt
5. <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/collections>