* In C# putem sa avem un tip de programare ca in python, unde direct executam instructiunile fara functii, fara clase, dar in versiunile mai vechi trebuia facut asa:

namespace nume{

class clasa{

static void Main(){

}

}

}

**Input/output**

* **Pentru a afisa mesaje in consola folosim:**

Console.Write()

Console.WriteLine()

Console.WriteLine(“{0} {1}”,x,y);

* **Pentru a scrie in consola:**

Console.ReadLine(); - pana la \n

Console.Read() – doar un caracter(char)

Daca dorim sa scriem alt tip de date, facem o conversie ca in Python la input introdus, doar ca se face asa:

Convert.ToDouble();

Convert.ToBoolean();

Convert.ToInt16(); - int pe 16 biti

Convert.ToInt32(); - cea implicita

Convert.ToInt64();

Fie punem drept parametru o variabila, fie punem direct Console.ReadLine();

Convert.toDouble(Console.ReadLine());

~~Convert.toDouble(Console.Read());~~ - deoarece se citeste doar un char, iar in C# orice char este un intreg corespunzator in ASCII, asta si fiind rezultatul conversiei.

* **Comentarii:**

// - pe o linie

/\*

\*/ - mai multe linii

**Arrays**

* **Ca si in C++ si Java, un array este un obiect creat dynamic, dar de tip System.Array**
* **[][] se pun mereu pana la variabila!**

**~~int a[][];~~**

* Toate au un atribut **length**, ca si in Java.
* Ca si in Java, orice array necesita alocarea dinamica a memoriei cu new.

int[] table = new int[nr elemente]

int[] table = new int[nr elemente]{1,2,3,4,5} – asa ceva in Java nu e posibil, insa numarul de elemente dintre {} trebuie sa fie egal cu cel declarat intre []

int[] table = {1,2,3,4,5}

* Ca si in Java, cand se creaza un array doar cu new, el automat se initializeaza cu 0,null sau false,oriunde nu ar fi, spre deosebire de C++ care nu face asta daca array e declarant direct intr-o metoda.

**MultiArrays**

**Rectangular Array(regulat)**

* Acest tip de Array este special pentru C#.
* El se declara asa:

int[,] a = new int[lung1,lung2];

* Accesarea se face: a[poz1,poz2]
* Nu putem crea un tabloul cu tablouri de difertie dimensiuni! Trebuie mereu de oferit doua lungimi intre ,

~~int a[,] = new int[10,]~~

* Putem sa le cream si cu dimensiune automata:

int [,] = new int[]{{1,2,3},{4,5,6}}

dar dimensiunea lor trebuie sa fie egala mereu!

~~int [,] = new int[]{{1,2,3,3,3},{4,5,6}}~~

**Jagged Array(neregulat)**

* Este un array de arrayuri in care fiecare array are un numar nedetrminat de elemente.
* Arrayurile unui astfel de tablou pot avea dimensiuni variabile, ca si In Java.

E sufficient sa nu specificam cate elemente are fiecrae tablou la initializare, si sa o facem manual pentru fiecare dupa.

int[][] a = new int[2][];

a[0] = new int[5];

a[1] = new int[10];

* Putem combina tabelele neregulate cu cele regulate:

**int**[][, ] arr = **new** **int**[3][, ] {**new** **int**[, ] {{1, 3}, {5, 7}},**new** **int**[, ] {{0, 2}, {4, 6}, {8, 10}},**new** **int**[, ] {{11, 22}, {99, 88}, {0, 9}}};

dar tabelele create cu [,] trebuie sa aiba acelasi numar de elemente!

* int[5][10] a – nu mai e socotit jagged array
* nu putem ca si in Java sa setam automat si numarul de tabele si de elemente ale acestuia daca avem un tablou de tablouri:

~~int[][] a = new int[5][10];~~

facem:

int[][] a = new int[5][]

si abea apoi pentru fiecare tablou stabilim numarul de elemente, ca in C++;

**Modificatori de acces**

* public
* protected-internal – poate fi accesat de orice clasa in acelasi fisier, dar numai de clasele copil in alte fisiere, ceva asemanator cu Java cu packageurile.
* protected – nu il putem accesa decat in clasa de baza si in clasa derivata, ca in C++
* internal (e default) – putem sa il accesam doar in fisierul curent, nu package ca in java, ci fisier doar!
* Private-protected – poate fi accesat de clasa curenta si doar de clasele copii din acelasi fisier.
* Private

**Interface**

* In C# nu se permite mostenirea multipla a claselor, dar se permite a interfatelor
* Interfatele nu pot avea membri private
* Toti membrii by default sunt public si abstract.
* Interfatele pot contine doar metode si proprietati, nu si variabile declarate!
* Cand suprascriem o metoda in clasa, nu trebuie sa folosim override

**Pasarea argumentelor in metode**

* Cand trimitem argumentele la o metoda, putem specifica numele lor in metoda func(a : 10, b: 20), asa nu conteaza ordinea in care le trimitem.
* Putem avea argumente cu valori default, la final.

**Out, Ref si In**

* In c#, datele primitive sunt trimise prin valoare, iar obiectele prin referinta. Totusi, un parametru dintr-o functie, odata ce i se trimite un obiect, el va avea o referinta la el, insa aceasta referinta o poate modifica prin egalarea lui cu un alt obiect, si asa pierde referinta parametrului trimis , ca in Java exact.
* Ref – parametrul din metoda, desi e egalat cu un alt obiect, il va influenta pe cel care are referinta. Ref va face ca parametrul sa nu aiba doar o simpla referinta, ci chiar sa copie adresa de memorie, si deci orice modificare va fi facuta chiar in adresa de memorie, ca & la C++. Obiectul trimis prin ref trebuie sa fie initializat pana a fi trimis!
* Out – exact ca ref, doar ca obiectul ce este trimis nu trebuie neaparat sa fie initializat pana la trimiterea lui, dar trebuie neaparat initializat in metoda.

Ref si out trebuie folositi si la parametrii formali, si actuali!

* In – se asigura ca obiectul nu va fi modificat in metoda, ci doar folosit.

**Override Method**

* Pentru a putea suprascrie o metoda, trebuie sa o declaram ca virtuala in clasa de baza si apoi sa folosim override in loc de virtual in clasa copil.
* Putem evita metodele virtuale, ca in C++, ceea ce va duce ca metoda apelata din clasa parinte nu va mai cauta in namespace principal daca nu cumva e o metoda suprascrisa. Pentru asta, la clasa parinte nu punem nimic la metoda, iar in clasa copil cand o suprascriem, folosim **new**
* **Base = super din Java**

**ReadOnly**

O variabila membru declarata ca ReadOnly poate fi modificata doar in constructor sau la momentul crearii obiectului.

**Properties**

Folosim properties in loc de setteri si getteri ca metode. Se face asa:

TipAccesor TypeRteurn nume{

Set { member = oriceNume}

Get { return member }

}

Nu conteaza ce nume are propertie, sa nu coincida doar cu membrii deja existenti. Cand vom vrea sa folosim setterul, facem asa:

Obiect.oriceNume = Valore

Cand vrem sa folosim getter:

Obiect.oriceNume