Tutorial 1.

Introducere rapidă în C#

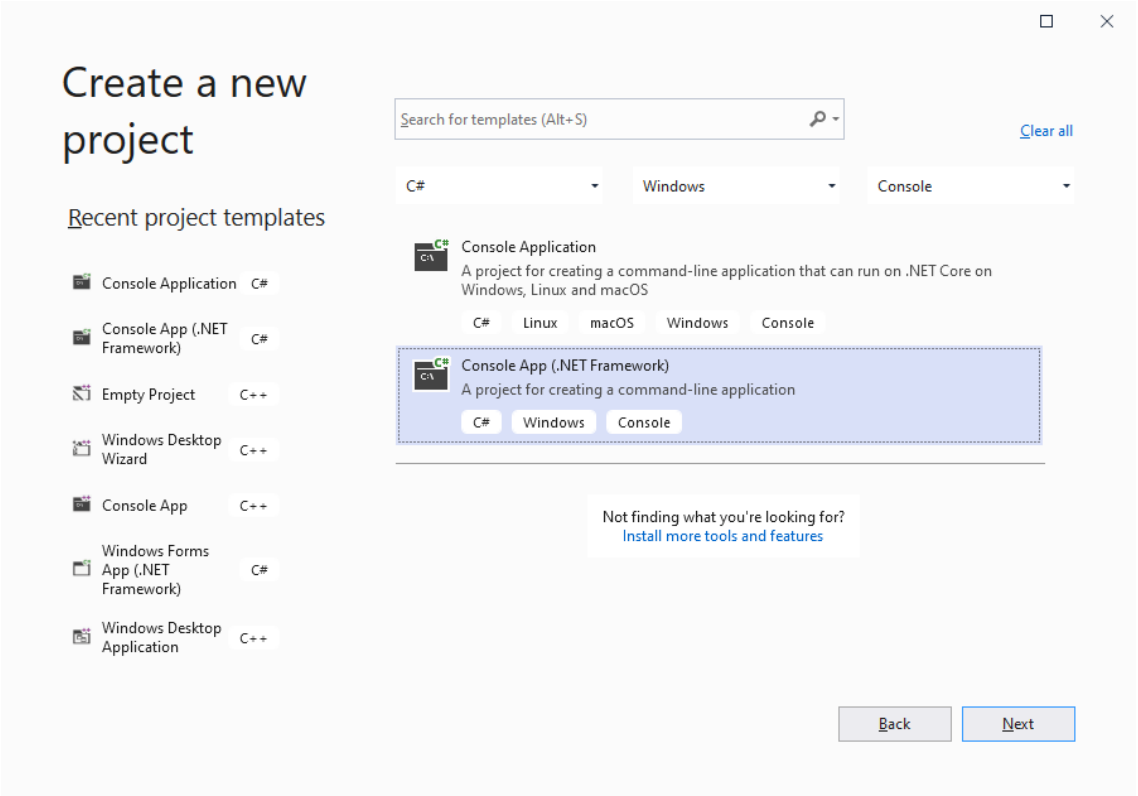
Limbajul C# a fost creat și dezvoltat de către firma Microsoft pentru a pune la dispoziția programatorilor un limbaj asemănător limbajului C++ dar mai simplu de folosit și mai sigur la rulare și, pe cât posibil, la fel de eficient. Limbajul a fost orientat din start către .NET, cadul actual de dezvoltare software al firmei Microsoft, apărut în același timp cu C#, în 2002.

Limbajul a fost creat de la zero de către o echipă condusă de Anders Hejlsberg, cel care în anii 80 ai secolului trecut a pus la punct pentru firma Borland compilatorul de mare succes Turbo Pascal și apoi, în anii 90, limbajul și mediul de programare Delphi, extensia orientată pe obiecte a limbajului Pascal. La firma Microsoft, înainte de a începe proiectul C#, Hejlsberg a fost implicat în dezvoltarea limbajului J#, o variantă a limbajului Java, abandonată de Microsoft în 2007. Cu o atât de largă experiență, el a reușit să proiecteze un limbaj nou, modern, care respectă strict principiile POO, și în care a preluat cele mai bune idei din C++, Java și Delphi. In plus, a dotat Visual Studio C# cu facilitățile de dezvoltare rapidă a aplicațiilor existente deja în Visual Basic și Borland Delphi.

Limbajul C# are, practic, sintaxa instrucțiunilor și a expresiilor din C/C++ dar nu este o extindere a limbajului C++, așa cu este C++ pentru C. Pentru cei care cunosc C++ trecerea la C# este foarte ușoară dacă se ține cont de deosebirile semnalate mai jos.

Pentru început o să inițiem un proiect C# de tip consolă în Visual Studio 2019.

Din fereastra de start alegem itemul `Create a new project` și apoi fixăm în cele trei căsuțe derulante limbajul `C#`, sistemul de operare `Windows` și tipul proiectului `Console`, și apoi alegem itemul `C# Console App(.NET Framework)`. Se poate alege și itemul `C# Console Application`, dar proiectele de acest tip, independente de platformă, create cu ediția 2019, nu pot fi deschise de ediții mai vechi de Visual Studio.



Apăsăm butonul `Next` și salvăm proiectul cu numele `CSharpTutorial`, de exemplu.

Proiectul nostru are un singur fișier de cod, Program.cs, și aici edităm următoarea varianta C# a programului Hello World, pe care s-ar putea să o găsim gata scrisă, măcar parțial:

```
using System;

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

Editarea, compilarea, link-editarea și rularea se lansează din aceleași meniuri ca în cazul unui proiect C++.

Listăm în continuare principalele deosebiri între C# și C++.

1. Structura programului. Un program C# este o colecție de definiții de clase și structuri, dintre care una trebuie să conțină *metoda* `Main()`. In C# nu se mai pot defini funcții și variabile globale în afara claselor, ca în C++, aici toate variabilele sunt *variabile membre* ale unei clase sau structuri, iar funcțiile sunt *funcții membre.* Pentru a evidenția această deosebire între cele două limbaje, în documentația C# funcțiile, care nu pot fi decât funcții membre, se numesc *metode* iar variabilele membre se numesc *câmpuri*.

Metoda `Main()` trebuie să fie declarată statică, ca să poată fi apelată de sistemul de operare fără instanțierea clasei în care apare. Vom reveni.

Numele clasei în care apare metoda `Main()` nu este obligatoriu `Program`, dar acesta este foarte potrivit.

2. Organizarea codului. Bibliotecile de clase sunt organizate în *spații de nume*, care pot fi imbricate, caz în care pentru a fi desemnate se utilizează "operatorul punct", exact ca la accesarea membrilor unei clase. De exemplu, `System.Windows.Forms.Form` desemnează clasa `Form` definită în spațiu de nume `System.Windows.Forms.`

Există un spațiu de nume implicit, care cuprinde toate clasele din biblioteca .NET, inclusiv cele definite de utilizator, și care poate fi accesat doar cu operatorul de rezoluție `::` întâlnit în C++, astfel

```
static void Main(string[] args)
 {
     global::System.Console.WriteLine("Hello World!");
 }
```

Aici este apelată metoda `WriteLine` a clasei `Console` definită în spațiul de nume `System` din .NET.

Pentru a avea acces la o clasă definită în exteriorul spațiului de nume curent fără să mai prefixăm numele clasei cu numele spațiului unde este definită, se folosește directiva `using`, ca în exemplul de mai sus. Fără directiva `using` programul nostru trebuie scris astfel:

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

După cum se vede, directiva `using` din C++

`using namespace std;`

are acum sintaxa simplificată

`using System;`

Directiva de precompilare

`#include<iostream>`

obligatorie în C++, aici nu mai este necesară, programatorul având acces direct la orice clasă publică din .NET.

Definirea unei clase în C# este mai simplă decât în C++, toate declarațiile și definițiile membrilor clasei se scriu direct între acoladele pereche care urmează numelui clasei, ca în exemplul de mai sus, unde în clasa `Program` este definită o singură metodă, metoda `Main()` .

3. Obiecte de tip referință vs. obiecte de tip valoare. Trecerea de la C la C++ a fost o revoluție în programare, ea a pus bazele programării orintate pe obiecte. Limbajul C++ este și astăzi lider în industria informatică, este foarte eficient dar are o problemă majoră: obligă programatorul să utilizeze alocarea dinamică a memoriei, ceea ce presupune cunoștințe aprofundate de programare și atenție maximă la alocarea/dealocarea memoriei. Erorile de alocare dinamică sunt foarte grave, apar numai la rulare, sunt ascunse și greu de depistat, și, de regulă, sunt fatale.

La trecerea de la C la C++, claselor li s-a păstrat *comportamentul de tip valoare* al structurilor din C, deoarece inițial compilatoarele de C++ le transformau în structuri C dotate cu pointeri către funcțiile membru. Prin urmare, în C++, simpla declarare a unei variabile de tip clasă alocă pe stivă spațiul necesar pentru toate variabilele membru (şi a pointerilor interni de acces la funcţiile membru) iar iniţializarea şi atribuirea obiectelor se execută prin copiere membru cu membru. O clasă care are ca membru un tablou de 1000 de `double` nu încape pe stivă, cel care proiectează clasa este nevoit să o doteze cu o variabilă membru de tip pointer către `double`, pe care să-l inițializeze în constructor cu adresa zonei de memorie alocate în heap cu operatorul `new` , zonă care trebuie dealocată cu operatorul `delete` în destructor. Mai mult, pentru a evita dealocarea multiplă, trebuie să supraîncarce obligatoriu operatorul de atribuire și să furnizeze un construcor de copiere. Un coșmar!

In C# alocarea dinamică se execută în fundal, automat, fără intervenția programatorului: obiectele de tip clasă pot fi create numai prin apelul explicit al unui constructor iar aceștia pot fi apelați numai cu operatorul `new`. Constructorul creează obiectul în *heap*, iar `new` returnează *o referință* către obiect, care poate fi a înțeleasă ca fiind adresa locației de memorie a obiectului creat.

```
using System;
```

```
namespace CSharpTutorial
{
    class Exemplu
    {
        public int i;            //camp
        public Exemplu(int i)    //constructor
        {
            this.i = i;
        }
        public void Print()     //metoda
        {
            Console.WriteLine(i);
        }
    }
    class ProgramExemplu
    {
        static void Main(string[] args)
        {
            Exemplu ex1, ex2;
            ex1= new Exemplu(123);
            ex1.Print();    //123
            ex2 = ex1;
            ex2.Print();    //123
            ex2.i = 666;
            ex2.Print();    //666
            ex1.Print();    //666
        }
    }
}
```

Aici am definit clasa `Exemplu` pe care am dotat-o cu câmpul `int i`, constructorul `Exemplu(int)` și metoda `Print()`. Declarația

```
Exemplu ex1, ex2;
```

provoacă numai alocarea pe stivă a două variabile de tip "*referință* către clasa `Exemplu`", care ocupă fiecare câte 32 de biți, cât este lungimea unei adrese de memorie.

În instrucțiunea

```
ex1= new Exemplu(123);
```

apelăm cu operatorul `new` constructorul clasei, care construiește în heap un *obiect*, o instanță a clasei `Exemplu`, iar adresa locației returnate de operatorul `new` o atribuim referinței `ex1`.

Cu atribuirea de referințe

```
ex2 = ex1;
```

copiem în `ex2` adresa obiectului referit de `ex1`, prin urmare apelul

```
ex2.Print();  //123
```

are rezultatul scontat.

În ultimile trei instrucțiuni

```
ex2.i = 666;
ex2.Print();  //666
ex1.Print();  //666
```

observăm cum modificarea câmpului `i` al obiectului din heap accesat prin referința `ex2` de pe stivă este văzută și prin referința `ex1`.

Am creat un "balaur cu două capete" cu corpul în memoria heap și capetele pe stivă. In C# acesta nu mai sperie pe nimeni: este dealocat automat la ieșirea din `Main()`, când dispar de pe stivă cele două capete, dar în C++ astfel de situații trebuie tratate cu multă grijă.

Am arătat mai sus că *referințele* din C# sunt de fapt *pointerii* din C/C++, ascunși sub o altă denumire deoarece au altă sintaxă față de pointerii propriu-ziși, care au fost păstrați dar care pot fi folosiți numai în blocuri de cod declarate cu cuvântul cheie `unsafe`! Să observăm și transformarea pointerului `this` din C++ în referința `this`, care se referă și aici tot la obiectul curent, dar are sintaxa simplificată `this.membru` în loc de `this->membru`.

În C# obiectele de tip clasă sunt alocate dinamic în mod automat, și tot automat sunt dealocate de către *colectorul de gunoaie*, un program care funcționează în fundal numărând, din când în când, câte referințe țintesc fiecare obiect curent și eliberează memoria ocupată de obiectele orfane, cele cu zero referințe. Se spune că obiectele de tip clasă sunt *obiecte de tip referință*.

In C# operatorul `new` acționează altfel decât în C++ dar rezolvă aceeași problemă, mult mai elegant, iar operatorul `delete` a dispărut deoarece nu mai este nevoie de el.

Reținem: în C# instanțierea unei clase poate fi efectuată numai cu apelul unui constructor prin operatorul `new`, care returnează o *constantă de tip referință* către obiectul alocat în heap. Prin această referință putem accesa câmpurile și metodele obiectului țintit și, de regulă, ea se păstrează prin atribuirea ei unei *variabile* de tipul clasei obiectului (sau de tipul unui strămoș).

Această automatizare a alocării dinamice din C++ a apărut prima dată în Java, apoi a fost preluată, parțial, în Delphi și VisualBasic, de aici a ajuns în C#.

Pentru completare, vezi
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types

Spre deosebire de Java, unde obiectele de tip structură au dispărut complet, în C# ele au fost păstrate cu tot cu comportamentul lor de *tip valoare* din C/C++.

Ca să exemplificăm diferența dintre clase și structuri în C# (amintim că în C++ nu este nici o diferență majoră) în programul precedent vom schimba doar `class Exemplu` cu `struct Exemplu.` Programul rulează fără probleme:

```
using System;
namespace CSharpTutorial
{
    //class Exemplu
    struct  Exemplu
    {
        public int i;            //camp
        public Exemplu(int i)    //constructor
        {
            this.i = i;
        }
        public void Print()     //metoda
        {
            Console.WriteLine(i);
        }
    }
    class ProgramExemplu
    {
        static void Main(string[] args)
        {
            Exemplu ex1, ex2;
            ex1= new Exemplu(123);
            ex1.Print();    //123
            ex2 = ex1;
            ex2.Print();    //123
            ex2.i = 666;
            ex2.Print();    //666
            ex1.Print();    //123 în loc de 666
        }
    }
}
```

Observăm că la rulare apare o singură diferență, la apelul din ultima linie de cod. Diferența e mică, dar esențială și se datorează alocării statice a structurilor. Mai precis, declarația

```
Exemplu ex1, ex2;
```
provoacă alocarea pe stivă a două *variabile de tip structură* care ocupă fiecare spațiul necesar cât să încapă un obiect de tip `Exemplu`.

În instrucțiunea
```
ex1 = new Exemplu(123);
```
apelăm cu `new` constructorul obiectului, care creează o instanță a structurii `Exemplu`. Rezultatul returnat de operatorul `new` este, în acest caz, chiar structura creată, pe care apoi operatorul de atribuire o depune, prin copiere membru cu membru, în locația de pe stivă a variabilei `ex1`.

Acum atribuirea
```
ex2 = ex1;
```
provoacă copierea în locația lui `ex2` a conținutului locației lui `ex1`, octet cu octet, prin urmare acum pe stivă avem două obiecte identice dar distincte, așa că atribuirea
```
ex2.i = 666;
```
îl modifică numai pe `ex2.i`, nu și pe `ex1.i`. după cum observam:
```
ex2.i = 666;
ex2.Print();  //666
ex1.Print();  //123
```
Deoarece structurile sunt alocate static exact cum sunt alocate variabile de tip aritmetic, iar atribuirea se realizează prin copierea întregului obiect, se spune că instanțe structurilor sunt *obiecte de tip valoare.*

Structurile fiind alocate pe stivă, sunt alocate/dealocate automat de ridicarea/coborârea stivei, ele nu necesită alocare dinamică și nu sunt gestionate de colectorul de gunoaie. Totuși, pentru a nu umple rapid stiva ele trebuie să fie de mici dimensiuni și, din acest motiv, în C#, structurile nu pot fi extinse prin derivare.

Menționăm în final că obiectele de tip `struct` sunt de fapt obiecte de tip clasă alocate static, ele provin clasa `System.ValueType` care la rândul ei provine direct din `System.Object`.

Vezi https://docs.microsoft.com/en-us/dotnet/api/system.valuetype?view=net-6.0

4. Proprietăți. Un principiu de bază în POO este *încapsularea* informației în obiecte, realizată, în principal, prin specificarea pentru fiecare membru, explicit sau implicit, a unui anumit grad de acces către el din exteriorul clasei.

În C# gradele extreme sunt `public` – accesibil de oriunde este accesibilă clasa – și `private` – accesibil numai din interiorul clasei. Pentru ceilalți modificatori de acces, vezi https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

Pentru protejarea structurii interne a unui obiect, câmpurile claselor sunt declarate, de regulă, de tip privat și sunt accesate în C++ prin funcții publice de acces iar în C# prin *proprietăți*.

Să studiem următoarea definiție a clasei `Carte` pe care am dotat-o numai cu două câmpuri, `titlu` de tip string și `pret` de tip întreg.

Câmpul `titlu` l-am declarat privat deoarece dorim ca titlul fixat la crearea obiectului să nu mai poată fi schimbat, dar să poată fi citit.

Câmpul `pret` l-am declarat privat deoarece vrem ca în interiorul clasei să stocăm prețul în bani, deci într-o variabilă de tip întreg, dar utilizatorul să furnizeze și să primescă prețul în lei, deci printr-o valoare de tip double.

Avem implementate aici două rezolvări: una în stil C++, cu *metodele de acces* `getTitlu()`, `getPret()` și `setPret()` și cealaltă, în stil C#, cu *proprietățile* publice `Titlu` și `Pret`.

```
using System;
```

```
namespace CSharpTutorial
{
    class Carte
    {
        private string titlu;
        private int pret;

        public Carte(string ttl)
        {
            titlu = ttl;
            pret = 0;
        }

        //metode de acces publice, stil C++
        public string getTitlu()
        {
            return titlu;
        }
        public double getPret()
        {
            return pret * 0.01;
        }
        public void setPret(double prt)
        {
            if (prt < 0) pret = 0;
            else pret = (int)Math.Round(prt * 100);
        }
        //proprietati publice, stil C#
        public string Titlu
        {
            get
            {
                return titlu;
            }
        }

        public double Pret
        {
            get
            {
                return pret * 0.01;
            }
            set
            {
                if (value < 0) pret = 0;
                else pret = (int)Math.Round(value * 100);
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Carte crt = new Carte("Tutorial C\x23");
            crt.setPret(12.34);
            Console.WriteLine(crt.getTitlu() + " costa " + crt.getPret() + " lei");
            crt.Pret = 43.21;
            Console.WriteLine(crt.Titlu + " costa " + crt.Pret + " lei");
        }
    }
    //pe monitor:
    //Tutorial C# costa 12.34 lei
```

```
    //Tutorial C# costa 43.21 lei
    //Press any key to continue . . .
}
```

În C# un obiect poate fi dotat cu câmpuri, metode și proprietăți. O proprietate desemnează cu un singur nume o pereche de metode de acces get/set, sau numai una, care sintactic se comportă ca fiind numele unui câmp dar care, la compilare, este înlocuit cu apelul uneia dintre metodele get/set, în mod adecvat.

Mai precis, la citirea valorii proprietății numele ei este înlocuit cu apelul accesorului său `get`. De exemplu, dacă notăm cu `Pret_get`, accesorul `get` al proprietății `Pret`, atribuirea

```
double prt = crt.Pret;
```

este transformată de compilator în

```
double prt = crt.Pret_get();
```

Analog, la setarea valorii proprietății este apelat accesorul `set`. De exemplu, dacă notam cu `Pret_set` acest accesor al lui `Pret`, atribuirea

```
crt.Pret = 43.21;
```

este transformată de compilator în apelul

```
crt.Pret_set(43.21);
```

Observație: cuvântul cheie `value` care apare în definiția unui accesor `set` desemnează parametrul primit de acesta la apel, parametru care are tipul proprietății. În cazul proprietății `Pret`, `value` are tipul `double`.

Stilul corect de programare în C# prevede ca la proiectarea unei clase sau structuri, toate câmpurile să fie declarate private, urmând ca accesul la ele să fie permis, dacă este nevoie, numai prin proprietăți publice.

Pentru mai multe informații, vezi
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties

5. Static, non-static. Analog limbajului C++, un câmp membru al unei clase C# poate fi declarat static, caz în care el este alocat unic la nivelul clasei, valoarea sa fiind aceeași pentru toate instanțele clasei.

În C# un câmp static poate fi accesat numai prin numele clasei:

```
using System;
namespace CSharpTutorial
{
    class Exemplu
    {
        static public int Nr;
        public Exemplu(int n)
        {
            Nr = n;
        }
        public void Print()
        {
            Console.WriteLine("Nr=" + Nr);
        }

    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            Exemplu ex1 = new Exemplu(1000);
            //Console.WriteLine(ex1.Nr);//eroare de compilare
            Console.WriteLine(Exemplu.Nr);  // 1000
            ex1.Print();                    // Nr=1000;
            Exemplu ex2 = new Exemplu(2000);
            Console.WriteLine(Exemplu.Nr);  // 2000
            ex2.Print();                    // Nr=2000
            ex1.Print();                    // Nr=2000
        }
    }
}
```

Observăm că cele două apeluri `ex1.Print()` au rezultate diferite, deși între ele nu mai apare în cod referința `ex1`.

Câmpurile statice pot fi apelate fără nici o instanțiere a clasei, caz în care, dacă nu au fost inițializate la definirea clasei, au valoarea nulă implicită.

Analog, metodele declarate statice se apelează numai cu numele clasei și sunt folosite, de regulă, fără nici o instanțiere a clasei.

```
using System;
namespace CSharpTutorial
{
    class MyMath
    {
        static public double  Triplu(double x)
        {
            return 3 * x;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            double x = MyMath.Triplu(10.0);
            Console.WriteLine("x=" + x); // x=30
        }
    }
}
```

Dacă nu am fi declarat mai sus metoda `Triplu()` ca fiind statică, a fi fost nevoiți să o apelăm printr-o instanță a clasei, care în acest caz nu modifică cu nimic rezultatul:

```
using System;
namespace CSharpTutorial
{
    class MyMath
    {
        public double  Triplu(double x)
        {
            return 3 * x;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
```

```
            MyMath mm = new MyMath();
            double x = mm.Triplu(10.0);
            Console.WriteLine("x=" + x);
        }
    }
}
```

La definirea unei clase, metodele declarate statice pot folosi, în corpul lor, numai membrii statici ai clasei. Link: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

6. Derivarea claselor. Exact ca în C++, având gata definită o clasă, numită *clasa bază*, putem să definim o altă clasă, *clasa derivată*, care să conțină toți membrii clasei bază și, pe lângă aceștia, să adauge eventual noi membri. Derivarea se semnalează tot cu simbolul : .

```
using System;
namespace CSharpTutorial
{
    class ClasaBaza
    {
        public void ScrieBaza()
        {
            Console.WriteLine("CLASA BAZA");
        }
    }
    class ClasaDeriv : ClasaBaza
    {
        public void ScrieDeriv()
        {
            Console.WriteLine("CLASA DERIVATA");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            ClasaDeriv refDeriv = new ClasaDeriv();
            refDeriv.ScrieBaza();  //CLASA BAZA
            refDeriv.ScrieDeriv(); //CLASA DERIVATA

            ClasaBaza refBaza = new ClasaBaza();
            refBaza.ScrieBaza(); //CLASA BAZA
            //refBaza.ScrieDeriv(); //eroare de compilare

            refBaza = refDeriv; //ok
            //refDeriv = refBaza; //eroare de compilare
            //refDeriv = new ClasaBaza();//eroare de compilare

            ClasaBaza refBazaToDeriv = new ClasaDeriv(); //ok
            //refBazaToDeriv.ScrieDeriv(); //eroare de compilare
        }
    }

}
```

O atribuire între două referințe este admisă numai dacă au același tip sau există o conversie, implicită sau explicită, de la tipul din dreapta la cel din stânga. Orice clasă este impli-

cit convertită la oricare dintre strămoșii săi: pur și simplu nu mai sunt luați în considerare membrii adăugați prin derivare.

In urma declarației cu inițialezare

```
ClasaBaza refBazaToDeriv = new ClasaDeriv(); //ok
```

spunem că referința `refBazaToDeriv` are tipul declarat `ClasaBaza` și tipul acual `ClasaDeriv`. Compilatorul ține cont numai de tipul declarat al referinței, prin urmare permite accesul doar la membrii tipului declarat, tipul actual manifestându-se numai la rulare:

```
refBazaToDeriv.ScrieDeriv(); //eroare de compilare
```

7. Suprascrierea/ascunderea metodelor. La derivarea unei clase este posibil să redefinim o metodă moștenită, prin folosirea în clasa derivată a aceluiași nume și a acelorași parametri.

Dacă folosim același nume dar cu altă listă de parametri, nu facem altceva decât să *supraîncărcăm* metoda la nivelul clasei derivate, caz în care această formă a metodei poate fi accesată numai printr-o referință de tipul clasei derivate (către un obiect din clasa derivată, desigur!).

In exemplul următor, metoda `Scrie()` este supraîncărcată atât la nivelul clasei bază, cât și la derivare:

```
using System;
namespace CSharpTutorial
{
   class CBaza
    {
        public void Scrie()
        {
            Console.WriteLine("text scris de Scrie()");
        }
        public void Scrie(string msg)
        {
            Console.WriteLine(msg+ " scris de Scrie(string)");
        }
    }
    class CDeriv:CBaza
    {
        public void Scrie(int n)
        {
            Console.WriteLine("Scris de Scrie(int) in anul {0}", n);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            CDeriv refDerivToDeriv = new CDeriv();
            refDerivToDeriv.Scrie();
            refDerivToDeriv.Scrie("Mesaj");
            refDerivToDeriv.Scrie(2222);

            CBaza refBazaToBaza = new CBaza();
            refBazaToBaza.Scrie();
            refDerivToDeriv.Scrie("Mesaj");
            //refBazaToBaza.Scrie(2222);//eroare

            CBaza refBazaToDeriv = new CDeriv();
            refBazaToDeriv.Scrie();
            refDerivToDeriv.Scrie("Mesaj");
            //refBazaToDeriv.Scrie(2222);//eroare
        }
    }
}
```

Eroarea de compilare la apelul `refBazaToDeriv.Scrie(2222)` apare deoarece compilatorul ține cont numai de tipul declarat al referinței și nu de tipul actual al obiectului țintit de ea. Referința `refBazaToDeriv` este de tip `CBaza` care nu are definită metoda cu semnătura `Scrie(int)`.

Dacă la derivare încercăm să redefinim o metodă moștenită folosind aceeași semnătură (nume și parametri) compilatorul ne obligă ori să *ascundem* metoda veche ori sau să o *suprascriem*.

Ascunderea metodei vechi se semnalează numai în clasa derivată, folosind cuvântul cheie `new` astfel:

```
using System;
namespace CSharpTutorial
{
    class CBaza
    {
        public   string FormeazaString()
        {
            return "TEXT de BAZA ";
        }

        public void Print()
        {
            Console.WriteLine(FormeazaString() +"scris prin Print de
"+this.ToString());
        }
    }


    class CDeriv:CBaza
    {
        public  new string FormeazaString()
        {
            return "TEXT DERIVAT ";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            CBaza refBazaToBaza = new CBaza();
            refBazaToBaza.Print();
            Console.WriteLine(refBazaToBaza.FormeazaString() + "scris prin apelare din
refBazaToBaza");

            CDeriv refDerivToDeriv = new CDeriv();
            refDerivToDeriv.Print();
            Console.WriteLine(refDerivToDeriv.FormeazaString() + "scris prin apelare
din refDerivToDeriv");

            CBaza refBazaToDeriv = new CDeriv();
            refBazaToDeriv.Print();
            Console.WriteLine(refBazaToDeriv.FormeazaString() + "scris prin apelare
din refBazaToDeriv");

        }
    }
   //pe monitor:
```

```
   //TEXT de BAZA scris prin Print de CSharpTutorial.CBaza
   //TEXT de BAZA scris prin apelare din refBazaToBaza
   //TEXT de BAZA scris prin Print de CSharpTutorial.CDeriv
   //TEXT DERIVAT scris prin apelare din refDerivToDeriv
   //TEXT de BAZA scris prin Print de CSharpTutorial.CDeriv
   //TEXT de BAZA scris prin apelare din refBazaToDeriv
   //Press any key to continue . . .

}
```

Observăm că noul comportament al metodei `FormeazaString()` se manifestă numai când metoda este apelată direct printr-un obiect de tip derivat accesat printr-o referință de tip derivat. În toate celelalte situații metoda are comportamentul inițial, cel ascuns la derivare.

Suprascrierea este o tehnică esențială în POO, foarte des folosită și foarte eficientă. Ea este permisă numai dacă în clasa de bază metoda a fost declarată *virtuală*, cu `virtual`. In acest caz metoda poate fi suprascrisă în clasa derivată, folosind cuvântul cheie `override` astfel:

```
using System;
namespace CSharpTutorial
{
    class CBaza
    {
        public   virtual string FormeazaString()
        {
            return "TEXT de BAZA ";
        }
        public void Print()
        {
            Console.WriteLine(FormeazaString() +"scris prin Print de
"+this.ToString());
        }
    }
    class CDeriv:CBaza
    {
        public override string FormeazaString()
        {
            return "TEXT DERIVAT ";
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            CBaza refBazaToBaza = new CBaza();
            refBazaToBaza.Print();
            Console.WriteLine(refBazaToBaza.FormeazaString() + "scris prin apelare din
refBazaToBaza");

            CDeriv refDerivToDeriv = new CDeriv();
            refDerivToDeriv.Print();
            Console.WriteLine(refDerivToDeriv.FormeazaString() + "scris prin apelare
din refDerivToDeriv");

            CBaza refBazaToDeriv = new CDeriv();
            refBazaToDeriv.Print();
            Console.WriteLine(refBazaToDeriv.FormeazaString() + "scris prin apelare
din refBazaToDeriv");

        }
    }
    //pe monitor:
```

```
    //TEXT de BAZA scris prin Print de CSharpTutorial.CBaza
    //TEXT de BAZA scris prin apelare din refBazaToBaza
    //TEXT DERIVAT scris prin Print de CSharpTutorial.CDeriv
    //TEXT DERIVAT scris prin apelare din refDerivToDeriv
    //TEXT DERIVAT scris prin Print de CSharpTutorial.CDeriv
    //TEXT DERIVAT scris prin apelare din refBazaToDeriv
    //Press any key to continue . . .


}
```

Observăm că acum comportamentul vechi al metodei suprascrise se manifestă numai când este apelată direct sau indirect (prin metoda `Print()`) printr-o referință de tipul bazei către un obiect de bază, în toate celelalte situații se manifestă comportamentul definit la derivare.

Explicația constă în faptul că metodele obișnuite sunt legate direct de obiectul curent (pointerii către metodă din corpul obiectului țintesc direct către codul metodei) în timp ce metodele virtuale sunt legate indirect de acesta, prin intermediul unui "tabel al metodelor virtuale" asociat clasei, care este actualizat la fiecare derivare.

Reținem: comportamentul unei metode suprascrise este cel definit în clasa obiectului actual, indiferent de tipul referinței prin care este accesat obiectul.

8. Clasa Object. In C# nu este permisă moștenirea multiplă, cum este cazul în C++, aici toate clasele au o singură clasă părinte. Singura excepție este dată de clasa `System.Object` care este rădăcina arborelui relației de derivare a claselor, prin urmare orice clasă și structură C# moștenește direct sau indirect clasa `System.Object`. Această clasă poate fi desemnată în codul sursă și cu cuvântul cheie `object`, un alias al clasei `System.Object`.

Dintre metodele moștenite de la `System.Object`, cea mai des folosită este `ToString()`, care returnează un mesaj ce conține informații despre obiectul curent. Metoda este virtuală și poate fi suprascrisă, altfel returnează numai numele clasei obiectului curent.

```
using System;
namespace CSharpTutorial
{
    class Exemplu
    {
    }

    class ExtraExemplu: Exemplu
    {
        string text;
        public ExtraExemplu(string  msg)
        {
            text = msg;
        }
        public override string ToString()
        {
            return  text;
        }
    }


    class Program
    {
        static void Main(string[] args)
```

```
        {
            Exemplu a = new Exemplu();
            Console.WriteLine(a.ToString());    //CSharpTutorial.Exemplu
            Console.WriteLine(a);    //CSharpTutorial.Exemplu

            ExtraExemplu aaa= new ExtraExemplu("AHA!");
            Console.WriteLine(aaa);  //AHA!
            string mesaj = "mesaj: " + aaa;
            Console.WriteLine(mesaj); //mesaj: AHA!
        }
    }
}
```

Mai sus am definit clasa `Exemplu` în aparență vidă, totusi ea are un constructor implicit, `Exemplu()`, și moștenește direct clasa `System.Object`.

Observăm că `a` este dotată cu metoda `ToString()` moștenită de la `System.Object`, care în acest caz returnează numele clasei obiectului curent.

La apelul

```
Console.WriteLine(a);
```

este apelată metoda

```
void System.Console.WriteLine(object value);
```

care scrie pe monitor stringul returnat de metoda `value.ToString()`.

Aici

```
string mesaj = "mesaj: " + aaa;
```

observăm cuvântul cheie `string` care desemnează clasa `System.String` în care, după cum se vede, operatorul de adunare a fost suprascris astfel încât adunarea unui string cu un obiect concatenează stringul dat cu stringul returnat de metoda `ToString()` a obiectului.

O referință de tip `object` poate avea orice tip actual, adică poate ținti obiecte din orice clasă, dar o instanță `object` poate fi țintită numai de o referință `object`.

9. Calculul aritmetic. Tipurile aritmetice fundamentale `char`, `int`, `double`, se regăsesc și în C# , sunt alocate static și se comportă exact ca în C++, dar trebuie menționat că în C# variabilele de acest tip sunt transformate în mod automat, atunci când este nevoie, în obiecte de tip `System.Object`, printr-un mecanism numit *boxing/unboxing*.

```
using System;
namespace CSharpTutorial
{
    class Program
    {
        static void Main()
        {
            int a = 13;
            object ob = a;  //boxing;
            Console.WriteLine("ob=" + ob);//ob=13;
            int b =  (int)ob;//unboxing
            Console.WriteLine("b="+b); //b=13
            string mesaj = a.ToString(); //boxing
            Console.WriteLine("a=" + mesaj);//a=13
        }
    }
}
```

10. Calculul logic. In C# tipul logic `bool` este complet separat de tipul `int`, nu mai există conversia implicită de la zero la `false` și de la nenul la `true`, împiedicând de exemplu programatorul să mai confunde atribuirea cu comparația, greșeală frecventă în C/C++.

In C#, spre deosebire de C++, următorul cod nu trece de faza de compilare

```
bool testDeEgalitate(int a, int b)
{
    if (a = b) return true;
    else return false;
}
```

Primim mesajul de eroare `Cannot implicitly convert type 'int' to 'bool'`, eroare care nu apare și în C++ unde codul de mai sus este compilat fără probleme, dar în acest caz un apel de forma `testDeEgalitate(13, 15)` returnează rezultatul `true`! De ce?

11. Tablouri. În limbajul C# nu mai există noțiunea de tablou din C/C++, definită ca o locație de memorie cu adrese succesive, în care sunt stocate un șir de elemente toate de același tip. În locul acestora în C# se folosesc obiecte de tip `System.Array`, pentru care limbajul este prevăzut cu o sintaxă asemănătoare tablourilor din C/C++. În exemplul următor în tabloul `tab` sunt încărcate primile `dim` puteri ale lui 2.

```
static void Main()
{
    int dim = 12;
    int[] tab = new int[dim];
    tab[0] = 1;
    for (int i = 1; i < tab.Length; i++)
    {
        tab[i] = 2 * tab[i - 1];
        Console.WriteLine(tab[i]);
    }
    Console.WriteLine("Din nou:");
    foreach ( int p in tab)
    {
        Console.WriteLine(p);
    }
}
```

Am exemplificat mai sus și parcurgerea cu `foreach` a unui tablou, vezi
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements
și, pentru `System.Array`, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/

12. Siruri de caractere. Tipul `string` deja utilizat în exemplele mai sus este un alias al clasei `System.String`. Menționăm că nici în C# nu există tipul fundamental *șir de caractere* ci numai clase de stringuri. Constantele de tip string de forma `"acesta este un mesaj"` pot fi atribuite oricărui obiect de tip `string` iar limbajul dispune de mai multe metode de scriere a acestor constante.

Vezi https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/

Reținem că în clasa `System.String` operatorul de adunare `+` a fost *supraîncărcat* astfel încât să concateneze stringurile.

13. Supraîncărcarea operatorilor. În C# este permisă supraîncărcarea operatorilor dar, spre deosebire de C++ nu toți operatorii pot fi supraîncărcați, una din excepțiile notabile fiind operatorul de atribuire. Este bine de știut că în limbajul Java, care provine și el din C++, nu este permisă supraîncărcarea operatorilor.

Mai jos am definit o structură pentru lucrul cu fracții numerice, dotată cu două câmpuri întregi, numărătorul `int a` și numitorul `int b`, și cu un câmp logic, `bool isNaN`, care să semnalizeze dacă numitorul e nul sau nu.

```
using System;
namespace CSharpTutorial
{
    struct FractieApeB
    {
        //Campuri
        private int a, b;  // fractia = a/b
        private bool isNaN;// NaN = Not a Number

        //Metode
        public FractieApeB(int aa, int bb)
        {
            a = aa;
            b = bb;
            isNaN = (bb == 0);
        }
        public override string ToString()
        {
            if (isNaN) return string.Format("fractie invalida");
            if (a == 0) return string.Format("0");
            if (a * b > 0) return string.Format("{0}/{1}", Math.Abs(a), Math.Abs(b));
            return string.Format("-{0}/{1}", Math.Abs(a), Math.Abs(b));
        }
        public void Print(String mesaj)
        {
            Console.WriteLine(mesaj + this.ToString());
        }

        //Proprietati
        public int A
        {
            get
            {
                return a;
            }
            set
            {
                a = value;
            }
        }
        public int B
        {
            get
            {
                return b;
            }
            set
            {
                b = value;
                isNaN = (b == 0);
            }
        }
```

```
    //Spupraincarcarea operatorilor

    public static FractieApeB operator +(FractieApeB p, FractieApeB q)
    {
        if (p.isNaN || q.isNaN) return new FractieApeB(0, 0);
        return new FractieApeB(p.a * q.b + q.a * p.b, p.b * q.b);
    }
    public static FractieApeB operator -(FractieApeB p, FractieApeB q)
    {
        if (p.isNaN || q.isNaN) return new FractieApeB(0, 0);
        return new FractieApeB(p.a * q.b - q.a * p.b, p.b * q.b);
    }
    public static FractieApeB operator *(FractieApeB p, FractieApeB q)
    {
        if (p.isNaN || q.isNaN) return new FractieApeB(0, 0);
        return new FractieApeB(p.a * q.a, p.b * q.b);
    }

    public static FractieApeB operator /(FractieApeB p, FractieApeB q)
    {
        if (p.isNaN || q.isNaN) return new FractieApeB(0, 0);
        return new FractieApeB(p.a * q.b, p.b * q.a);
    }

    public static bool operator ==(FractieApeB p, FractieApeB q)
    {
        if (p.isNaN || q.isNaN) return false;
        return p.a * q.b == q.a * p.b;
    }
    public static bool operator !=(FractieApeB p, FractieApeB q)
    {
        if (p.isNaN || q.isNaN) return false;
        return p.a * q.b != q.a * p.b;
    }

    //conversie int -> fractie
    public static implicit operator FractieApeB(int a)
    {
        return new FractieApeB(a, 1);
    }

    //conversie fractie -> double
    public static implicit operator double(FractieApeB f)
    {
        return f.a / (double)f.b;
    }

    //suprascrierea unor metode mostenite de la Sistem.Object
    public override bool Equals(object o)
    {
        if (o.GetType() != this.GetType()) return false;
        else return this == (FractieApeB)o;
    }

    public override int GetHashCode()
    {
        return a.GetHashCode() + b.GetHashCode();
    }
}
```

```
    class Program
    {
        static void Main(string[] args)
        {

            FractieApeB q = new FractieApeB(1, -2);
            q.Print("q=");  // q = -1 / 2

            q.B = 0;
            q.Print("q=");  // q = fractie invalida

            //operatii si conversii:
            FractieApeB f = new FractieApeB(2, 3), g = new FractieApeB(4, 7);
            q = (3 + f + g) / (f - g);
            q.Print("q ="); // q = 1869 / 42
            double qq = q;
            Console.WriteLine("q=" + q.A / (double)q.B + " qq=" + qq);
            //q = 44.5 qq = 44.5

            //comparatii:
            FractieApeB h = new FractieApeB(3, 5), hh = new FractieApeB(33, 55);
            if (h == hh) Console.WriteLine("sunt egale");
            else Console.WriteLine("nu sunt egale");
            //sunt egale

            Object ob = new FractieApeB(333, 555);
            if (ob.Equals(h)) Console.WriteLine("sunt egale");
            else Console.WriteLine("nu sunt egale");
            //sunt egale

            //test:
            Console.WriteLine(new FractieApeB((new FractieApeB(1, 2)).A, 3));
            Console.WriteLine((object)new FractieApeB((new FractieApeB(1,2)).A, 3));
        }
    }
}
```

Observăm că în structura `FractieApeB` au fost supraîncărcați operatorii aritmetici și operatorii de comparație, și au fost definite conversii implicite de la `int` la `FractieApeB` și de la `FractieApeB` la `double`.

Ce rezultate au ultimele două apeluri ale metodei `Console.WriteLine` și cum pot fie explicate?