Install Microsoft Visual Studio -> Visual Studio Setup.exe -> It will download installer, then start installing it -> a screen then appears for the things you want to download -> .NET framework for console -> around 7 GB, download then install -> Launch Visual Studio 2022 -> Create new Project -> Choose C# project for console -> name the project -> Solution space name could be different , it is a container for many different projects that you will build later on -> chose the directory of your workspace -> Create Project.

Literals: fixed values, literally the character that is typed.  
Data types: integer, long, float, double, decimal  
Variable name: may use Camel case; start with alphabet or underscore; container for holding values

Console.WriteLine("Hello World!!"); // Outputs this line on terminal and appends a new line  
int marks; // Variable Declaration- int can hold only integers of 32-bit signed integer – default  
marks = 21; // Variable initialization using assignment operator, done from right to left  
int age=33, x, y, salary = 2400000; // Multiple Variable Declaration and initialization.  
// x = 5, y= 6; Cannot assign like this using comma separator  
long ageOfUniverse = 160000000000L; // long integer - 64 bit signed   
double score = 36.4342D; // double precision floating point number – default  
float weight = 65.4f; // single precision floating point number - suffix f or F.  
decimal money = 793.34M; // decimal number

A structure is a value type that derives implicitly from [System.ValueType](https://docs.microsoft.com/en-us/dotnet/api/system.valuetype), which in turn is derived from [System.Object](https://docs.microsoft.com/en-us/dotnet/api/system.object). Like classes, structures define both data (the fields of the structure) and the operations that can be performed on that data (the methods of the structure). This means that you can call methods on structures, including the virtual methods defined on the [System.Object](https://docs.microsoft.com/en-us/dotnet/api/system.object) and [System.ValueType](https://docs.microsoft.com/en-us/dotnet/api/system.valuetype) classes, and any methods defined on the value type itself. In other words, structures can have fields, properties, and events, as well as static and nonstatic methods. You can create instances of structures, pass them as parameters, store them as local variables, or store them in a field of another value type or reference type. Structures can also implement interfaces.

Value types also differ from classes in several respects. First, although they implicitly inherit from [System.ValueType](https://docs.microsoft.com/en-us/dotnet/api/system.valuetype), they cannot directly inherit from any type. Similarly, all value types are sealed, which means that no other type can be derived from them. They also do not require constructors.

**Objects:  
#**In objects from the real world (as well as in the abstract objects) we can distinguish the following two groups of their characteristics:

- **States** – these are the characteristics of the object which define it in a way and describe it in general or in a specific moment

- **Behavior** – these are the specific distinctive actions, which can be done by the object.

**#**Objects in OOP combine data and the means for their processing in one. They correspond to objects in real world and contain data and actions:

- **Data members** – embedded in objects variables, which describe their states.   
- **Methods -** implement behavior  
  
**Class:  
#**The **class** defines abstract characteristics of objects. It provides a structure for objects or a pattern which we use to describe the nature of something (some object). **Classes are building blocks of OOP** and are inseparably related to the **objects**. Furthermore, each object is an **instance** of exactly one specific class.

We are going to give as an **example a class and an object**, which is its instance. We have a **class Dog** and an **object Lassie**, which is an instance of the class **Dog** (we say it is an object of type **Dog**). The class **Dog** describes the characteristics of all dogs whereas **Lassie** is a certain dog.   
  
**Class** in the OOP is called a definition (**specification**) of a given type of objects from the real-world. The class represents a pattern, which describes the different states and behavior of the certain objects (the copies), which are created from this class (pattern).

**Object** is a copy created from the definition (specification) of a given class, also called an **instance**. When one object is created by the description of one class we say **the object is from type "name of the class"**.  
**#**The class defines the **characteristics of an object** (which we are going to call **attributes**) and its **behavior** (actions that can be performed by the object). The attributes of the class are defined as its own variables in its body (called **member variables**). The behavior of objects is modeled by the definition of **methods** in classes.

**#**In the context of such behavior the object consists of two things: current **state** and **behavior** defined in the class of the object. The state is specific for the instance (the object), but the behavior is common for all objects which are instances of this class.

A **class** in C# is defined by the keyword **class**, followed by an identifier (name) of the class and a set of data members and methods in a separate code block.

**Classes** in C# can contain the following elements:

- **Fields** – member-variables from a certain type;   
- **Properties** – these are a special type of elements, which extend the functionality of the fields by giving the ability of extra data management when extracting and recording it in the class fields;   
- **Methods** – they implement the manipulation of the data.

public class Cat

{

// Field name

private string name;

// Field color

private string color;

public string Name

{

// Getter of the property "Name"

get

{

return this.name;

}

// Setter of the property "Name"

set

{

this.name = value;

}

}

public string Color

{

// Getter of the property "Color"

get

{

return this.color;

}

// Setter of the property "Color"   
set

{

this.color = value;

}

}

// Default constructor

public Cat()

{

this.name = "Unnamed";

this.color = "gray";

}

// Constructor with parameters

public Cat(string name, string color)

{

this.name = name;

this.color = color;

}

// Method SayMiau

public void SayMiau()

{

Console.WriteLine("Cat {0} said: Miauuuuuu!", name);

}

}   
The example class **Cat** defines the **properties Name** and **Color**, which keep their values in the hidden (private) **fields name** and **color**. Furthermore, two **constructors** are defined for creating instances of the class **Cat**, respectively with and without parameters, and a **method** of the class **SayMiau()**.  
After the example class is defined we can now use it in the following way:   
static void Main()

{

Cat firstCat = new Cat();

firstCat.Name = "Tony";

firstCat.SayMiau();

Cat secondCat = new Cat("Pepy", "red");

secondCat.SayMiau();

Console.WriteLine("Cat {0} is {1}.",

secondCat.Name, secondCat.Color);

}

Calling the method **Console.WriteLine(…)** of the class **System.Console** is an example of usage of a **system class** in C#. We call system classes the classes defined in **standard libraries** for building applications with C# for example the classes **String**, **Environment** and **Math .**It is important to know that the implementation of the logic in classes is **encapsulated** (hidden) inside them. For the programmer it is important what they do, not how they do it and for this reason a great part of the classes is not publicly available (**public**). With system classes the implementation is often not available at all to the programmer. Thus, new **layers of abstraction** are created which is one of the basic principles in OOP.