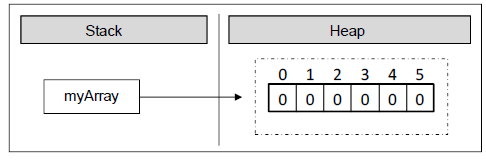
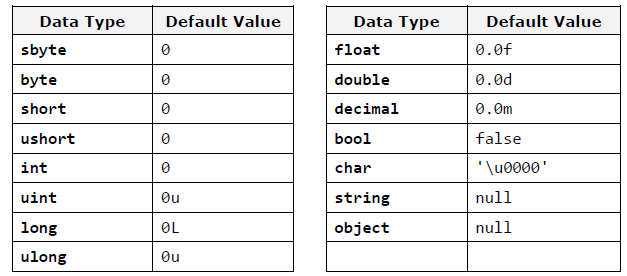
Install Microsoft Visual Studio -> Visual Studio Setup.exe -> It will download installer, then start installing it -> a screen then appears for the things you want to download -> .NET framework for console -> around 7 GB, download then install -> Launch Visual Studio 2022 -> Create new Project -> Choose 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

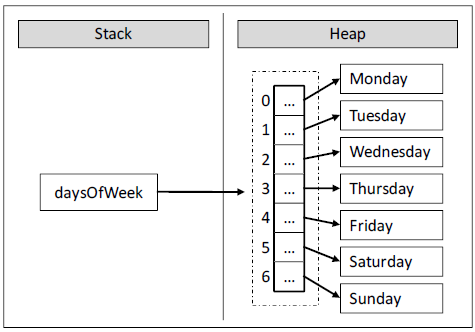
Console.WriteLine("Hello World!!"); // Outputs this line on terminal and appends a new line  
Console.ReadKey(); // Obtains the next character or function key pressed by user and displays it onto console. If it is the last statement the window will close following our key press.  
Console.ReadLine();// Reads the next line of characters from the standard input stream and returns this; return type: string  
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

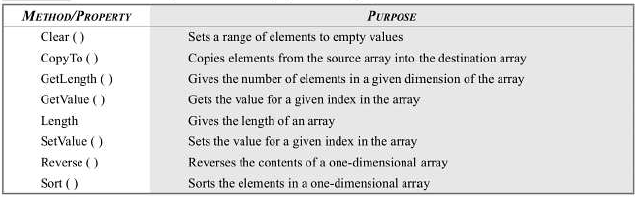
int num = Convert.ToInt32(Console.ReadLine()); // For Type Conversion; here string output is converted to integer

//--------------------------------------------------------Arrays-----------------------------------------------------//

//Array - ordered sequence(set of contiguous data points) of elements/variables of the same type (primitive or reference)  
//This list is given a single variable name - but each element is accessed by a numerical index/subscript starting from 0(called lower bound)  
// 1-dimensional arrays - vectors ; 2-dimensional arrays – matrices  
int[] marks; // [] indicates we are declaring an array of int type elements, not a single element  
string[] names; // On declaration, we create a variable with the identifier marks, names in the **program's execution stack**  
float[] average, percentile; // No memory is allocated yet; declaration only creates a reference which doesn't have a value(points to null)  
//In above lines, we declared *int array reference, string array reference*, etc.  
marks = new int[3]; // We *create array* with the '*new*' keyword - allocates area for 3 integers in the **dynamic memory(heap),**indexed 0 to N-1  
names = new string[5];// Array creation automatically initializes the array elements to a default value, in case user didn't initialize it  
// For bool array, default value is false ; for int array, it is set to 0  
Console.WriteLine(marks[2] + " " + names[3] ); // Outputs: 0 ; string is initialized to null, not seen on output   
if ( names[4] == null )  
{  
 Console.WriteLine("The value is null"); // Outputs: The value if null ; this is how we check  
}  


string[] locations = new string[4]; // Declaration and Creation can be combined into one step  
marks[0] = 4; // Initialization can be done elementwise by accessing each index  
marks[1] = 9; // Trying to access an array beyond its bound will generate error   
marks[2] = -3; // We can access given elements of the array both for reading and for writing, which means we can treat elements as variables.  
  
//marks[**3**] = 10; --> The .NET Framework does automatic check; outputs : System.IndexOutOfRangeException: 'Index was outside the bounds of the array.’  
// Initialization: Specify each array item within the scope of curly brackets ({}) - helpful when creating an array of known size quickly specify the initial values  
string[] wizards = new string[6] { "Harry", "Potter", "Ginny", "Hermione", "Ronald", "Yennefer" };  
// Declaration, Creation, Initialization all in same step  
// Need not specify the size of the array within [], when using curly bracket syntax to specify elements as length of array is auto inferred by number of items within the scope of the curly brackets  
average = new float[] { 4.6f ,7.8f ,9.2f }; // Creation and Initialization can be done in one step   
bool[] states = { true, false, false, true, true };  
string[] daysOfWeek = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};  
// Use of the new keyword is optional  
/\* Strings in C# are immutable reference types, unlike integer which are value types. Array of integers holds the int value but array of string holds the reference to the elements and not the actual values. In above case we allocate an array of seven elements of type string. The type string is a reference type (object) and its values are stored in the dynamic memory. The variable daysOfWeek is allocated in the stack memory, and points to a section of the dynamic memory containing the elements of the array. The type of each of these seven elements is string, which itself points to a different section of the dynamic memory, where the real value is stored.\*/

  
// If there is mismatch b/w the declared size within[] & the no. of initializers (too many or too few ), we are issued a compile-time error.  
var prices = new[] { 345.6, 627.3, 419, 215.6 };// Define implicitly typed local arrays using ‘var’, like implicitly typed local variables  
// Here we allocate a new array variable without specifying the type contained within array itself(note we must use the new keyword in this approach).  
// Compiler auto determines underlying type; Items in array’s initialization list must have **same** underlying type; Mixed types generate compile time error  
var greetings = new[] { "Hello", null, "Namaste" }; // prices is really double, greetings is string  
Console.WriteLine("prices is a: {0}", prices.ToString()); // Outputs: prices is a System.Double[]  
// object.ToString returns a string that represents the current object;

// In C#, all arrays are class based; any array we create is automatically derived from **System.Array** class, which defines many methods and properties to efficiently manipulate arrays; these are accessed using the dot operator.  
  
Console.WriteLine("Wizards array has {0} names.", wizards.Length); // Outputs Wizards array has 6 names

Console.ReadLine();

//----------------------------------------------------------Methods----------------------------------------------------------//  
class Arithmetic // Every method must be contained within a class   
{  
 int Cube (int x) // Method Declaration Syntax: access-modifiers return-type method-name(formal-parameter-list)  
 { // method-name is any valid C# identifier – Cube ; prefer PascalCasing Convention  
 // parameter list- contains all variable names with type which we give as input  
 return x \* x \* x; // Type of value method returns (type in declaration)- can also be any class type  
 }  
 float Product(float x, float y) // Each formal-parameter must be declared for their type individually  
 {  
 float m = x \* y; // m - local variable  
 return m; // Function - Methods which return any value   
 }  
 void Sum(float x, float y) // Can never omit return type even if method doesn't return anything- void  
 {  
 Console.WriteLine("Sum is " + (x + y)); // return statement is omitted  
 }  
}

//----------------------------------------Classes---------------------------------------------//

/\* A Software models real world things (goods, cars, spheres) or abstract concepts like vectors, list, stacks etc as Objects.\*/  
// All those real world or abstract Objects have two distinct characteristics - State and Behaviour  
// State - characteristics which define the condition of object at a given moment / general time  
// State - represented by values of data members / fields / member variables  
// Behaviour - specific distinctive actions that can be done by the objects   
// Behaviour – modelled or implemented by function members/ methods  
// Class - User defined data type with template for field data & members which operate on this data(contructors,properties,methods,events,etc)  
// The data members of class - field and properties are collectively called Attributes.  
/\* Properties - special elements which control access to fields, extend the functionality of the fields by giving the ability of extra data management when extracting and recording it in the class fields (using get , set).\*/   
// Objects - Variables of this user-defined data type/class; called Instances of the class; object creation: Instantiation  
// The state is specific to instance/object but behaviour is common to all instances of the class.

class EmptyMan // class - keyword, followed by valid C# identifier  
{  
 // State, Behaviour are all optional  
} // Empty class has many uses: Marker interfaces, Type parameters in generics, Singleton pattern, Placeholder Code, etc

public partial class Molecules // Syntax: Access-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.