**//---------------------------------------------------------- Strings -----------------------------------------------------------//**

// **ASCII** is able to record only 128 or 256 characters (with 7-bit or 8-bit table).  
// In. NET Framework each character has a serial number from the **Unicode table**.(16 bit code table)  
// Thus **Unicode** table stores 2^16 = 65,536 characters. Some characters are encoded in a specific way, so as to use two characters of the Unicode table to create a new character – the resulting signs exceed 100,000.  
// A **String** is a **sequence of characters** stored in a certain address in memory.  
// .NET Framework provides the class **System.String :** the keyword **string** is an alias in C# of the **System.String** class.  
// Since type **string** is a class, it complies with the principles of object-oriented programming. Its values are stored in the dynamic memory (**managed heap**), and the variables of type **string** keeps a **reference** to an object in the heap.

string newStr; // Variable **declaration** of type **string** ; not allocating memory for it!   
// The declaration informs the compiler that variable **newStr** will be used & expected type for it is **string**. - did not yet create a variable in the memory ; not available for processing yet (value is **null**, which means no value).  
// Uninitialized variables of type **string** do not contain empty values, it contains the special value **null** – and each attempt for manipulating such a string will generate an error/exception : **NullReferenceException**

// **Initialize** a string variable by **setting a string literal** / **assign a predefined textual content** to a variable of type **string**. Used when we know the value.  
  
string greeting = "Hello, C#"; // Creating/Declaring and Initializing a string

 // Internal representation of the String class : an array of characters.  
// We may use the type **char[]** and fill in the array’s elements character by character, but this is disadvantageous. Filling in the array happens character by character, not at once. We should know the length of the text in order to be aware whether it will fit into the already allocated space for the array. The text processing is manual.   
// The character sequences stored in a variable of the string class are never changing (**immutable**). After being assigned once, the content of the variable does not change directly – if we try to change the value, it will be saved to a new location in the dynamic memory and the variable will point to it., but it is allowed only to read characters (and not to write to them):

string str = “abcde“; // Strings are very similar to the char arrays (**char[]**) ,in the sense that they have properties such as **Length,** allows …  
char ch = str[1]; // … access by index with the indexer operator []. Indexing, as it is used in arrays, takes indices from **0** to **Length-1**., but… // …unlike char[] , string **cannot be modified**. It is allowed only to read characters (and not to write to them).  
// This produces compilation error : str[1] = ‘a’;   
// This produces IndexOutOfRangeException error: ch = str[50];

string quote = "Book's title is \"Intro to C#\"";  
// Displaying special characters in the source code is called **escaping**. For example, to make quotes in the example are part of the text, we place them within the variable after the escaping character backslash (**\**). In this way the compiler recognizes that the quotes are not used to start or end a string, but are a part of the data.  
  
 public partial class Molecules // Syntax: Access- IndexOutOfRangeException

modifier Type-modifier class ClassName  
{  
 public int radius; // Instance Variable: All objects will have their own copy/different value of this field.  
 public float velocity;  
 public float charge;  
 public string name;  
 protected double VolumeConstant = 4 \* Math.PI / 3; //   
 public void PrintState()  
 {  
 Console.WriteLine("This {0} molecule is moving with a speed {1}",name,velocity);  
 }  
 public void RadiusChange(int delta) => radius += delta;  
}

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

**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.