©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

CURS ELEMENTAR DE PROGRAMARE IN LIMBAJUL C#

1. Prelucrarea informatiei de catre sistemul de operare Microsoft Windows
2. Functii sau metode. Clase. Structuri. Interfete. Spatii de nume. Proprietati si accesori
- **2.1**. Functii sau metode. Delegari
- **2.2**. Clase
- **2.3**. Structuri
- **2.4**. Interfete
- **2.5**. Spatii de nume
- **2.6**. Proprietati si accesori

3. Structura generica a unui program scris in limbajul C#
4. Tipuri de programe executabile. Biblioteci de clase si de controale
5. Variabile si constante
- **5.1**. Tipuri de valori si conversii
- **5.2**. Denumirea variabilelor
- **5.3**. Conventii de nume pentru variabile
- **5.4**. Gradul sau nivelul de accesibilitate
- **5.5**. Constante

6. Tablouri de date. Enumerari. Alte tipuri de colectii de date, specifice platformei Microsoft .NET
- **6.1**. Tablouri
- **6.2**. Enumerari
- **6.3**. Tipuri de colectii de date specifice platformei .NET

7. Iteratii
- **7.1**. Perechea do-while
- **7.2**. Ciclul for
- **7.3**. Ciclul foreach

8. Instructiuni de evaluare si control - blocul if-else; blocul switch
- 8.1. Blocul if-else
- 8.2. Blocul switch

9. Operatori
- **9.1**. Prezentare generala
- **9.2**. Precedenta operatorilor (ordinea operatiunilor)
- **9.3**. Supraincarcarea (+) operatorilor

10. Instructiuni
11. Tratarea erorilor
12. Prezentarea unor obiecte si controale esentiale din platforma .NET si a principalilor membri ai acestora
13. Fisiere
14. Resurse
15. Serializarea si deserializarea obiectelor
16. Dataset-uri (Seturi de date). Gestionarea datelor cu SQL Server
- **16.1**. Dataset-uri
- **16.2**. Gestionarea datelor cu Microsoft SQL Server

17. Clasa System.Math
18. Referinte
19. Interfata mediului de programare Microsoft Visual Studio .NET Express Edition
20. Reguli fara de care nu se poate crea un program
21. Unelte asociate platformei .NET
22. Medii de programare in Visual C#
23. Navigarea in mediul de programare Microsoft Visual C# Express

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

CURS ELEMENTAR DE PROGRAMARE IN VISUAL C#

1. PRELUCRAREA INFORMATIEI DE CATRE SISTEMUL DE OPERARE MICROSOFT WINDOWS

Prin informatie in sistemul de operare se intelege, in mod generic, orice mesaj interpretabil de respectiva platforma. Exemple de mesaje comune sunt: click-ul stinga/dreapta pe un buton, comanda de minimizare a unei ferestre, apasarea tastei F2 pentru redenumirea unui fisier cu programul Windows Explorer.

Mesajele au, desigur, o destinatie. Acesta este, de obicei, fereastra (*window*) care primeste mesajul si anunta sistemul ca acesta trebuie prelucrat.

Sa luam spre exemplu mesajul BM_CLICK care anunta sistemul ca a fost trimisa comanda click-stinga spre un buton (ButtonMessage-Click).

La pornirea sistemului si a oricarei aplicatii software Windows, fiecare fereastra (de exemplu casuta de log-on, butoanele si casetele de text ce le contine, butonul *Start* etc.) primeste un **identificator** (*handle*). Acesta va fi cunoscut in sistem drept identificatorul sau eticheta respectivei ferestre, iar orice mesaje primite de fereastra vor fi trimise sistemului odata cu acesta. Asadar, BM_CLICK primit de butonul *Start* va desfasura meniul asociat acestuia, si nu va duce la executia click-ului stanga pe nici un alt buton (decat in cazuri in care sistemul este corupt de virusi sau modificat in alt fel).

Mesajul BM_CLICK este un mesaj compus. Butonul primeste, de fapt, asociatia de mesaje WM_LBUTTONDOWN si WM_LBUTTONUP, iar fereastra care il contine primeste mesajul BN_CLICKED.

Mesajele de tip xN_ sunt de *notificare*, adica mesaje a caror executie se poate modifica incat sa declanseze un alt lant de evenimente decat cel asteptat in mod generic, prin definitie. Executia mesajelor in general poate fi modificata; de exemplu, interceptarea notificarii BN_CLICKED poate genera maximizarea sau inchiderea ferestrei care contine butonul, sau orice alta functie.

In cele ce urmeaza, termenul *fereastra* sau *window* va fi inlocuit de termenul *control* (cand ne referim la componente ale unei interfete) sau *formular* (cand ne referim la interfata unui program).

Asadar, avem perechea mesaj-identificator de control; o consecinta imediata, intentionata, a acesteia este faptul ca pe ecran exista un singur pointer de mouse si un singur cursor de text, astfel incat mesajele sa fie prelucrate **pe rand** pe masura ce sunt lansate in sistem (apasare de tasta, miscare a pointerului, click-dreapta, click-stanga, miscarea rotii mouse-ului etc.).

Un mesaj nu poate fi trimis simultan mai multor controale vizuale.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Se spune ca, asa cum este cunoscut in sistemul de operare, mesajul **BM_CLICK** este o *constanta*.

Constantele si ***variabilele*** pot fi privite ca marimi fizice in sistemul de operare. Valoarea constantelor specificate in fisierele ce asigura functionarea **sistemului** ramane aceeasi de la pornire pana la inchidere si este aceeasi intre pornirile computerului, fiind esentiala pentru functionarea corecta a sistemului de operare. BM_CLICK va avea mereu valoarea 245; aceasta valoare este stocata intr-o *biblioteca de legaturi dinamice* (Dynamic Link Library, sau DLL) a sistemului de operare, impreuna cu alte constante si functii care asigura functionarea acestuia si interactiunea cu utilizatorul.

(Notiunea de **functie** sau **metoda** va fi definita in capitolul 2)

Bibliotecile esentiale pentru functionarea minimala a unei platforme Windows 32-bit (Windows 95, 98, ME, NT, 2000, XP, Vista) sunt: kernel32.dll, user32.dll si gdi32.dll. Indiferent de particula "32", numele lor este sugestiv pentru faptul ca ele contin, respectiv, definitii pentru functii si constante necesare functionarii sistemului de operare, interceptarii si prelucrarii actiunilor utilizatorului si functii pentru sustinerea interfetei grafice.

Continutul acestor biblioteci poate fi accesat dintr-un mediu de programare, dar *numai pentru citire*. Cu ajutorul lor, un program poate efectua operatiuni simple si complexe, de la modificarea textului de pe un buton, crearea sau ascunderea unei ferestre de program sau unui control, pana la administrarea legaturii la Internet etc.

Limbajele de programare asa-numite *de nivel inalt* asigura interactiunea cu sistemul de operare intr-un mod usor de invatat, bazat in mod obisnuit/clasic pe engleza simpla. Daca nu ar exista limbaje orientate spre invatarea usoara, un programator ar trebui sa invete, pe langa mesajul BM_CLICK, alte 100 mesaje de baza potrivite pentru circa 300 functii avand nume nu intotdeauna intuitive.

Aceasta sarcina o au, de obicei, programatorii in Visual C++/MFC. Limbaje mai accesibile, care pot indeplini aceleasi sarcini ca C++, precum Visual Basic si Visual C# (*si:-sharp* sau *si:-diez*) pun la dispozitia programatorului functii generice a caror apelare genereaza de fapt apelarea functiilor stocate in bibliotecile de sistem. De exemplu, modificarea textului unui buton denumit "bcom" se face, in C++, apeland functia *SetWindowText* careia i se furnizeaza identifcatorul butonului si textul dorit. In Visual Basic, respectiva linie de comanda (care in C++ era posibil mai lunga de 50 caractere), este inlocuita cu comanda **bcom.Caption= "text"**; in C#, comanda este **bcom.Text= "text"**.

Aici am aflat deja caracteristica de baza a limbajelor moderne, aceea de a referi un obiect (control grafic sau alta entitate informationala) dupa numele sau si nu neaparat dupa identificator (*handle*).

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

In ceea ce priveste limbajele de programare in general, acestea pot fi clasificate cel mai simplu dupa metodologia de executie a programelor rezultate:

1. Limbaje non-structurate

Aceasta categorie cuprinde unele dintre primele limbaje de programare, dintre care unele supravietuiesc cel putin prin nume, exemplele clasice fiind FORTRAN (FORmula TRANslator) 77 si primele versiuni ale COBOL.

Caracteristica definitorie a acestor limbaje este aceea ca instructiunile se executa una dupa alta, in ordinea asezarii lor in codul programului. Desi exista numai cate o instructiune pe linie, sunt permise salturi la linii identificate prin "etichete". De asemenea, sunt permise grupuri de instructiuni care conduc la un scop anume, aranjate in *subrutine*.

Aceste limbaje duc intotdeauna la crearea unor programe cu cod "spaghetti", adica intreg codul se afla intr-un singur fisier, intr-un singur "listing" greu de urmarit si depanat in cazul programelor mari.

2. Limbaje orientate pe obiect (Object-Oriented, sau OO)

Aceasta categorie cuprinde limbaje moderne de programare (C++, C#, Java, Visual BASIC 7, Pascal-Delphi).

Caracteristica definitorie a acestora este aceea ca anumite portiuni de cod pot fi separate de restul programului, in sensul ca pot fi definite **clase** si **spatii de nume**, toate aceste fiind entitati de prelucrare a unor tipuri de informatie prin anumite metode. Executia este liniara, dar numai in interiorul unor blocuri de instructiuni proiectate intentionat pentru acest scop.

De exemplu, un program care trebuie sa se ocupe de clienti si facturi poate fi structurat astfel: o sectiune de program se zice ca este clasa de functionalitate care se ocupa de orice entitate de tip client, iar alta sectiune este clasa care se ocupa de orice entitate de tip factura. Astfel, din oricare din aceste doua clase, sau **modele**, pot fi create **obiecte** de acel tip, **fiecare independent de restul**.

Avantajele acestui tip de limbaj sunt evidente:

- **intretinere extrem de usoara a codului**, datorita structurarii informatiei in entitati specifice numai anumitor tipuri de operatiuni
- **expansiune** ulterioara avand ca material de lucru entitatile de baza (clasele care constituie modelul obiectelor)
- **interactiunea** intre entitatile de baza (clasele si spatiile de nume pot fi facute astfel incat sa comunice intre ele)
- **mostenirea**, care reprezinta capacitatea unei clase de a prelua si folosi cel putin o parte din functionalitatea alteia, devenind astfel o *subclasa*

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

- **incapsularea**, adica izolarea unor date si metode numai in portiuni de cod care pot fi reutilizate, in asa-numite cutii negre a caror functionalitate detaliata nu avem neaparata nevoie sa o cunoastem
- **accesibilitatea** sau **vizibilitatea**, numai unde este nevoie de ele, a unor date (constante, variabile) si metode
- asa-zisa **ascundere** sau **umbrire** a variabilelor, care reprezinta capacitatea programului de a lucra cu entitati numite identic, dar vizibile in regiuni diferite ale codului (clase, spatii de nume)
- programul poate fi privit ca o colectie logic structurata de **date si metode** care se constituie in **membri** ai claselor

Programele moderne sunt create in **medii de programare vizuala**. Acestea sunt la randul lor programe care permit utilizatorului sa scrie instructiunile care alcatuie un program, sa proiecteze interfata programului sau, sa aranjeze mai multe programe intr-o *solutie* sau *proiect*, sa creeze pachete (kituri) de instalare, sa testeze, compileze (pregatire pentru executie) si sa execute programul. De exemplu, un program pentru platforma Windows poate fi elaborat cu Microsoft Visual Studio, iar unul pentru platforma Linux/Unix cu mediul de programare NetBeans.

Mediile decente de programare vizuala in C# ofera programatorului facilitatile *AutoComplete* si *IntelliSense*. Aceasta inseamna ca functiile detinute de o entitate, ca si proprietatile acesteia (text, culoare, vizibilitate s.a.), intr-un cuvant **membrii** entitatii respective, pot fi derulate de catre mediul de programare, iar din lista acestora se poate alege un membru al respectivei entitati la tastarea simbolului **.** (punct) dupa numele acesteia. Asadar, tastarea caracterelor **bcom.** va duce la afisarea tuturor functiilor disponibile *clasei* din care face parte butonul *bcom* - clasa **Button**.

Desigur ca, pentru a beneficia de aceasta functionalitate, entitatea trebuie sa fie cunoscuta mediului de programare, adica mediului in care se creeaza programul. Aceasta se poate realiza fie creand butonul prin tragere din casuta de controale pe fereastra programului si denumindu-l *bcom*, fie prin definire/generare *programatica* sau *dinamica*, adica direct in codul programului, intr-o regiune care sa ii asigure vizibilitatea pentru comanda pe care vrem sa i-o dam.

Am expus pana aici notiuni fundamentale pentru oricine e in contact cu orice program - esenta circulatiei informatiei in platforma Microsoft Windows, notiunea de identificator, constanta, control si biblioteca; de asemenea, tipurile de limbaje de programare si functionalitatea *AutoComplete* si *IntelliSense*.

2. FUNCTII SAU METODE.
CLASE. STRUCTURI.
SPATII DE NUME. PROPRIETATI SI ACCESORI

***Incepand din acest capitol vor fi folosite un numar restrans de cuvinte-cheie ale limbajului C#. Prezentarea tuturor cuvintelor-cheie este facuta in Anexa 6*.**

Mentiune speciala merita metodele *MessageBox* si *ToString* respectiv afisand o caseta de mesaj (un dialog) si transforma un obiect intr-unul de tip text. Prima dintre acestea poate fi folosita cu generozitate pentru urmarirea fluxului si datelor manipulate de un program.

2.1. FUNCTII SAU METODE. DELEGARI

Functiile (denumire caracteristica C++; in C# se prefera denumirea de ***metode***) sunt entitati de prelucrare a informatiei. De regula, o functie executa o comanda, de exemplu *2+3*, si intoarce utilizatorului rezultatul (*5*).

Functiile pot primi si folosi *parametri* sau *argumente*. Acestia sunt datele care teoretic sunt necesare prelucrarii informatiei in cadrul functiei, desi utilizarea lor nu este obligatorie (ceea ce denota o proiectare defectuoasa a functiei). Astfel, functia care calculeaza valoarea 5 poate avea (ii pot fi trimisi) un numar oarecare de parametri, rezultatul combinarii lor matematice in interiorul functiei fiind insa 5, in fiecare caz. Poate primi, de exemplu, doi parametri (2 si 3, 1 si 4, 4 si 1, 3 si 2, 5 si 0, 1/5 si 25 etc.), trei parametri (1 ~ 2 ~ 2, 3 ~ 1 ~ 1, 0 ~ 4 ~ 1, samd) sau chiar unul singur (5).

Functiile pot fi apelate si fara parametri; importanta este prelucrarea datelor de catre ele pentru a obtine un rezultat. 5 poate fi obtinut si din apelarea unei functii fara parametri, de exemplu una care intoarce simplu numarul 5.

De exemplu, functia intrinseca sistemului de operare *SetWindowText* va necesita intotdeauna parametri, anume identificatorul controlului caruia urmeaza a i se modifica textul, precum si textul in sine.

Declaratia acestei functii este (in C++ si accesibila in Visual Studio 6):

BOOL SetWindowText(HWND *hWnd*, LPCTSTR *lpString*)

Descrierea sa ar fi urmatoarea:

"Se defineste functia SetWindowText, de tip **BOOL**ean (succesul apelarii este adevarat/fals), care primeste parametrii hWnd de tip **HWND** (**H**andle to **W**ind**o**w) si lpString de tip **LPCTSTR** (**L**ong **P**ointer to **C**ons**t**ant **Str**ing – pointer sau adresa de memorie corespunzand unui sir constant de caractere)."

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Aceasta functie intoarce o valoare nenula (*true-adevarat*) daca reuseste (adica daca sunt indeplinite conditiile: hWnd exista; lpString este declarat in cod; exista resurse pentru indeplinirea comenzii) si nula (*false-fals*) daca esueaza.

Se spune ca functia *SetWindowText* este de tip boolean fiindca rezultatul apelarii este de tip adevarat/fals sau 1/0, adica poate avea numai una din doua valori de tip logic. Daca, de exemplu, apelarea sa ar fi trebuit sa intoarca explicit un numar intreg, ar fi trebuit declarata ca (in loc de BOOLean se declara a fi de tip INTeger):

INT SetWindowText(HWND *hWnd*, LPCTSTR *lpString*)

O functie care nu intoarce nici un rezultat, dar care executa totusi instructiuni, este declarata drept **void**, atat in C++, cat si in C#:

void SetWindowText(HWND *hWnd*, LPCTSTR *lpString*)

Functiile care intorc (furnizeaza) rezultat trebuie sa contina instructiunea:

return *Rezultat*;

Rezultat este o variabila de acelasi tip cu cel declarat al functiei.

In functiile de tip **void** nu este permisa existenta instructiunii **return**.

Rezultatul functiilor le defineste tipul.

Relativ la declaratiile de functii, in C# se defineste notiunea de **supraincarcare** (**overloading**), care este ilustrata in urmatorul exemplu.

Fie functia

```
private int AdunareSimpla(int iTermen1, int iTermen2)
{
        int iRezultat;
        iRezultat= iTermen1+ iTermen2;
        return iRezultat;
}
```

Aceasta functie poate fi *supraincarcata* prin redefinirea parametrilor:

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
private int AdunareSimpla(int iTermen1, int iTermen2, string sParam3)
{
        int iRezultat;
        iRezultat= iTermen1* iTermen2;
        return iRezultat;
}
```

Prima varianta a functiei primeste doi parametri, ii aduna si intoarce un numar intreg.

A doua varianta primeste trei parametri, ii inmulteste, dar intoarce tot un numar intreg.

Dupa cum se observa, desi este vorba de doua functii distincte, numele lor este acelasi, iar rezultatul este de acelasi tip; ceea ce ne spune ca o functie este supraincarcata este **tipul parametrilor**. Ansamblul parametrilor se defineste ca **semnatura** functiei.

Nu conteaza ce operatiuni au loc in cadrul metodelor, atat timp cat se respecta cerinta ca valoarea rezultata (daca functia nu este declarata de tip **void**) sa fie de tipul declarat al functiei (**int** in acest exemplu).

Supraincarcarea metodelor se face prin modificarea tipului si/sau numarului parametrilor, numele metodelor fiind acelasi.
Semnatura unei metode (signature) reprezinta numele metodei, urmat de lista parametrilor.
Metodele pot intoarce orice tip de date, inclusiv tipuri clasa.

Functiile sau metodele pot fi alaturate intr-o structura de program, de cod unitar, pentru a alcatui o biblioteca de functii. De asemenea, ca si in bibliotecile de sistem, in bibliotecile personalizate pot fi introduse constante. Toate acestea pot fi utilizate ulterior, prin referire la biblioteca de catre programul care are nevoie de ele.

De exemplu, o biblioteca personalizata creata in Visual C# sau Visual Basic poate redefini valoarea mesajului BM_CLICK din 245 in 255 si sa o foloseasca nu ca mesaj spre un buton, ci pentru a o trimite ca parametru unei functii. De regula insa, in Visual C++ mesajele de sistem sunt folosite ca atare si este indicat ca aceasta sa fie practica in orice program.

Metodele care sunt membri ai unei clase se pot apela reciproc, indiferent de gradul lor de accesibilitate (**private** sau **public**).

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

De exemplu:

```
class ClasaMea //aici se declara inceputul blocului unei clase
{ //acolada de deschidere a acestei entitati
        int i= 4;

        private int IncrementareCuUnu(int UnIntreg)
        {
                UnIntreg++;
                return UnIntreg;
        }
        public void MetodaOcolitoarePentruIncrementare()
        {
                IncrementareCuUnu(i);
                //i= 4 se trimite metodei IncrementareCuUnu
        }
        private void AfiseazaValoareaUnuiIntreg ()
        {
                MetodaOcolitoarePentruIncrementare();
                MessageBox.Show(i.ToString());
                //va fi afisat mesajul "5";
        }
}
```

Acest cod ilustreaza apelarea metodelor. Pentru aceasta, am urmat pasii:

1. Am declarat metoda care efectueaza operatiunile propriu-zise (in acest caz, adunare);
2. Am declarat metoda care va apela metoda care lucreaza efectiv;
3. Am declarat o metoda suplimentara prin care procesam rezultatul pasilor 1 si 2 (afisarea rezultatului cu metoda *MessageBox*).

Variabila ***i***, de tip valoare intreaga, este cea care este supusa modificarii. Astfel se ilustreaza o regula de baza in C#, anume ca variabile declarate in interiorul unui bloc sunt vizibile in blocurile continute in acesta. Deci, intrucat variabila ***i*** este declarata direct in corpul clasei, va fi vizibila in corpurile metodelor care sunt continute la randul lor in clasa.

Metodele se pot autoapela (pot fi **recursive**).

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Recursivitatea metodelor este reprezentata tipic de factorial:

```
private int factorial(int i)
{
	if (i>1) // daca i este nenul si pozitiv, factorialul poate fi calculat
	{
		return i* (factorial(i-1)); // 2!= 2*(2-1)!
	}
	else // daca i este 0 sau negativ, intoarcem valoarea 1
	{
		return 1;
	}
}
```

Procedeul prin care o metoda se autoapeleaza se numeste recursivitate.

Metodele pot primi un numar variabil de argumente (necunoscut la momentul declararii metodei, dar cunoscut la apelarea acesteia); aceasta, insa, se poate face numai daca in afara acestora se specifica explicit cel putin un parametru. Cuvantul-cheie folosit pentru o lista de parametri (al caror numar este necunoscut initial si care devine cunoscut la apelarea metodei) este **params**.

De exemplu, fie metoda:

```
int Metoda(string s, params int[] args)
{

}
```

Se observa cele doua paranteze drepte de langa tipul **int**, care semnifica faptul ca metoda va primi un tablou (v. 6.1.) de argumente, toate de tip intreg (**int**). Metoda va putea fi apelata in mai multe moduri, de exemplu:

```
int valoare= Metoda("un text", 72);
int valoare= Metoda("un text", 72, 189);
int valoare= Metoda("un text");
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Tipul argumentelor specificat de **params** trebuie sa fie respectat atunci cand acestia se trimit metodei. Apelul metodei in felul urmator este incorect si imposibil, intrucat "alt text" trimis ca valoare a argumentului **params** nu este o valoare de tip **int**, cum acesta este definit in semnatura metodei:

```
int valoare= Metoda("un text", "alt text"); //INCORECT
```

Se mai spune ca modificatorul **params** este folosit pentru a crea un parametru care acopera un numar variabil de parametri efectivi.

O metoda nu poate admite mai mult de un parametru de tip **params**.

params poate fi situat numai pe ultimul loc in lista de parametri ai unei metode si poate fi si singurul parametru al unei metode.

Metodele pot fi declarate **virtual**. O astfel de metoda este declarata intr-o clasa numita **de baza** si redefinita apoi in cel putin una dintre clasele derivate.

Fiecare dintre clasele derivate poate, asadar, dispune de (sau **implementa**) o versiune proprie a unei metode virtuale.

Metodele virtuale **au corp de instructiuni**.

De asemenea, metodele pot fi declarate **abstract**. Aceste metode vor fi de asemenea implementate in clasele derivate din clasa de baza, in care sunt definite, dar **nu au corp de instructiuni**.

Exemple de utilizare a acestor doi specificatori de acces (virtual, abstract) sunt oferite in Anexa 9.

De exemplu, fie clasa:

```
class Clasa1
{
        public virtual string s(int i)
        {
                return i.ToString();
        }
}
```

Derivam o clasa din Clasa1 (cream o clasa care mosteneste Clasa1):

```
class Clasa2 :  Clasa1
// : semnifica faptul ca membrii declarati public in clasei Clasa1
//vor fi vizibili in Clasa2
//Se spune ca Clasa2 mosteneste Clasa1
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
//Informatii despre mostenire sunt oferite in continuare (2.2. - Clase)
{
        override public string s(int i) //versiunea lui s pentru Clasa2
        {       //versiunea proprie din aceasta clasa, care ignora
                //versiunea din clasa de baza
                i= i+1;
                MessageBox.Show(i.ToString());
                return i.ToString();
                //finalizarea este aceeasi ca in clasa de baza (return),
                //indiferent de instructiunile suplimentare
        }
}
```

O metoda virtuala se declara ca* virtual *intr-o clasa de baza, fiind apoi extinsa in clasele derivate.
Extinderea metodelor virtuale se realizeaza utilizand modificatorul* override*.
Metodele abstracte nu au corp si nu sunt implementata in clasa de baza. Declaratia lor consta din tipul rezultatului, nume si lista de parametri.
Membrii virtuali si abstracti nu pot fi privati.
Metodele abstracte se pot declara numai in clase abstracte.

DELEGARI

Delegarile (**delegate**) sunt obiecte care refera metode. O delegare permite apelarea unei metode facute cunoscute abia la momentul apelarii.

Metodele pot fi, astfel, apelate prin delegari. Aceeasi delegare poate referi diverse metode, esential fiind tipul ambelor structuri.

Forma generica a declaratiei unei delegari este:

delegate *tip nume*(parametri);

Delegarile sunt *multicast*, adica ofera posibilitatea de a se crea un lant de metode care vor fi automat apelate la invocarea unei delegari. Adaugarea metodelor se face cu operatorul +=, iar inlaturarea lor cu operatorul -=.

Un exemplu de utilizare este prezentat in continuare. Fie doua metode, str1 si str2, care intorc si primesc ca parametru o variabila de tip **string**.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
delegate string sdel(string s);

private string str1(string s)
{
        return s+ " delegat 1";
}

private string str2(string s)
{
        return s+ " delegat 2";
}

string textul= "Un text";
sdel sfinal;
sdel deleg= new sdel(str1); //un delegat spre metoda str1
sdel deleg2= new sdel(str2); //inca un delegat, spre metoda str2
sfinal= deleg;
MessageBox.Show(sfinal(textul));
//aplicam un multicasting:
sfinal+= deleg2;
MessageBox.Show(sfinal(textul));
//daca scriam:
//sfinal= deleg+ deleg2
//rezultatul ar fi fost afisarea stringului modificat in str2
```

Delegarile nu au o justificare evidenta, fiind mai simplu sa apelam direct metodele. Totusi, ele ofera posibilitatea de a executa o metoda fara a sti care este acea metoda inca de la momentul compilarii, ci aceasta este cunoscuta numai in momentul apelarii. Multicasting-ul este util in filtrarea operatiunilor efectuate pe un anumit tip de date.

2.2. CLASE

Am vazut ca entitati informationale sau de prelucrare a informatiei, precum functiile si constantele, pot fi organizate in biblioteci de functii ("dll"-uri).

Bibliotecile sistemului de operare Microsoft Windows contin mai mult de 20000 de declaratii de functii si constante. A invata denumirea si functionalitatea chiar si numai a 100 dintre acestea, impreuna cu parametrii pe care ii necesita fiecare functie, este prea mult chiar si pentru un programator veteran care prefera sa scrie litera cu litera functiile (de aceea s-a si inventat functionalitatea *AutoComplete*).

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

De regula insa, functiile sunt organizate dupa categoria din care fac parte. Astfel, cele care se ocupa de interfata grafica sunt stocate in biblioteca gdi32.dll, cele care asigura interactiunea cu utilizatorul in user32.dll etc.

Mai departe, in cadrul bibliotecilor insesi, functiile si constantele pot fi organizate – de exemplu, in functii care traseaza linii curbe, functii care se ocupa de prelucrarea variabilelor de tip **string** (text) si functii care asigura accesul la Internet.

Aceasta a doua metoda de orgzanizare este cea mai utila programatorului, fiindca in primul rand ii de a faciliteaza accesul structurat la functii cu destinatii specifice si ii ofera posibilitatea de a restrange cautarea cu IntelliSense (daca este nevoie).

Astfel, functiile pot fi (si trebuie sa fie, intr-un program decent cu functionalitate diversa) organizate in **clase**.

O clasa este o entitate informationala in care sunt definite metode, constante si proprietati (**date si metode**, sau **membri**), apelabile depinzand de gradul lor de accesibilitate (private, publice, interne etc.) dintr-o alta portiune a programului sau, prin referire la biblioteca ce o contine, din alt program.

Clasele sunt *modelul* dupa care sunt create **obiectele**.
Un obiect este o instanta a unei clase.

Obiectele expun functionalitatea (membrii) claselor; daca nu cream nici un obiect care sa derive dintr-o clasa, putem construi oricate clase dorim, dar functionalitatea lor ne va fi inaccesibila.

Pentru a intelege, dam urmatorul exemplu: exista specia canidelor, din care face parte catelul Rex. Specia este o notiune abstracta privita ca o clasa, dar Rex este un reprezentant al sau, ce poseda caracteristicile sale de baza (mustati si urechi de o anumita conformatie, ochi verzi etc). Nu pot exista mai multe specii de canide, ci mai multi reprezentanti ai acestei specii.

Atunci cand se creeaza un obiect se spune ca se creeaza o instanta a unei clase.

Un **constructor** initializeaza un **obiect** atunci cand acesta este creat, rezervand memorie pentru toate datele definite de clasa-model. Constructorii au acelasi nume cu clasa din care fac parte, fiind similari din punct de vedere sintactic cu metodele obisnuite; nu au insa un tip explicit.

Constructorul se utilizeaza la initializarea obiectului creat. Constructorii nu au un tip explicit, dar au acelasi nume cu clasa in care sunt definiti.

Forma generala a unui constructor este (observati parantezele de dupa numele clasei; la declararea unei clase parantezele nu se folosesc):

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
nume_clasa()
{
        //codul constructorului
}
```

Structura generica a unei clase este:

```
class ClasaMea
{
        public ClasaMea() //acesta este constructorul
        {
                //aici se afla codul constructorului, executat imediat
                //la crearea unei instante a clasei (a unui obiect)
        }
        //urmeaza declaratii de constante si variabile, de exemplu:
        int x= 2;
        ulong ul;
        string s= "abc_xyz-123+890";
        //declaratii de metode, de exemplu:
        public void ModificaTextulPeButonul (Button b,
                                                string sTextNou)
        {
                b.Text= sTextNou;
        }




        proprietati accesibile ale clasei, de exemplu:
        private byte b;     //variabila utilizata pentru proprietate
        public byte bx      //proprietate: bx
        {
                get
                {
                        return b;
                }
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
            set
            {
                b=value;
            }
        }
    }
```

In acest exemplu sunt declarate variabile de tip **private** si **public**. Acesti *modificatori de acces*, impreuna cu **static**, **internal** si **protected**, definesc accesibilitatea membrilor clasei.

Membrii clasei generice prezentate sunt: numarul intreg **x**, textul **s**, metoda **ModificaTextulPeButonul** si intregii pe 8 biti **b** si **bx**.

Cuvantul-cheie **value** este folosit ca atare; reprezinta stocarea valorii atribuite proprietatii in variabila privata **b**, de unde va fi iarasi preluata de catre metoda **get**.

Din aceasta clasa putem crea o instanta a sa (un obiect) astfel:

```
ClasaMea cm= new ClasaMea();
```

Operatorul new aloca memorie pentru un obiect si il initializeaza utilizand constructorul acestuia.

Observati prezenta parantezelor; acestea definesc constructorul implicit al clasei, fara parametri; daca defineam explicit un constructor (cu parametri sau fara), in paranteze trebuia sa specificam valorile parametrilor de initializare (daca existau). Parantezele sunt, oricum, obligatorii la crearea unui obiect dintr-o clasa.

Dupa ce am creat acest obiect, unii dintre membrii clasei vor fi vizibili - anume membrii declarati **public**. Restul membrilor, chiar daca nu sunt declarati **private**, cum este **x** (int x= 2), sunt accesibili numai in cadrul clasei respective (adica sunt de uz intern pentru procedurile din clasa, nu din obiectele derivate).

Instantierea obiectului **cm** nu se putea efectua prin conversie explicita din tipul generic **object** la tipul **ClasaMea**:

```
object cm= new object();
cm= (ClasaMea)cm;
```

^ INCORECT, IMPOSIBIL ^

!!! ***Tipul object este tipul de baza al tuturor variabilelor*** **!!!**

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Astfel, o variabila de tipul **int**, una de tipul **string**, una de tipul **byte** s.a.m.d. sunt toate **object** si convertibile implicit la **object.** Pe de alta parte, clasele nu sunt tipuri de valori, ci structuri de date, asadar conversiile din **object** in clase **nu sunt posibile**.

In interiorul unei clase se pot defini alte clase (clase imbricate) - procedeu aplicat de obicei in cazurile in care structurarea membrilor pe un singur nivel nu este suficient de explicita sau de organizatoare a informatiei.

Nu se pot imbrica, insa, clase denumite identic - **membrii nu pot avea numele tipului in care sunt inclusi**.

De exemplu, daca avem o clasa (Clasa1) cu doi membri care implementeaza operatii cu variabile numerice, unii dedicati tipului **int** (*Clasa1.ImpartireInt*) si altii tipului **long** (*Clasa1.AdunareLong*), este mai util ca in cadrul Clasa1 sa definim alte doua clase (*ClasaInt* si *ClasaLong*) care sa contina membrii orientati spre fiecare tip. Aceasta operatiune ne va permite sa alegem din clasa *Clasa1.ClasaInt* membri orientati spre tipul **int**, iar din *Clasa1.ClasaLong* membrii dedicati tipului **long**. Acest procedeu este ilustrat in continuare.

```
public class c //clasa principala, in care vor fi definite 2 clase specializate
{
	public class c_strings //clasa care se ocupa de string-uri
	{
		public string s1(string s)
		{
			return s+ "abc");
		}
		public string s2(string s)
		{
			return s+ "def");
		}
	} //sfarsitul clasei care se ocupa de string-uri

	public class c_nums //clasa care se ocupa de valori numerice
	{
		public string n1(int i)
		{
			return i+ 100;
		}
		public string n2(double d)
		{
			return d/2;
		}
	} //sfarsitul clasei care se ocupa de valori numerice
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

} //sfarsitul clasei principale (c)

In cod, operatorul punct (**.**) utilizat dupa numele unui obiect astfel creat ofera posibilitatea de a referi clasele aflate "sub nivelul 1", anume prin calea completa spre ele, care in cazul de fata este ***c.c_strings*** sau ***c.c_nums***.

Clasele pot fi **mostenite**.

Aceasta inseamna ca dintr-o clasa putem deriva alta clasa care sa ne asigure, prin obiectele derivate din ea, accesul la membrii **publici** ai clasei de baza. Procesul in sine se numeste, evident, **mostenire** (**inheritance**).

O clasa a carei mostenire dorim sa o interzicem trebuie declarata **sealed**.

Fie clasa Clasa1 care contine 3 membri:

- o variabila de tip **int**
- o variabila de tip **string**
- o metoda denumita StergeTextul

Se creeaza clasa Clasa2, care o dorim a mosteni Clasa1; linia de cod care efectueaza aceasta operatiune este:

public Clasa2 : Clasa1

Mostenirea este specificata cu caracterul "doua puncte" (:)

Atributul **public** poate lipsi, iar in acest caz Clasa2 va fi privata, deci vizibila numai in spatiul de nume (blocul de informatii/membri) in care este declarata.

Se presupune, aici, ca ambele clase fac parte din acelasi spatiu de nume.

Clasa2 poate fi construita, la randu-i, dupa cum dorim, incorporand in ea diversi membri. Indiferent insa ce structura va avea, pe langa membrii sai, la o instantiere a sa intr-un obiect, acesta va expune si membrii clasei Clasa1 (**de baza** sau **parinte** al clasei Clasa2).

Mostenirea este o caracteristica de baza a programarii orientate/bazate pe obiecte (sau obiectuale).

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Clasele pot fi declarate **abstract**:

```
abstract Clasa1
{
        membri
}
```

O clasa abstracta este o clasa asa-zisa de baza, adica una care numai stabileste un sablon general, comun pentru toate clasele derivate, fiecare dintre acestea completand detaliile caracteristice.

O astfel de clasa fixeaza natura metodelor pe care clasele derivate trebuie sa le implementeze, fara insa a oferi o implementare a metodelor.

Metodele dintr-o clasa abstracta care vor fi implementate trebuie sa fie declarate abstracte, la randul lor.

In clasele derivate, cuvantul-cheie **override** va fi asociat cu metodele care asigura functionalitatea efectiva.

Daca o clasa derivata nu implementeaza toate metodele abstracte din clasa de baza, atunci si aceasta trebuie declarata abstracta.

Nu se pot crea instante ale claselor abstracte utlizand cuvantul-cheie **new**.

Fie clasa:

```
abstract Clasa1()
{
        public abstract int Arie(int latime, int lungime);
        //metoda declarata la randul sau abstracta, fara corp;
        //va fi implementata in clasele derivate
}

class Clasa2 : Clasa1
{
        override public int Arie(int latime, int lungime)
        {
                return latime * lungime;
                //implementarea metodei abstracte Aria din Clasa1
        }
}
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

O clasa derivata este o versiune specializata a unei clase de baza, pe care o mosteneste.
O clasa derivata contine membrii publici ai clasei de baza.
O clasa derivata nu poate accesa membrii privati ai clasei de baza.
base ***se refera intotdeauna la constructorul clasei de baza situate imediat peste clasa apelanta.***
O clasa derivata poate fi clasa de baza a unei alte clase.
Constructorii sunt apelati in ordinea stabilita de derivare.
O clasa care contine cel putin o metoda abstracta trebuie de asemenea sa fie declarata ca abstracta.
Nu se pot crea instante ale claselor abstracte.
O clasa declarata ca sealed nu poate fi mostenita.

Utilizarea cuvintelor-cheie abstract si override sunt expuse in Anexa 8.

2.3. STRUCTURI

Structurile (*struct*) sunt entitati asemanatoare claselor, ce indeplinesc aceeasi functionalitate cu o clasa avand aceiasi membri. Claselor cu putini membri le sunt preferate structurile.

Un exemplu ar fi o clasa **Pahar**, care are numai doua proprietati si nu dispune de nici o metoda: **bool** *ContineLichid* si **double** *ProcentDeUmplere*. Ca sa avem acces la acesti (numai doi) membri, nu are rost (ca programatori comozi, in primul rand) sa cream o clasa in toata regula. Astfel ca folosim o structura.

Exemplul prezentat de obicei in manuale este **Point**, care reprezinta o locatie pe o suprafata desenabila, determinata de doua coordonate:

```
public struct Point
{
	public int x, y;
	public Point(int p1, int p2)
	//crearea unui obiect Point va fi apelata cu acesti 2 parametri,
	//de exemplu Point p= new Point(100, 250);
	{
		x= p1;
		y= p2;
	}
}
```

Atunci cand vom crea un obiect de acest tip, vom face aceasta **initializand neaparat parametrii** sai, de exemplu:

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Point p= new **Point**(100, 250);

Daca nu initializam parametrii, instantierea unui obiect derivat dintr-o structura este imposibila. **Structurile nu pot defini constructori**.

2.4. INTERFETE

Se spune ca o interfata **defineste un contract**.
Altfel spus, interfetele sunt **modele pentru membrii unei clase**.
Interfetele pot fi **implementate** in clase, respectivele clase trebuind sa respecte specificatiile din interfetele asociate.
Interfetele pot contine metode, proprietati si evenimente. Interfetele insesi nu furnizeaza functionalitate pentru membrii pe care ii defineste; ele furnizeaza numai membrii claselor care le vor implementa.
Pentru intelegerea notiunii, fie de exemplu interfata:

```
interface InterfataSimpla
{
        double MetodaCuContinutAleator(int h);
}
```

Orice clasa care o va implementa **va contine obligatoriu** corpul metodei MetodaCuContinutAleator; desigur, metoda va trebui sa intoarca un rezultat de tip **double**, dupa ce primeste un parametru de tip **int**:

```
class c: InterfataSimpla //clasa c implementeaza (nu mosteneste) interfata
{
        public double MetodaCuContinutAleator(int x) //implementarea metodei
        {       //nu e obligatoriu ca parametrii metodei implementate ("x") sa
                //aiba acelasi nume cu cei din metoda de baza ("h")
                return x/2;
        }
}
```

La fel cu clasele, interfetele se pot mosteni intre ele.

2.5. SPATII DE NUME

La randul lor, clasele pot fi organizate in **spatii de nume** (***namespace***). Acestea sunt regiuni declarative care asigura consistenta apelarii membrului **f** din clasa **c** existenta in spatiul de nume **sn**, calea spre acest membru fiind **sn.c.f**

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Ratiunea de a exista a spatiilor de nume este, in mare, aceasta: diversi programatori pot denumi identic atat bibliotecile de functii (stocate de obicei in fisiere .dll) pe care le pun la dispozitia utilizatorilor, cat si clasele si functiile din interiorul acestora. Doua programe provenind de la producatori diferiti pot coexista intr-un sistem, apeland functii din biblioteci specifice fiecaruia, dar avand acelasi nume (si functiile, si fisierele-biblioteca). Spatiile de nume ofera protectie impotriva eventualelor ambiguitati la apelarea acestor biblioteci, prin denumirea regiunilor declarative astfel incat cel putin programatorul sa cunoasca originea functiilor apelate. Astfel, sunt asigurate cel putin doua niveluri de protectie dependenta exclusiv de numele regiunilor declarative: spatiu de nume -> clasa.

Prin declararea unui spatiu de nume, cu **namespace**, se defineste un domeniu de valabilitate a membrilor continuti.

Structura generica a unui spatiu de nume este:

```
namespace SpatiulMeuDeNume
{
        [...]
        public class Clasa1
        {
                membrii clasei publice Clasa1
        }
        private class Clasa 2
        {
                membrii clasei private Clasa2
        }
}
```

Continuand prin a declara in spatiul de nume-parinte inca unul, denumit **AltSpatiuDeNume**, iar in interiorul sau doua clase denumite tot **Clasa1** si **Clasa2**, referirea ulterioara la fiecare clasa va trebui facuta specificand spatiul de nume-parinte, astfel:

SpatiulMeuDeNume.**AltSpatiuDeNume**.Clasa1.MembruAlClasei1

Clasa Clasa2 este declarata ca fiind **private** in spatiul de nume **SpatiulMeuDeNume**, asadar, ca orice membru privat, nu este vizibila in exteriorul blocului ce defineste aceasta structura.

Asadar, spatiile de nume, ca si clasele, pot fi incluse unul in altul si membrii lor vor putea fi accesate specificand calea completa spre ei.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

In cadrul unui spatiu de nume se pot declara clase, structuri, delegari, enumerari, interfete si alte spatii de nume.

Cuvantul-cheie **using** asigura accesul la alte spatii de nume. Directivele **using** trebuie incluse la inceputul fiecarui program, inaintea altor declaratii, de exemplu

using System;

sau

using System.Data.SqlClient;.

Un spatiu de nume poate fi impartit in mai multe fisiere - parti ale unui proiect; el va trebui sa fie declarat cu acelasi nume in toate fisierele, iar membrii specificati in fiecare fisier vor fi considerati continuti in acelasi spatiu de nume.

Asadar, putem privi atat numele de spatii, cat si clasele, ca forme structurate ale datelor prelucrabile in cadrul unui program, ce permit accesul facil si eliminarea ambiguitatilor la apelarea membrilor.

Spatiile de nume ale platformei Microsoft .NET sunt prezentate in Anexa 4.

2.6. PROPRIETATI SI ACCESORI

Un obiect poate avea **proprietati** sau **campuri**. Deja am oferit un exemplu in sectiunea 2.4., pe care il reluam aici. In cadrul clasei generice prezentate existau urmatoarele declaratii:

```
private byte b;
public byte bx
{
        get
        {
                return b;
        }
        set
        {
                b=value;
        }
}
```

Variabila *bx* de tip **byte** reprezinta o **proprietate** sau un **camp** al clasei din care face parte si pentru ea au fost definite posibilitatile prin care o putem accesa - anume **get** si **set**. Acestea se numesc **accesori** si reprezinta, respectiv, metoda prin care se obtine valoarea variabilei si metoda prin care proprietatii i se atribuie o valoare.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

In cadrul fiecarui accesor putem defini mai multe operatiuni, insa fiecare trebuie sa indeplineasca rolurile definitorii – de atribuire si intoarcere a valorii.

Accesorii definesc tipul de acces la proprietatea careia de care apartin. In exemplul de fata, proprietatea este variabila **bx**, de tip **byte**. Instructiunile din **get** si **set** pot efectua operatiuni diverse, insa **set** trebuie sa atribuie lui **b** o valoare ce se incadreaza in domeniul disponibil tipului **byte; get** intoarce, in orice caz, valoarea lui **b**, care a fost deja declarata de tipul **byte**.

Drumul acesta ocolitor prin care se poate accesa o proprietate este un alt exemplu de structurare a informatiei in cadrul unei clase. In exemplul acesta, variabila **b** este declarata privata, adica nu poate fi accesata din exteriorul clasei si este exclusiv pentru uz intern, iar valoarea sa este cea care este intoarsa atunci cand este interogata proprietatea/variabila publica **bx**.

Pentru a putea accesa bidirectional o proprietate, puteam la fel de bine sa declaram o variabila publica **bx**, a carei valoare o puteam stabili si accesa direct din alta portiune a programului.

Accesul bidirectional nu este, insa, intotdeauna cel dorit de programator; putem numai expune valoarea unei proprietati, nu sa facem posibila si modificarea sa, ori invers. In aceste cazuri, vom declara numai accesorul **get** (astfel proprietatea devenind *read-only*), respectiv **set**.

Se observa ca accesorul **set** contine o atribuire gen ***variabila* = value**.

Aceasta inseamna, pur si simplu, ca atunci cand din cod modificam valoarea lui **bx**, de exemplu **bx= 2**, 2 devine **value**; pe scurt, accesorul **set** atribuie lui **b** orice valoare valida i se trimite.

Blocurile **get** si **set** pot sa fie vide (sa nu contina instructiuni).

3. STRUCTURA GENERICA A UNUI PROGRAM SCRIS IN LIMBAJUL C#

Cu limbajul C# poate fi creata o gama larga de programe (proiecte): *de consola* (destinate pentru rulare in mediul MS-DOS - Microsoft Disk Operating System); cu interfata grafica pentru platforma Windows (*Windows Forms*); fara interfata grafica pentru platforma Windows (*servicii*); *aplicatii web*, destinate accesarii prin Internet si de pe minicomputere (Pocket-PC); *controale* cu functionalitati complexe pentru incorporare in programe; *biblioteci* de clase.

Un program C# consta dintr-unul sau mai multe fisiere-sursa, denumite **unitati de compilare**, organizate intr-un de **proiect** (sau **solutie**). Aceste fisiere contin codul ce urmeaza a fi interpretat si compilat de catre platforma .NET intr-un program executabil.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Programatorul construieste programe fie prin tastare directa a instructiunilor, fie folosind automatizari de tip *wizard* al caror rezultat este generarea automata de cod de catre mediul de programare.

Textul programelor C#, si in general al celor create cu mediul de programare Visual Studio, este text neformatat (**bold**, *italic* sau cu diacritice); compilatorul interpreteaza codul ca text simplu. Totusi, mediul de dezvoltare poate fi modificat astfel incat sa fie mai placut (culoarea cuvintelor-cheie si a variabilelor, caracteristicile setului de caractere etc.).

Orice program creat cu C# contine obligatoriu sectiunile:

1. Sectiunea **using**, plasata in fata oricarei alte instructiuni, specifica ***modulele*** (**referintele** sau **ansamblurile de functionalitate** sau **spatiile de nume .NET sau de alta natura**) care vor furniza functionalitate programului. Totusi, fiecare instructiune **using** e relevanta numai clasei in care este folosita, nu intregului program, ceea ce este o consecinta directa a separarii entitatilor functionale intr-un program .NET.

   De exemplu, linia de comanda

   using System.Windows.Forms;

   anunta compilatorul de cod ca **in clasa curenta** vor fi efectuate referiri la ansamblul care contine metode si constante necesare lucrului cu ferestre si controale vizibile prin care utilizatorul va interactiona cu programul; linia

   using System.Graphics;

   anunta ca **tot in clasa curenta** vor fi necesare functii destinate interactiunii cu interfata grafica a sistemului.
   Este indicat sa fie referite numai ansamblurile folosite efectiv. Implicit, la crearea unui nou program, mediul de programare adauga linia:
   using System.Data;
   intrucat acest spatiu de nume contine clase utile accesarii bazelor de date, dar nu orice program are nevoie de aceasta, respectiva instructiune poate fi stearsa.

2. Sectiuni sau blocuri **namespace**, care contin toate declaratiile de clase ale programului.

3. Sectiuni **class**, care prin membrii lor vor asigura functionalitatea efectiva a programului.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Astfel, structura generica a unui program C# este:

```
using Ansamblu1;
using Ansamblu2;
[…]
namespace SpatiuDeclarativ
{
    class Clasa
    {
            declaratiile membrilor clasei, constante, variabile, metode:
            private int x;
            public string s1;

            private void Metoda1(int iArg1, string sArg2)
            {
                    operatii efectuate de functie
            }
    }
}
```

De obicei, codul incepe cu instructiunea **using System**. Spatiul de nume **System** contine toti membrii platformei .NET. Aceasta instructiune nu este obligatorie, dar omiterea sa inseamna ca vor trebui "calificate" numele tuturor membrilor spatiului acestuia de nume, care sunt folositi (de exemplu, **System.IO.**FileInfo, in loc de **IO.**FileInfo).

Executia unui program incepe cu metoda **Main**(). Aceasta poate fi declarata publica, poate folosi argumente si contine instructiunile de initializare a executiei: de exemplu, creeaza interfata de tip *Windows Form* sau creeaza un prompt pentru o aplicatie de consola. In aceasta metoda se initiaza, asadar, interactiunea cu utilizatorul.

Un exemplu detaliat al utilizarii metodei Main se afla in Anexa 8.

Fiecare regiune declarativa, incepand cu namespace si terminand cu corpul metodelor si al eventualilor accesori, trebuie sa inceapa cu acolada deschisa si sa se incheie cu acolada inchisa.

Instructiunile sunt interpretate si executate in ordinea dispunerii in cod - liniar sau prin derivare prin salturi spre alte secvente de instructiuni.

Blocurile de cod, adica instructiuni incluse in acolade, nu provoaca intarzieri in executie. Folosirea lor este chiar indicata pentru marirea lizibilitatii codului si a sigurantei programarii.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Pot fi definite **regiuni de cod** care pot fi restranse si extinse; scriind **#region** pe o linie semnalam mediului de programare (nu compilatorului de cod, care nu va interpreta aceasta linie drept o instructiune executabila) ca urmeaza o regiune in care am dorit sa organizam informatia intr-un mod anume, de regula pentru mai usoara inspectare a codului si pentru grade marite de lizibilitate si organizare. Corpul unei regiuni se incheie cu declaratia **#endregion**. In mediul de programare, in stanga liniei care contine **#region** va aparea un patrat marcat cu simbolul**+**; efectuand click-stanga pe acest simbol, regiunea de cod cuprinsa pana la **#endregion** va fi restransa, organizand astfel o portiune de cod.

INTRODUCEREA DE COMENTARII

Dupa orice instructiune (pe orice linie), putem adauga **comentarii**. Comentariile pot exista si ca linii independente de program; ele nu afecteaza negativ nici compilarea, nici executia programului, fiind numai informatii plasate de programator in locuri despre care considera ca necesita explicatii.

Ceea ce urmeaza identificatorului de comentariu, reprezentat de doua slash-uri directe succesive (//), nu va fi interpretat de compilator.

Putem, astfel, comenta linii de cod inutile in loc sa le stergem, plasand // in fata lor.

Acest identificator de comentariu anunta mediul de programare ca este comentata o singura linie (care poate sa nu contina nimic, pe de alta parte). Pentru a comenta un intreg bloc de cod, se folosesc simbolurile **/* (slash direct-star)** pentru definirea regiunii comentate (la inceputul primei linii de comentat) si ***/ (star-slash direct)** pentru inchiderea comentariului (la sfarsitul ultimei linii de comentat). Desigur ca putem fie comenta linie cu linie portiuni intregi de cod, fie sa comentam o singura linie prin aceasta a doua metoda.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

4. TIPURI DE PROGRAME EXECUTABILE. BIBLIOTECI DE CLASE SI DE CONTROALE

Pe platformele Microsoft Windows, programe executabile sunt toate programele care indeplinesc functii. Astfel de programe sunt omniprezentele fisiere cu extensia .exe, fisierele de comenzi MS-DOS cu extensia .com (Command), .bat (Batch) si .pif (Program Information File), si nu in ultimul rand bibliotecile .dll (Dinamic Link Library). Insa, un dublu-click pe un fisier .dll nu va porni nici un program, decat daca sistemul este instruit sa faca astfel, sau daca respectiva biblioteca este un .exe cu extensia modificata si cu structura compatibila a unui executabil .exe tipic. Structura fisierelor .dll este, insa, aceeasi cu a fisierelor .exe, formatul lor respectand specificatia **PE** (**P**ortable **E**xecutable).

Cu C# se poate crea o larga varietate de programe, care vor avea extensia .exe sau .dll. Importanta acestora este oferita de accesibilitatea pentru utilizator: de regula, ne intereseaza ce face un .exe, si mai putin cum acesta apeleaza functii dintr-un .dll. Aceste **biblioteci de clase**, insa, sunt create cu scopul de a servi drept eminenta cenusie din spatele ferestrei pe care o intelegem ca program Windows. Ele contin, de regula, mare parte din functionalitatea programului. Metodele definite in biblioteci pot fi, desigur, parte integranta a executabilului .exe, insa printre avantajele utilizarii unei biblioteci sunt:

- reducerea dimensiunii executabilului, o parte din functionalitate fiind stocata separat
- intretinerea mai usoara a programului, prin inlocuirea numai a fisierului-biblioteca in cazul unei erori in functionalitate sau a modernizarii (*upgrade*)
- alocarea dinamica de memorie, functionalitatea necesara fiind apelata numai la nevoie, in loc sa se incarce intreg codul executabil ce poate contine functii niciodata apelate, impreuna cu interfata afisata intotdeauna.

Pe de alta parte, prin **biblioteca de controale** se intelege un executabil de tip .dll care asigura interactiunea cu utilizatorul printr-o interfata grafica functionala transpusa intr-un control vizibil.

Printre controalele comune ale sistemului de operare Microsoft Windows, cu care lucram zilnic, se afla: butoane (obiecte din clasa **Button**), casete de text (**TextBox**), etichete (**Label**), liste derulante (**ComboBox**) si simple (**ListBox**), bare de scroll (**VScrollBar**/**HScrollBar**) si meniuri (**Menu**/**menuItem**).

Fiecare astfel de control "elementar" apartine clasei superioare **Control**.

Evident, controalele primesc un *handle* dupa ce fereastra-parinte si controlul in sine sunt create, iar din acel moment pot fi accesate.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Controalele enumerate mai sus sunt simplissime si sunt "continute" (definite, specificate) si accesibile in/din spatiul de nume **System.Windows.Forms**.

Utilizarea unor controale din spatiul de nume Windows.Forms este ilustrata in Anexa 5.

O fereastra generica de program cu interfata grafica este la randu-i continuta in acelasi nume de spatiu si este denumita **formular** (*form*). Apartenenta acestora la acelasi spatiu de nume ofera acelasi statut cu cel al unei casete de text sau ale unei fereastre ce contine mai multe controale.

Aplicatiilor software complexe nu le sunt suficiente, insa, controalele simple puse la dispozitie de mediul de programare. Pentru diversificare si efectuarea unor interfete si operatiuni complexe, programatorii pot crea controale proprii (*user controls*) - de exemplu, unele care sa se ocupe de adaugarea unui tabel intr-o baza de date la un simplu click-dreapta pe un buton. Respectivul buton, insa, a fost plasat initial pe o interfata grafica reprezentand noul control, iar pe aceasta mai pot exista inca alte controale, simple sau derivate; respectivul control se va comporta ca un intreg atunci cand este plasat pe interfata altui program si poate fi transportat in mai multe programe care sa ruleze simultan.

Un exemplu de control este o bara de comenzi cu pictograme (*icons*) pe care putem executa click. Respectiva bara este, de fapt, o regiune grafica desenabila pe care se traseaza imagini la intervale regulate si pe care se definesc regiuni care raspund la mesaje ca WM_MOUSEMOVE si BM_CLICK.

Putem, desigur, crea un control care sa contina laolalta o bara de comenzi, un meniu si alte controale simple, sau chiar un formular care sa se constituie in *user control*. Varietatea controalelor este limitata de creativitatea programatorului si de functionalitatea intentionata.

Fiecare astfel de control va primi, la crearea sa, un identificator (*handle*) si va primi mesaje, la fel cu partile sale componente, al caror **parinte** este. De asemenea, fiecare control personalizat va mosteni clasa **Control**, beneficiind astfel de accesul la membrii acesteia (pozitia pe fereastra-parinte, vizibilitatea, culoarea si textul daca este cazul s.a.m.d.).

Dupa crearea si compilarea unui astfel de control, programatorul va avea la dispozitie un fisier - biblioteca .dll care contine controlul si functionalitatea sa. Diferenta intre biblioteca de clase si cea de controale este aceea ca, de obicei, prima nu poate fi implementata grafic, ci numai exploatata prin intermediul functionalitatii, grafice sau nu, a programului apelant.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Pentru a putea utiliza orice biblioteci trebuie sa se realizeze **referinte** la ele. Programul la care lucram nu va sti de existenta unei functii **f** continute intr-o clasa **c** incorporata intr-un spatiu de nume **sn** declarat intr-o biblioteca **b**, chiar daca fisierul .dll respectiv se afla in acelasi director cu programul. Vom face cunoscuta functia respectiva realizand referinta la **ansamblul** (biblioteca de clase sau controale) **b**.

Operatiunea de *referire* la biblioteci se efectueaza in mediul de programare Microsoft Visual Studio .NET prin click-dreapta in fereastra *Solution Explorer* si selectand optiunea de meniu *References*. Depinzand de natura controlului (nativ al platformei de programare .NET, control clasic COM/OCX de Windows sau proiect personal) vom bifa (sau naviga spre, si selecta) controlul sau fisierul dorit. De acum inainte, vom putea instantia un obiect **oc** derivat din clasa **c** din spatiul de nume **sn**, intrucat mediul de programare va sti despre ce locatie este vorba.

De exemplu, daca dorim sa utilizam in program functionalitatea expusa de biblioteca de grafica System.Drawing, vom face referire la acea biblioteca si vom scrie in sectiunea using instructiunea:

using System.Drawing;

In lipsa unei referiri explicite (creata cum s-a expus mai sus) la o biblioteca de functii (spatiu de nume), instructiunea using [spatiu de nume] **NU** ne va pune la dispozitie functionalitatea acestuia.

Exemple de creare a unor tipuri de programe si de realizare a referintelor sunt oferite in Anexa 8.

5. VARIABILE SI CONSTANTE

Variabilele sunt, cum am spus, echivalentul in sistemul de operare al marimilor fizice. Sunt entitati care pot lua valori incadrate in intervalul impus de tipul lor.

In C# se pot defini sapte categorii de variabile: statice, de instanta, elemente de tablou, parametri-valoare, parametri-referinta, parametri de output si variabile locale. Aceste categorii nu vor fi studiate in amanunt in cadrul acestui curs de baza.

In exemplul:

```
class A
{
	public static int x;
	int y;
	void F(int[] v, int a, ref int b, out int c)
	{
		int i = 1;
		c = a + b++;
	}
}
```

x este o variabila *statica*, y este o variabila *a instantei* (a clasei), v[0] este un *element* al unui tablou, a este un *parametru-valoare*, b este un *parametru-referinta*, c un *parametru-output*, iar i este o *variabila locala*.

!!! Un obiect derivat dintr-o clasa care dispune de o variabila-membru public va dispune de o *copie* a acestui membru. Clasa initiala (modelul ca atare) va dispune de propria valoare. !!!

Fiecare valoare ocupa un spatiu (o regiune) in memorie. Exista variabile carora li se rezerva (prin declararea explicita, sau **initializare**) 1 bit, 8, 16, 32 sau 64.

Bitul reprezinta cea mai mica unitate informationala interpretabila de procesor. Poate lua valoarea 1 sau 0 (aprins/stins, deschis/inchis, adevarat/fals) si sta la baza matematicii booleene.

Un **byte** sau **octet** reprezinta o secventa de 8 biti. O pereche de octeti formeaza o structura compusa, cu o portiune **high-order-byte** si una **low-order-byte**; aceasta alcatuieste, la randu-i, o unitate informationala ce cuprinde, insa, doua valori distincte si care poate fi totodata interpretata ca o singura valoare.

Intr-un sistem de numerotare binar pe 8 biti, numerele de la 0 la 10 sunt:

0: 00000000
1: 00000001
2: 00000010
3: 00000011
4: 00000100
5: 00000101
6: 00000110
7: 00000111

8: 00001000
9: 00001001
10: 00001010

Regula este: adunarea se face la dreapta, 1+1= 0 si se trece 1 in fata primei cifre nenule din numarul initial. 2 (10) devine 3 (11) prin adunare la dreapta cu 1: 0+1= 1.

Pe un sir de 8 biti se pot stoca 256 valori (2^8), de la 0 la 255.

Pe un sir de 64 biti se pot stoca 2^{64} valori.

In continuare vor fi expuse tipurile de valori din care pot fi create variabile, in limbajul C#.

5.1. TIPURI DE VALORI SI CONVERSII

*Intervalele de valori posibile pentru fiecare tip de date nu sunt aceleasi in toate limbajele de programare. Spre exemplu, din tipul de date **int** se poate genera un set de variabile cu valori intre –32768 si 32767 in Visual Basic, iar in Visual C# intervalul pentru **int** contine peste 4 miliarde de valori intregi.*

byte	0 la 255 Numar intreg fara semn, pe 8 biti
char	U+0000 la U+ffff Caracter Unicode, pe 16 biti (de exemplu, 'a' , '}'). *Numeric*
decimal	$\pm 1.0 \times 10^{-28}$ la $\pm 7.9 \times 10^{28}$ Stocare pe 128 biti; precizie: 28-29 zecimale
double	$\pm 5.0 \times 10^{-324}$ la $\pm 1.7 \times 10^{308}$ Stocare pe 64 biti; precizie: 15-16 zecimale
float	$\pm 1.5 \times 10^{-45}$ la $\pm 3.4 \times 10^{38}$ Stocare pe 32 biti; precizie: 7 zecimale
int	-2,147,483,648 la 2,147,483,647 Numar intreg cu semn, pe 32 biti
long	–9,223,372,036,854,775,808 la 9,223,372,036,854,775,807 Numar intreg cu semn, pe 64 biti
sbyte	-128 la 127 Numar intreg cu semn, pe 8 biti
short	-32,768 la 32,767 Numar intreg cu semn, pe 16 biti
uint	0 la 4,294,967,295 Numar intreg fara semn, pe 32 biti

ulong	0 la 18,446,744,073,709,551,615 Numar intreg fara semn, pe 64 biti
ushort	0 la 65,535 Numar intreg fara semn, pe 16 biti
string	Set de caractere Unicode (de tip **char**). Date - text ("abc").

Tipul **double** asigura deopotriva o precizie extraordinara si un interval extrem de larg (~$2*10^{316}$ valori). Spre ilustrare, numarul particulelor elementare cu masa din Univers este estimat a fi 10^{90}. Tipul **decimal** este preferat pentru calcule financiare, iar variabile de tip **short** sunt folosite in principal in parcurgeri de serii de date relativ scurte.

Doua tipuri speciale de date sunt **null** si **object**.

null reprezinta valoarea nula (sau **Nothing**, cunoscut programatorilor in Visual Basic "clasic" – pre-.NET) si poate fi folosita si pentru a distruge valoarea unei variabile (nu de orice tip, insa).

Tipul **object** poate fi folosit pentru a declara o variabila generica (tipul **Variant** in Visual Basic "clasic") si a-i atribui valori de diverse tipuri pe durata executiei programului (poate sa denote un **string**, un **long**, un **bool** etc.). Problema este ca sistemul nu stie de la inceput ce tip va avea variabila initializata ca **object**, alocand memoria strict necesara acesteia numai la atribuirea unei valori (acest procedeu se numeste *late binding* – legare tarzie la un tip de date).

Intre tipurile de valori se pot efectua **conversii**, de exemplu din **ulong** in **long**. Exista doua tipuri de conversie: **implicita** si **explicita**.

Conversia implicita presupune existenta deja a valorii de tipul x in intervalul de care beneficiaza noul tip (de exemplu, valoarea 32 declarata ca **ushort** este inclusa in intervalul rezervat datelor de tip **short,** si poate asadar fi utilizata ca **short**).

Conversia explicita (***cast***) este necesara in cazul tipurilor incompatibile sau atunci cand conversia are loc prin restrangerea la alt domeniu de valori (de exemplu, **long** in **int**).

In continuare sunt prezentate conversiile implicite.

- Din sbyte in short, int, long, float, double, sau decimal.
- Din byte in short, ushort, int, uint, long, ulong, float, double, sau decimal.
- Din short in int, long, float, double, sau decimal.
- Din ushort in int, uint, long, ulong, float, double, sau decimal.
- Din int in long, float, double, sau decimal.
- Din uint in long, ulong, float, double, sau decimal.
- Din long in float, double, sau decimal.
- Din ulong in float, double, sau decimal.
- Din char in ushort, int, uint, long, ulong, float, double, sau decimal.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

- Din float in double.

  O conversie explicita are in C# forma:

  (TipNou) VariabilaDeTipVechi

  De exemplu:

  sbyte sb= 32;
  byte sb2= (**byte**)sb;

  In continuare sunt prezentate conversiile explicite.

- Din sbyte in byte, ushort, uint, ulong, sau char.
- Din byte in sbyte si char.
- Din short in sbyte, byte, ushort, uint, ulong, sau char.
- Din ushort in sbyte, byte, short, sau char.
- Din int in sbyte, byte, short, ushort, uint, ulong, sau char.
- Din uint in sbyte, byte, short, ushort, int, sau char.
- Din long in sbyte, byte, short, ushort, int, uint, ulong, sau char.
- Din ulong in sbyte, byte, short, ushort, int, uint, long, sau char.
- Din char in sbyte, byte, sau short.
- Din float in sbyte, byte, short, ushort, int, uint, long, ulong, char, sau decimal.
- Din double in sbyte, byte, short, ushort, int, uint, long, ulong, char, float, sau decimal.
- Din decimal in sbyte, byte, short, ushort, int, uint, long, ulong, char, float, sau double.
  Un exemplu de cast este cel de la **short** la **int**:

  short s= 123;
  int i= **(int)**s;

In acest exemplu, valoarea variabilei *s* este continuta in domeniul de valori de tip **int**, astfel ca nu este trunchiata. In schimb, in exemplul

int i= 257;
byte b= (byte)i;

rezultatul va fi b= 1, deoarece domeniul tipului **byte** este 0…255. Valoarea rezultata prin cast este obtinuta prin trunchierea valorii initiale.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Metoda Parse()

Parse() poate ajuta la incercarea de a o variabila de un tip intr-una de alt tip. Variabila rezultata va detine toate proprietatile si metodele acelui tip de date.

De exemplu, urmatoarea secventa de cod transforma variabila de tip text (String) "2009-12-25" intr-una de tip data calendaristica (DateTime). La aceasta se poate apoi adauga un numar de 6 zile si 4 minute:

```
string s= "2009-12-25";
DateTime d= DateTime.Parse(s);
d.AddDays(6).AddMinutes(4);
```

Similar, un String poate fi transformat intr-o variabila de tip numeric:

```
string s= "4.2";
int i= int.Parse(s);                //rezultatul este 4
double d= double.Parse(s);          //rezultatul este 4.2
```

Metoda Parse() apartine tuturor tipurilor numerice de baza si tipului DateTime, astfel fiind utila cand de exemplu dorim sa interpretam ca un numar continutul unei casete in care utilizatorul scrie un text-valoare numerica.

Intrucat Parse() nu verifica inainte de executie valoarea argumentului primit si incearca sa il transforme oricum in tipul de date caruia apartine, este necesar sa efectuam operatiile de acest fel in interiorul unor blocuri de tratare a erorilor (try-catch-finally). De exemplu, urmatoarea secventa de cod va intrerupe executia programului din cauza unei erori de conversie:

```
string s= "abcd";
int i= int.Parse(s);   //eroare, "abcd" nu este vazut ca valoare numerica
```

Pe de alta parte, in urmatoarea secventa de cod aceasta problema se rezolva atribuind o valoare implicita in cazul in care nu se poate interpreta ceea ce doreste utilizatorul:

```
private bool bFacturaOK()
{
    string s= valoare_factura.Text;          //textul dintr-o caseta de editare
    double valoare_factura_implicita= -1;    //sau alta valoare de control
    double valoare_factura;
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
        try
        {
                valoare_factura = double.Parse(s);
        }
        catch  //daca utilizatorul nu a specificat o valoare numerica, eroare:
        {
                valoare_factura= valoare_factura_implicita;
        }
        finally //acest bloc se executa indiferent ce s-a petrecut - eroare sau nu:
        {
                if (valoare_factura== valoare_factura_implicita)        //-1
                {
                        //ceva pare sa fie in neregula, asa ca atentionam utilizatorul:
                        MessageBox.Show("Specificati o valoare pozitiva a facturii");
                        //totusi, iesim din aceasta metoda cu rezultat false:
                        return false;
                }
                else
                {
                        // pare sa fie OK, dupa interpretare s-a obtinut ceva relevant
                        return true;
                }
        }
}
```

5.2. DENUMIREA VARIABILELOR

Regulile sunt: numele variabilelor (si constantelor, de asemenea) trebuie sa inceapa cu un caracter literal, non-numeric, si sa nu apartina multimii cuvintelor-cheie ale limbajului (precum **int**, **return** si **namespace**; _int, _return si _namespace sunt, insa, nume permise de variabile, la fel cu @int, @return si @namespace; astfel de nume sunt, insa, contraindicate intr-un program decent).

Urmatoarele exemple de nume de variabile ilustreaza aceste reguli:

DECLARATII CORECTE:

```
int i= 0;
int _i= 1;
byte b245= 245;
byte b_240= 240;
string s= "abc";
string s_t= "def";
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

DECLARATII INCORECTE:

```
int 2i= 0;
int 700_i= 1;
byte +b= 40;
byte b%1= 10;
string ]str= "abc";
string str)123= "def";
string double= "asta chiar ca e imposibil";
```

Este evident ca se pot combina caractere sau siruri de caractere cu valori numerice, pentru a obtine nume de variabile imprastiate pe un spectru extrem de larg. Un programator atent, insa, declara numai variabilele de care are nevoie si numai in cadrul sectiunii unde vor fi necesare (de regula in corpul metodei sau al clasei care le utilizeaza).

Numele sunt *sensibile la tipul literelor* (*case-sensitive*), ceea ce inseamna ca majusculele sunt tratate diferit de minuscule; de exemplu, o variabila sau constanta denumita str1 va fi diferita de sTr1.

5.3. CONVENTII DE NUME PENTRU VARIABILE, METODE, CLASE ETC.

Termenul **conventie** inseamna ca desi programatorul are libertatea de a denumi variabilele utilizate intr-un program propriu dupa bunul plac, trebuie sa urmeze anumite reguli, care pot fi chiar auto-impuse. Totusi, atunci cand la un program lucreaza mai multe persoane si care este lasat pentru intretinere altui grup, conventiile clasice de denumire sunt:

- numele variabilelor incepe cu litera mica si poate contine majuscule
- numele constantelor este scris numai cu majuscule
- numele metodelor incepe cu litera mare
- numele claselor incepe cu litera mare
- numele spatiilor de nume incepe cu litera mare

De exemplu, numele unui control de tip comanda (intrare) de meniu (**menuItem**) ar putea fi *menuitem100* sau *mi100*; numele unor variabile de tip **ulong** ar putea fi *ul2*, *ul16*, s.a.m.d.

In bibliotecile de sistem, se urmeaza strict conventii de denumire, de exemplu la prima privire la numele BM_CLICK vom sti intotdeauna ca este vorba despre o constanta (numai majuscule) si de un mesaj destinat unui buton (BM_). Aceasta organizare este in primul rand utila programatorilor in C++ si in general celor care apeleaza in mod direct datele (functii si constante) continute in bibliotecile de sistem.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

5.4. GRADUL SAU NIVELUL DE ACCESIBILITATE/VIZIBILITATE

Mai devreme am mentionat cuvintele-cheie **private** si **public**.

Orice variabila declarata ca **private** este accesibila numai in **blocul de cod** in care este definita, adica in **sectiunea declarativa delimitata de acolade ({ si })**.

De exemplu, o variabila definita ca **private** intr-o clasa nu va fi afisata ca membru al acelei clase dupa instantierea unui obiect de respectivul tip. Pentru aceasta, variabila trebuie sa fie declarata **public** direct in corpul clasei (pe primul nivel, in exteriorul altor blocuri de metode si proprietati *get*/*set*).

Pe de alta parte, si variabilele declarate **private** sunt disponibile in toate sectiunile de cod *cuprinse* in cea in care sunt declarate.

Se spune ca un **domeniu de valabilitate** defineste vizibilitatea si durata de viata a obiectelor. Orice bloc (secventa de instructiuni cuprinse intre acolade) defineste un domeniu de valabilitate.

Variabilele se pot declara in orice punct in cadrul unui bloc definit de acolade, insa vor putea fi accesate numai de instructiuni *ulterioare* pe acelasi nivel (direct in acelasi bloc) si in blocuri interioare blocului unde sunt definite.

Variabilele declarate fara indicator de accesibilitate sunt private, adica vizibile numai in interiorul blocului de cod unde sunt declarate.

Indicatorii **internal** si **protected**, sau **protected internal** se refera la accesibilitatea variabilelor declarate in intreg programul. **static** indica o variabila care apartine tipului (clasei) din care este derivat un obiect. De exemplu, clasa:

```
public class MyBaseC
{
        public struct MyStruct
        {
                public static int x = 100;
        }
}
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Pentru a accesa/apela membrul static *x*, trebuie folosita intreaga cale spre acesta (daca nu este apelat din acelasi bloc unde este declarat – ca zin care poate fi accesat direct prin numele sau):

MyBaseC.MyStruct.x

Mai departe, fie o clasa care reprezinta un angajat al unei companii. Presupunem ca aceasta clasa contine o metoda de numarare a angajatilor si o proprietate care stocheaza acest numar. Atat metoda, cat si proprietatea, nu apartin nici unei instante-angajat. In loc, ele apartin clasei-companie. Asadar, acestea ar trebui declarate ca membri statici ai clasei.

Specificatorul de acces **protected** serveste in cazul in care se doreste ca membrii declarati astfel sa fie vizibili in clasele care mostenesc clasa in care acestia se afla, dar sa nu fie vizibili in alte scopuri. Cu alte cuvinte, membrii declarati astfel se comporta ca si cum ar fi publici, dar sunt accesibili numai din clasele derivate din clasa care ii contine.

De obicei, variabilele private si publice sunt cel mai folosite, natura lor fiind suficienta pentru a realiza un program coerent si stabil.

Regulile de aur ale utilizarii *responsabile* a variabilelor sunt:

Nu se declara mai multe variabile decat strictul minim necesar.

Pe cat posibil, este indicata atribuirea de valori noi acelorasi variabile, pentru reutilizare, in loc de a defini cate o variabila pentru fiecare valoare.

***Variabilele se fac disponibile (se declara) numai in regiunile in care este nevoie de ele, tinand cont ca instantierea claselor expune numai membrii publici ai lor*.**

5.5. CONSTANTE

Acestea sunt valori presupuse constante pe durata codului (de la declarare pana la sfarsit) si a executiei programului, care nu se pot modifica in nici un fel.

Astfel, in loc sa declaram o *variabila* PI ca:

double PI= 3.1415926;

a carei valoare o vom utiliza in intreg programul, nemodificata, mai logic este sa declaram aceasta valoare drept o *constanta*:

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

public const double PI= 3.1415926;

Astfel, compilatorul va sti intotdeauna ca dupa declararea lui PI nu se va interveni in cod asupra sa.

Daca, totusi, se incearca atribuirea unei valori unei constante, compilatorul va semnala o eroare.

Actiunea declaratiei **const** este diferita de cea a declaratiei **readonly**:

public readonly int y = 25; //initializare a unui camp numai-citibil

Atribuirea unei valori pentru un camp marcat **readonly** genereaza aceeasi eroare ca redeclararea valorii unei constante.

Campuri **readonly** nu pot fi declarate in interiorul metodelor.

O variabila nu va putea fi declarata de doua ori in blocuri de instructiuni din interiorul celor in care este declarat; sunt permise numai re-atribuirile de valori.

De exemplu:

```
class clasa
{
        string s= "abc";
        string s= "123";            //IMPOSIBIL
        s= "123";                   //PERMIS
}
```

iar:

```
class clasa
{
        string s= "abc";
        try //incercam sa redeclaram variabila s:
        {
                string s= "123";    //IMPOSIBIL
                s= "123";           //PERMIS
        }
        catch  //bloc de interceptare a unei erori; nu executa nimic:
        {
                //nici o instructiune
        }
}
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
class clasa
{
        try
        {
                string s= "123";
                s= "123";
        }
        catch
        {
                string s= "abc";       //PERMIS
                //"primul" s nu este vizibil in exteriorul blocului try
                s+= "def";    //nici "al doilea" s in exteriorul blocului catch
        }
}
```

Exemple de utilizare a variabilelor si constantelor sunt oferite in Anexa 2.

6. TABLOURI DE DATE; ENUMERARI; ALTE TIPURI DE COLECTII DE DATE, SPECIFICE PLATFORMEI MICROSOFT .NET

6.1. TABLOURI

Se defineste un tablou (**array**) de date ca fiind un ansamblu de valori de acelasi tip, ale carui elemente pot fi accesate dupa un **index**. Elementele continute intr-un tablou sunt accesibile dupa pozitia (indexul) acestora.

Trebuie tinut cont ca toate tipurile deriva din tipul **object**, astfel ca un tablou destinat stocarii de variabile de tip **object** va putea contine elemente de orice tip, **inclusiv tipul clasa**. In acest caz, declaratia tabloului ca fiind un tablou de date *de acelasi tip, adica **object***, este perfect valabila si tabloul va putea fi accesat ca oricare alt tip de tablou.

Un tablou se declara, generic, astfel:

Tip[] NumeTablou= {*element1, element2, ……* };

sau:

Tip[] NumeTablou= **new Tip[]** {*element1, element2, ……* };

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Tabloul se declara prin tipul elementelor, urmat de parantezele drepte deschisa si inchisa. Daca intre paranteze nu exista nici o valoare, tabloul se considera a fi unidimensional (o singura linie de matrice, sau un sir de valori).

Un exemplu de acces la un tablou bidimensional, de tip **int**, este:

```
int[,] n= new int[,] { {1, 2}, {3, 4}, {5, 6} };
n.SetValue(100, 0, 1);
        // am alocat valoarea 100 elementului situat pe linia 0 si
        // coloana 1 (la pozitia (0, 1) in tablou)
MessageBox.Show(n.GetValue(0,1).ToString());
        // am accesat valoarea elementului situat la pozitia (0, 1)
```

Un tablou unidimensional de constante care nu au acelasi tip se poate declara ca fiind de tip **object**, astfel:

```
object[] UnTablou= {1, 2, 3, "abc", 45, "x_yz", 6};
```

Dimensiunea unui tablou este data de numarul sau de elemente. Astfel, tabloul de mai sus are dimensiunea 7.

Elementele tablourilor pot fi accesate dupa un **index.** Primul element se afla la indexul sau pozitia 0, iar ultimul la pozitia 6. Se spune ca **tablourile sunt bazate-pe-0** (*0-based*), prin urmare accesarea elementelor se va face dupa un index ce porneste la 0.

De exemplu, daca dorim sa aflam valoarea celui de-al patrulea element, vom scrie:

```
object Elementul4= UnTablou[3];
sau
object Elementul4= (object)UnTablou[3]; //conversie explicita la object
sau
object Elementul4= UnTablou.GetValue(3);
```

(Metoda-membru **GetValue**(*pozitie*) al obiectelor de tip array este cea preferata pentru extragerea de valori.)

Ambele comenzi de mai sus ne vor furniza "abc", care este intr-adevar al patrulea element al tabloului, aflat la pozitia 3 in tablou, dupa enumerarea indexata cu baza 0.

Nu este permisa depasirea sfarsitului de tablou (ultimului element) la executie (specificarea unei pozitii mai mari decat ultima in tablou).

Elementele unui tablou pot fi modificate referindu-le in acelasi mod, de exemplu:

```
UnTablou.SetValue(3, "500000");
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Tablourile multidimensionale se initializeaza incluzand initializatorii fiecarui subtablou intr-un set propriu de acolade.

Un exemplu de tablou multidimensional este unul bidimensional, cu declaratia generica:

object[,] UnTablou= new **object** [**NumarLinii**, **NumarColoane**];

Asadar, se observa ca tablourile multidimensionale pot fi manipulate ca niste matrice. Declaratia de mai sus creeaza un tablou cu dimensiunea NumarLinii x NumarColoane, virgula avand rolul de a separa cele doua dimensiuni.

Pentru a obtine/atribui o valoare dintr-un astfel de tablou, se utilizeaza simplu coordonatele elementului:

Valoare= UnTablou[Linie, Coloana];

UnTablou[Linie, Coloana]= Valoare;

6.2. ENUMERARI

O enumerare (**enum**) este o structura coerenta de constante literale, care urmeaza regula: implicit, prima constanta are valoarea 0, a doua valoarea 1 si asa mai departe, indiferent de denumirile constantelor. Se poate stabili valoarea de inceput a enumerarii prin alocarea unei valori de tip **int** primului element.

Un exemplu de enumerare este:

enum Saptamina {Luni=2, Marti, Miercuri, Joi, Vineri, Simbata, Duminica};

In aceasta enumerare, Luni este 2 (valoare explicita de tip **int**), Marti 3 s.a.m.d. Sirul de valori Luni-Duminica se numeste **lista enumerarii**.

Aceasta declaratie defineste o enumerare ce are drept membri secventa Luni-Duminica.

In principiu, tablourile sunt preferate enumerarilor; si ele pot contine un numar mare de elemente cu denumiri sugestive, iar elementele unei enumerari sunt intotdeauna constante ce au valoare intreaga, pornind implicit de la 0 indiferent de numele lor. Enumerarile nu pot contine (si furniza in mod direct) valori literale (de tip **string**).

In cadrul codului, enumerarile se pot defini numai in exteriorul blocurilor de metode.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

6.3. TIPURI DE COLECTII DE DATE SPECIFICE PLATFORMEI .NET

Tipurile tablou si enumerare sunt utilizabile in contextul in care numarul de elemente ce se stocheaza este relativ mic. Dimensiunea maxima a tablourilor definibile depinde in mod direct de memoria disponibila in sistem.

Pentru operatiuni mai complexe (nu neaparat si mai complicate) ce implica seturi de date, platforma .NET pune la dispozitie alte tipuri de colectii de date, printre care:

- **SortedList**

Reprezinta o colectie de perechi cheie-valoare care sunt sortate dupa cheie si accesibile dupa cheie si index.

Un exemplu de SortedList este:

```
SortedList sl= new SortedList(); //instantiem un obiect de tip SortedList
sl.Add("1", 100); //adaugam un element ce are cheia "1" si valoarea 100
sl.Add("6", 300); //adaugam un element ce are cheia "6" si valoarea 300
sl.Add("2", new Button()); //adaugam un element tip Button cu cheia "2"
MessageBox.Show(sl["1"].ToString()); // extragem elementul cu cheia "1"
```

- **NameValueCollection**

Reprezinta o colectie sortata de **chei unice** si valori de tip **string** asociate fiecarei chei, colectia accesibila fie dupa cheie, fie dupa index. Poate fi folosita ca **dictionar** elementar de termeni textuali.

Un exemplu de NameValueCollection este:

```
System.Collections.Specialized.NameValueCollection nv=
                new System.Collections.Specialized.NameValueCollection();
nv.Add("1", "textul 1"); //adaugam o valoare
nv.Add("6", "textul 2"); //adaugam inca o valoare
nv.Add("1", "textul 3 tot la cheia 1"); //adaugam la valorile unei chei
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

- **StringCollection**

Reprezinta o colectie de **string**-uri (constante de tip text)
Un exemplu de StringCollection este:

```
System.Collections.Specialized.StringCollection sc=
                new System.Collections.Specialized.StringCollection();
sc.Add("textul 1");
sc.Add("Alt text, la pozitia 1 in colectie");
sc.Add(null);
MessageBox.Show(sc[0]); //se afiseaza "textul 1"
```

In loc de o constanta pe rand, la astfel de colectii pot fi adaugate intregi tablouri de date prin comanda **AddRange**.

7. ITERATII

Prin iteratie se intelege fiecare executare sau ciclare printr-o secventa de instructiuni sau a unei singure instructiuni, numarul de parcurgeri ale ciclului fiind stabilit de criterul de terminare a acestuia.

Astfel, se poate genera un cod care sa alature fiecarei valori din sirul de la 0 la 59 textul "Secunda numarul ", sau sa execute copierea unui fisier de 10 ori.

Declaratiile iterative sunt: **do**, **for**, **foreach**, **in**, **while**.

7.1. PERECHEA do-while

Traducerea literala este "executa instructiunile specificate atat timp cat este adevarata conditia".

do necesita ca respectivul ciclu sa fie terminat de o instructiune **while**, iar ciclul este parcurs cel putin o data.

while poate defini singur un ciclu, prin plasarea la inceputul blocului. Intrucat evalueaza o conditie cu rezultat boolean, ciclul **while** este parcurs de cel putin **0** ori.

Conditia care controleaza instructiunea **while** poate fi numai de tipul **bool**.

Structura blocului **do**-**while** este:

do {instructiuni} **while** (conditie adevarata)

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Un exemplu este:

```
int x= 2;
do
{
        x++; //incrementeaza valoarea lui x cu 1
}
while (x< 5); //cat timp x este mai mic decit 5
```

7.2. CICLUL for

Interpretarea este "**pentru fiecare valoare *x* din sirul atat-de-lung de valori** furnizat executa instructiunile specificate".

Iterarea se face printr-un domeniu de valori numerice.

Valoarea variabilei dupa care se cicleaza poate fi modificata in interiorul blocului *for*.

Structura acestui bloc este:

```
for (TipDeVariabila NumeVariabila= ValoareInitiala; ConditieDeVerificat;
                IncrementeazaVariabilaCuOValoare)
                { bloc de instructiuni }
```

Un exemplu simplu este:

```
for (int i= 0; i< 3; i++)
{
        MessageBox.Show(i.ToString());
}
```

In acest exemplu, valoarea intreaga **i** va fi incrementata cu 1 (i++) la fiecare parcurgere a ciclului, si va fi afisata apoi intr-o casuta de dialog. Ciclul va fi parcurs atat timp cat i este mai mic decat 3. Asadar, afisarea va fi efectuata de trei ori: prima oara pentru i= 0 (fiindca aceasta este valoarea initiala; putea fi 1, 2, 3 sau 100+140-16), a doua oara pentru i= 0+1, a treia oara pentru i= 1+1. Cand i devine 2, conditia i< 3 este indeplinita si ciclul se executa pentru ultima oara (de acum, i nu se mai incrementeaza).

Daca initializam **i** cu o valoare mai mare decat 3, ciclul nu se executa nici o data, intrucat conditia de executie a sa nu este indeplinita de la bun inceput.

Domeniul unei variabile declarate intr-o instructiune **for** se limiteaza la bucla **for** (blocul de instructiuni cuprins intre acolade). Variabila este necunoscuta in exteriorul acesteia.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

O bucla infinita este creata cu instructiunea **for** (; ;).

7.3. CICLUL foreach

Interpretarea este "**pentru fiecare obiect de tipul *x* existent in tabloul** specificat, verifica daca se indeplineste o conditie relativa la cele doua obiecte si executa instructiuni".

Iterarea nu se face ca la *for*, printr-un domeniu de valori numerice, ci prin elementele unui tablou al carui numar de elemnte NU este cunoscut (incat executarea procedurii cere mai mult timp decat in cazul *for* unde numarul elementelor/iteratiilor este cunoscut).

Valoarea variabilei dupa care se cicleaza nu poate fi modificata in interiorul blocului *foreach*, spre deosebire de cazul *for*.

Structura acestui ciclu este:

```
foreach (TipDeVariabila NumeVariabila in Tabloul NumeTablou)
{
        bloc de instructiuni
}
```

Variabila dupa care se cicleaza trebuie sa aiba acelasi tip cu elementele tabloului parcurs. Acest ciclu verifica daca valoarea variabilei este egala cu cea a unui element din tabloul parcurs; daca aceasta conditie este intalnita, ciclul poate fi intrerupt, fara a mai consuma timp de calcul verificand si restul elementelor, astfel:

```
int[] ia= {1, 2, 3};
foreach (int i in ia)
{
        if (i== 2)
        {
                MessageBox.Show(i.ToString());
                break; //nu mai verificam restul elementelor
        }
}
```

Instructiunea **break** este cea care intrerupe ciclul **foreach**, si indeplineste aceeasi functie daca este inclusa in cicluri **for** si **do...while**.

Instructiunea break intrerupe numai ciclul imediat (din al carui bloc de instructiuni face parte). Executia continua cu prima instructiune de dupa acolada de inchidere a ciclului.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Ciclul foreach parcurge in ordine toate elementele unui tablou, de la primul (index 0) la ultimul, nu invers.
In timpul lui foreach nu se pot modifica elementele tabloului parcurs.

Asociata instructiunii **foreach** este **in**, care realizeaza efectiv legatura dintre variabila a carei valoare este cautata si tabloul (sirul de valori) in care se cauta.

in poate fi utilizata si pe enumerari (**enum**) in afara de tablouri, insa aceasta nu este o practica curenta, enumerarile avand de obicei putini membri si nefiind folosite in mod curent.

Pentru a putea continua cu exemple concrete, este necesar sa prezentam ale notiuni elementare ale limbajului C# si sa revizuim/aprofundam unele dintre cele prezentate pana acum. La randul lor, capitolele urmatoare sunt aprofundate prin exemple in Anexe.

8. INSTRUCTIUNI DE EVALUARE SI CONTROL - BLOCUL IF - ELSE; BLOCUL SWITCH

8.1. BLOCUL if-else

Instructiunea **if** selecteaza pentru executie o instructiune bazandu-se pe valoarea unei expresii de tip boolean. Forma sa este:

```
if (expresie)
{
        instructiune1
}
else
{
        instructiune2
}
```

sau

if (expresie) *instructiune1;*
else *instructiune2;*

unde:
expresie este o expresie care poate fi convertita implicit la tipul bool;
instructiune1 reprezinta instructiunea (secventa de instructiuni) de executat atunci cand *expresie* este true.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

instructiune1 reprezinta instructiunea (secventa de instructiuni) de executat atunci cand *expresie* este false.

Blocul **if** este destinat efectuarii operatiunilor intentionate initial si este cel care verifica conditia din *expresie*.
Succesiunea celor doua blocuri este: **if** -> **else**.
Blocurile **if**-**else** pot exista numai in interiorul metodelor.
Blocurile **if**-**else** pot sa nu contina nici o instructiune.
expresie trebuie sa fie definita indiferent de continutul blocurilor **if** si **else**.
Un **else** se asociaza intotdeauna cu cel mai apropiat **if** din blocul sau care nu are deja asociat un **else**.
Blocul **else** trateaza eroarea intervenita in blocul **if** imediat precedent.
Blocul **else** nu poate exista independent de blocul **if**.
Prezenta blocului **else** este optionala.
Blocurile **if**-**else** pot fi imbricate, adica pot exista pe multiple nivele.

else poate reprezenta actiunea de indeplinit atunci cand nu se indeplineste nici o conditie dintr-o *scara* **if-else if**, adica:

if (expresie1)
{ *instructiune1* }
else if (expresie2)
{ *instructiune1* }
else if (expresie3)
{ *instructiune3* }
else
{ *instructiuni pentru cazul in care nu s-au indeplinit nici o conditie precedenta* }

1. Exemplu de bloc **if**

```
void MetodaCuIf()
{
	int i= 2;
	if (i== 2) //operatorul == intoarce o valoare de tip bool (true sau false)
	{
		i++; //incrementam i cu o unitate
	}
}
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

2. Exemplu de bloc **if-else**

```
void MetodaCuIfElse()
{
        int i= 6;
        if (i!= 3) //operatorul != intoarce o valoare de tip bool (true sau false)
        {
                i++;
/*(incrementam i cu o unitate daca valoarea sa nu este 3);
                aici, i devine 6+1= 7 */
}
else
{
        i= 70;
/* (atribuim lui i= 6 o valoare noua, 70) */
}
MessageBox.Show(i.ToString());
// va fi afisat "6"; i era intr-adevar diferit de 3 si a fost incrementat cu o unitate.
}
```

8.2. BLOCUL switch

Instructiunea **switch** este o instructiune de control, care ofera programatorului alegerea de a executa instructiuni structurate in multiple sectiuni, dependente de valoarea unei conditii evaluate la inceput.

Structura acestei instructiuni este:

switch (*expresie*)
{
        case *ExpresieConstanta*:
                instructiune
                instructiune de iesire din blocul ***case***
        default:
                instructiune
                instructiune de iesire din blocul ***case***
}

unde:
expresie este o expresie numerica (numar **intreg**) sau de tip **string**;
ExpresieConstanta stabileste, prin valoarea sa, blocul **case** caruia ii este transferat controlul;

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Instructiune reprezinta instructiunea (secventa de instructiuni) care va fi executata daca controlul va fi transferat blocului **case** sau **default**;
*Instructiunea de iesire din blocul **case*** transfera controlul executiei programului in exteriorul blocului **case**.

Blocurile **switch** pot exista numai in interiorul metodelor.
Blocurile **case** pot exista numai in interiorul unui bloc **switch**.
Blocurile **switch** pot sa nu contina nici un alt bloc.
Prezenta instructiunii **break** intr-un bloc **case** este obligatorie atunci cand controlul nu se transfera in exteriorul acelui bloc.
Prezenta instructiunii **goto case *x*** este justificata cand blocul nu contine instructiunea **break**.
Blocurile **case** si **default** pot sa nu contina nici o instructiune in afara de **break** sau **goto**.
goto este instructiunea de salt neconditionat. La intalnirea acesteia, in orice punct ar fi, fluxul executiei programului se transfera la locatia specificata de **goto**.
Blocul **default** este optional, dar recomandat. In interiorul sau se afla instructiunile care vor fi executate atunci cand *expresie* nu are nici una din *ExpresiileConstante*.

Atunci cand se detecteaza potrivirea cu una din instructiunile **case**, se executa secventa de instructiuni asociata acelei constante.
O secventa de instructiuni asociata unei constante **case** nu poate continua cu urmatorul **case**. Este necesara intreruperea secventei cu **break** sau **return**.

Exemplu de bloc **switch**:

```
void MetodaCuSwitch(string s)
{
	switch (s)
	{
		case "a":		//aceeasi evaluare ca if s== "a" (...)
			s+= "bc"; //s devine "abc"
			break; //executam instructiunea ce urmeaza blocului switch
		case "b":		//aceeasi evaluare ca if s== "b" (...)
		case "c":		//aceeasi evaluare ca if s== "c" (...)
			//putem specifica conditii succesive
			//dar si blocul de case multiplu trebuie parasit
			s= "def";
			break;
			//am tratat si cazul in care s= "b",
			//nu numai ultimul caz, in care s= "c"
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
            case "123_abc":             //la fel cu if s== "123_abc" (…)
                break; //nu se intimpla nimic, iesim din blocul switch
            default:                    //in orice alt caz, pentru orice alta valoare
                s= "XYZ";
                break;
        }
        //aici pot urma instructiuni care se executa dupa evaluarile din switch
}
```

Blocuri de evaluare **if - else** si **switch** pot exista unul in interiorul altuia.

9. OPERATORI

9.1. PREZENTARE GENERALA

Expresiile sunt construite din operanzi si operatori. Operatorii unei expresii indica operatiile ce vor fi aplicate operanzilor. Exemple de operatori sunt + (adunare), - (scadere), * (inmultire), / (impartire) si **new**, precum si chiar operatorul punct, care expune membrii unei clase. Exemple de operanzi sunt variabilele locale si proprietatile.

Exista trei categorii de operatori:

1. Operatori **unari**

Acesti operatori preiau un operand (*x*) si utilizeaza fie notatia **prefix** (de exemplu –*x*), fie notatia **postfix** (de exemplu *x*++)

2. Operatori **binari**

Acesti operatori preiau doi operanzi si toti utilizeaza notatia **infix** (de exemplu, *x* + *y*).

3. Operatorul **ternar**

Acest operator preia trei operatori, dintre care unul este o conditie a carei indeplinire este verificata. Si acesta utilizeaza notatia **infix**, iar forma sa este:

C*onditie* ? *DacaAdevarat_IntoarceValoarea_X* : *DacaFals_IntoarceValoarea_Y*

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Exemple de utilizare a operatorilor se gasesc in Anexa 1.

Lista operatorilor disponibili in C# este urmatoarea:

A. Operatori primari

x.y	accesarea membrului *y* al variabilei/membrului *x*
f(x)	invocarea metodei *f* trimitandu-i parametrul *x*
a[x]	accesarea unui element dintr-un tablou
x++	incrementarea lui *x* cu o unitate
x—	decrementarea lui *x* cu o unitate
new	crearea unei instante a unei clase/tip de date
typeof	evaluarea tipului unui obiect (**typeof**(*x*))
checked	folosit pentru restrangerea domeniului in care poate aparea o valoare care in mod natural nu se afla in acel domeniu (sub/supraevaluare)
unchecked	folosit pentru efectuarea operatiunilor matematice fara restrangerea domeniului de valori in cazul aparitiei unor erori de sub/supraevaluare

B. Operatori unari (un singur operand)

+	adunare
-	scadere
!	opus sau negatie
~	operator de complementare la nivel de bit
++x	incrementare prefix
--x	decrementare prefix
(T)x	conversie explicita (**cast**) in tipul T a lui *x*

C. Operatori multiplicativi

*	inmultire
/	impartire
%	modul

D. Operatori aditivi

+	adunare
-	scadere

*E. Operatori de deplasare / relocare a bitilor (**shift**)*

<<	mutarea unui bit la stanga
>>	mutarea unui bit la dreapta

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

F. Operatori relationali si de testare a tipurilor

<	mai mic decat
>	mai mare decat
<=	mai mic sau egal cu
>=	mai mare sau egal cu
is	verificarea tipului unui obiect
as	expresie *ca* tip

G. Operatori de egalitate

==	este egal cu ?
!=	este diferit de ?

H. "SI" LOGIC (AND)

&	combinare de conditii booleene. Adevarat daca ambii operatori (conditii) sunt **true**; altfel, intoarce **false**.

I. "SAU EXCLUSIV" LOGIC (XOR)

^	combinare de conditii booleene (*x* ^ *y*). Adevarat daca *x* este **true** si *y* **false**, sau daca x este **false** si y este **true**. Altfel, **false**.

J. "SAU" LOGIC (OR)

|	combinare de conditii booleene. Adevarat daca un operand este **true**

K. "SI" CONDITIONAL (AND)

&&	operator "de scurtcircuitare". Evalueaza numai prima conditie din (*x* && *y*) si continua la a doua (*y*) numai daca *x* este **true**.

L. "SAU" CONDITIONAL (OR)

||	similar cu &&, dar evalueaza *y* numai daca *x* este **false.**

M. OPERATORUL CONDITIONAL sau TERNAR

?:	expresia (ConditieIndeplinitaDeTipBoolean ? Rezultat1 : Rezultat2) va intoarce Rezultat1 daca conditia este **true** si Rezultat2 daca este **false**

N. OPERATORI DE ATRIBUIRE

=	atribuire simpla, de exemplu **int** i= 2;
*=	atribuire compusa: "valoarea curenta a variabilei din stanga este ea insasi inmultita cu ce urmeaza dupa ="

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

/=	similar inmultirii *=, valabil pentru impartire
%=	similar inmultirii *=, valabil pentru modulo
+=	similar inmultirii *=, valabil pentru adunare
-=	similar inmultirii *=, valabil pentru scadere
<<=	similar inmultirii *=, valabil pentru bit-shift stinga
>>=	similar inmultirii *=, valabil pentru bit-shift-dreapta
&=	similar inmultirii *=, valabil pentru evaluare logica
^=	similar inmultirii *=, valabil pentru evaluare logica
|=	similar inmultirii *=, valabil pentru evaluare logica

9.2. PRECEDENTA OPERATORILOR (ORDINEA OPERATIUNILOR)

Tabelul urmator expune operatorii in ordinea descrescatoare a *precedentei*.

Primari	x.y, f(x), a[x], x++, x--, new, typeof, checked, unchecked
Unari	+, -, !, ~, ++x, --x, T(x)
Multiplicativi	*, /, %
Aditivi	+, -
Bit-shift	<<, >>
Relationali	**<, <=, >, >=**
Testari de tip	is, as
Egalitate	==, !=
AND logic	&
XOR logic	^
OR logic	|
AND Conditional	&&
OR Conditional	||
Conditional	?:
Atribuire	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Regulile pentru operatiunile matematice sunt cele din aritmetica. Astfel, atat precedenta, cat si asociativitatea pot fi controlate prin paranteze.

De exemplu, x + y * z multiplica intai y cu z si apoi aduna rezultatul cu x, dar (x + y) * z aduna intai x cu y si multiplica rezultatul cu z.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

9.3. SUPRAINCARCAREA OPERATORILOR

Majoritatea operatorilor pot fi *supraincarcati*, sau *redefiniti*. Utilizatorul poate defini functionalitatea unor operatori, iar aceste versiuni personalizate *vor avea intaietate fata de cele implicite*.

Operatorii care pot fi supraincarcati sunt:

Unari: +, -, !, ~, ++, --, true, false
Binari: +, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=

In particular, nu este posibila supraincarcarea accesului la membri, invocarile de metode si nici operatorii **=, &&**, **||**, **?:**, **checked**, **unchecked**, **new**, **typeof**, **as** si **is**.

Un exemplu tipic este cel de redefinire a operatorului binar + pentru doua numere complexe (pentru semnificatia lui **this**, v. Anexa 6-Cuvintele cheie).

```
public struct Complex
{
        public int real;
        public int imaginary;

        public Complex(int real, int imaginar)
        {
                this.real= real;     //campul real al acestei (this) structuri
                this.imaginar= imaginar;    //campul imaginar
        }

        //declaram ce operator va fi supraincarcat (+),
        //tipurile care vor putea fi adunate (doua obiecte Complex-e)
        //si tipul valorii intoarse (Complex):
        public static Complex operator +(Complex c1, Complex c2)
        {
                return new Complex(c1.real+ c2.real,
                                   c1.imaginar+ c2.imaginar);
        }
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
        public static void Main()
        {
                Complex num1= new Complex(2,3);
                Complex num2= new Complex(3,4);
                //adunam cele doua obiecte de tip Complex (num1 si num2)
                //cu operatorul supraincarcat plus:
                Complex sumac= num1+ num2;
        }
}
```

10. INSTRUCTIUNI

Un program este alcatuit, in esenta, din instructiuni sau comenzi. Fiecare dintre acestea este o entitate informationala interpretabila de compilatorul C#, adica tradusa din text simplu, scris de catre programator, in instructiuni inteligibile pentru procesor. Ansamblul instructiunilor alcatuieste o unitate executabila - insusi programul final care poate fi rulat pe platforma .NET.

Exemple de instructiuni sunt:

```
int i= 2;
string UnSir= "un text anume";
if (i== 2) i++;
```

Fiecare instructiune din program, de exemplu atribuirile, ca:

```
string s= "un text";
```

referintele la biblioteci ale platformei .NET, ca:

```
using System.Graphics;
```

si intreruperea unui ciclu:

```
break;
```

trebuie incheiata cu simbolul ; (punct si virgula).

Exceptiile sunt putine, dar notabile:

- **namespace** *SpatiuDeNume*
- **class** *Clasa*
- declaratiile de metode, de exemplu:

```
void functie(int iarg1, string sarg2)
```

- inceputul si sfarsitul blocurilor de instructiuni (acoladele {, de inceput si }, de sfarsit)
- instructiunile de tratare a erorilor (**try**, **catch** si **finally**)

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

- **#region** [numele_regiunii_de_cod] si **#endregion** […]

O instructiune se poate intinde pe mai multe linii.
Importanta este respectarea sintaxei.
Mai multe instructiuni pot exista pe aceeasi linie, separate cu ;

Mediul de programare .NET detecteaza automat declaratiile de clase, variabile etc. si asteapta sa incheiem corect instructiunea respectiva. De exemplu, daca initiem o sectiune **namespace**, suntem obligati sa dam un nume acelui spatiu de nume si sa definim sectiunea ca atare, cuprinzand-o intre acolade. Daca nu efectuam aceste operatiuni, programul nu va fi compilat, semnalandu-ni-se erori de scriere.

Limbajul C# este sensibil la tipul literelor folosite in instructiuni (caractere/litere majuscule, minuscule).

In afara erorilor de scriere, care se afla la nivelul superior al erorilor in procesul de compilare, pot interveni erori in codul propriu-zis. De exemplu, putem atribui valoarea 200 unei variabile de tip **string**, in conditiile in care variabilele de tip **string** pot fi definite numai ca valori literale incadrate intre ghilimele; o valoare corecta ar fi "200".

Aceste erori se afla la un nivel mai jos si ne sunt, evident, semnalate de catre compilator, care se asteapta ca si noi ca programul sa ruleze cat de cat aproape de viteza de croaziera, fara incidente.

Putem usura sarcina compilatorului introducand in cod secvente de instructiuni care monitorizeaza fluxul de date prelucrate de program si erorile ce pot aparea.

In instructiuni pot exista spatii si caractere *Tab*; conteaza sintaxa. Linia:

```
int i= 2*3+14-9/6;
```

este mai greu de citit decat

```
int i= 2* 3+ 14- 9/6;
```

11. TRATAREA ERORILOR

Tratarea sau managementul erorilor (sau al exceptiilor - obiecte ale clasei *System.**Exception***) este una din sarcinile obligatorii ale programatorului care doreste sa stie ce se petrece de fapt atunci cand i se anunta o eroare si sa remedieze eroarea. "Ecranul albastru" care este afisat atunci cind "cade" sistemul de operare nu ofera informatii despre aplicatia care a cauzat eroarea si cu atit mai putin despre instructiunea care este vinovata.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Aceasta situatie se poate rezolva prin tratarea erorilor. Programatorul poate utiliza instructiunile **try**, **catch** si **finally** pentru a defini, respectiv:

- portiuni de cod care efectiv sunt testate (**try**);
- portiuni de cod unde se face managementul erorilor, de exemplu derivarea spre alte sectiuni si afisarea de mesaje despre eroare (**catch**)
- portiuni de cod care se executa indiferent daca erorile au fost tratate (**finally**).

Aceste blocuri de tratare a erorilor se plaseaza in cadrul blocurilor metodelor, intrucat acestea sunt cele care executa codul, si nu sectiunile **namespace** sau **class**, in sine.

Exemple de tratare a erorilor se afla in Anexa 3.

Secventa standard **completa** de tratare a erorilor cuprinde un bloc **try**, unul sau mai multe blocuri **catch** si un singur bloc **finally**.

Ordinea instructiunilor este ilustrata in exemplul urmator:

```
metoda NumeleMetodei(argumente)
{
        try
        {
                instructiuni ce urmeaza a fi testate
                (pot constitui, desigur, functionalitate efectiva a programului)
        }
        catch (TipExceptie ex)
        {
                instructiuni ce vor fi executate daca
                o eroare de tipul TipExceptie apare in blocul try
        }
        finally
        {
                instructiuni ce vor fi executate intotdeauna, dupa erori sau nu
        }
}
```

Erorile pot fi generate si voluntar, din varii motive care pot include terminarea intentionata a programului cu eroare si chiar distractia programatorului, folosind comanda **throw**. De exemplu, generarea unei erori de executie a programului se face cu instructiunea:

throw new ApplicationException();

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Bineinteles ca si aceste erori pot fi tratate ca unele comune, tot in cadrul unor blocuri **catch**.

catch poate avea ca parametru tipul erorii care trebuie interceptata, de exemplu, **catch** (System.IO.FileNotFoundException). In cazul in care este specificat tipul erorii de tratat, **numai** erorile de acest tip vor fi tratate; pentru alte tipuri de erori trebuie specificate blocuri **catch** specifice tipului fiecaruia. De asemenea, poate fi tratat un singur tip de eroare (**catch** tip_eroare), iar restul erorilor sa fie tratate intr-un singur bloc **catch** generic (fara parametru).

O eroare nu vine niciodata singura. Platforma .NET pune la dispozitia celor curiosi informatii despre tipul erorii si locul unde a fost generata. Aceasta informatie este stocata in membrii erorii, de exemplu **Message**, **Source** si **TargetSite**.

Message contine informatie sumara despre eroare (descrierea sa), intr-o limba inteligibila; **Source** contine denumirea spatiului de nume unde a aparut eroarea, iar **TargetSite** metoda in care a fost generata eroarea.

Toate informatiile despre eroare pot fi obtinute cu comanda *NumeEroare*.ToString().

Toate erorile interceptabile deriva din clasa **System.Exception**.

Blocurile* catch *si* finally *pot fi introduse in cod numai daca sunt precedate de un bloc* try*.
finally ***nu este neaparat necesar intr-o secventa de tratare a erorilor.***
Interceptarea tuturor erorilor, indiferent de tip, se face utilizand o instructiune* catch *fara parametri.
Monitorizarea blocurilor de cod pentru detectarea exceptiilor se face prin incorporarea lor intr-un bloc* try*.
Instructiunea catch intercepteaza exceptii. Executia va continua cu instructiunile situate dupa blocul* catch*.
Tipul exceptiei dintr-o instructiune* catch *trebuie sa coincida cu tipul exceptiei care se doreste a fi interceptata.
O exceptie care nu este interceptata conduce la terminarea fortata a programului.
Blocurile* try *pot fi imbricate.
Se poate utiliza cate un* catch *pentru fiecare tip de exceptie care se doreste a fi interceptata.
Blocul* finally *se executa dupa ce executia unui bloc* try-catch *se termina, indiferent de conditia care a produs terminarea.
***Intr-o secventa standard de tratare a erorilor poate exista un singur bloc* finally.**

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Utilizand aceasta metoda (*try-catch*) putem evita erorile fatale ale aplicatiilor pe care le programam. Teoretic, un astfel de bloc trebuie inclus in fiecare metoda susceptibila de a genera erori (accesarea fisierelor si a bazelor de date, operatii matematice complexe), intrucat diversitatea erorilor merge de la pierderea quasi-totala a memoriei sistemului pana la erori cauzate nemijlocit de neglijenta programatorului.

12. PREZENTAREA UNOR OBIECTE SI CONTROALE ESENTIALE DIN PLATFORMA .NET SI A PRINCIPALILOR MEMBRI AI ACESTORA

Am prezentat mai devreme facilitatile *AutoComplete* si IntelliSense, care ne permit ca, dupa tastarea unui punct imediat dupa numele unui obiect instantiat de noi sau cunoscut in mod implicit de catre mediul de programare, sa alegem dintre membrii acestuia, afisati intr-o lista.

Operatorul folosit pentru accesarea membrilor unei clase prin intermediul obiectelor este operatorul punct.

Problema principala este cunoasterea acestor obiecte de baza. Altfel spus, de unde stim ce anume ne-ar fi util pentru a realiza un lucru ?

Norocul programatorilor C# si in general al celor care utilizeaza pachetul Visual Studio .NET si orice mediu de programare modern este ca obiectele sunt structurate in **pachete** intr-un mod extrem. Astfel, nu vom gasi informatii despre clasa **Application** (care se refera la programul nostru in forma executabila) in clasa **String** sau **File**, nici informatii despre cum sa trimitem mesaje de posta electronica in spatiul de nume **Drawing**.

A. Fiindca am mentionat **Application**, sa incepem cu aceasta si sa-i studiem membrii esentiali.

Clasa Application este parte a spatiului de nume System.Windows.Forms.

1. **AllowQuit**

Este o proprietate booleana (de tip **bool**) fara accesor **set** dar cu accesor **get**, deci careia nu ii putem atribui o valoare; putem numai afla daca aplicatia poate fi parasita.

2. **CompanyName**

Este vorba, evident, de producatorul aplicatiei

3. **ExecutablePath**

Aceasta este calea spre fisierul executabil care a lansat aplicatia.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

4. **ProductName**
Denumirea aplicatiei
5. **StartupPath**
Locul (directorul) din care a pornit aplicatia. Este aceeasi cu ExecutablePath, dar nu include numele executabilului

B. **FileInfo**

Clasa **File** face parte din spatiul de nume System.IO. Din aceasta clasa nu se pot deriva obiecte. Totusi, lucrul cu fisiere nu este interzis, deoarece platforma .NET pune la dispozitie o larga varietate de fisiere si de metode de accesare a lor. Clasa **File**, insa, este o clasa protejata, sau inchisa, la fel ca si **String**. Prezentarea unora din membrii **File** se face la punctul C.

Protectia este implicit asigurata acestor clase de catre platforma Microsoft .NET, astfel ca nu putem defini obiecte **File** sau **String** carora sa le putem modifica *natura*. Ele sunt clase interne ale platformei .NET, intim legate de functionarea sistemului de operare, imuabile si accesibile de catre programator de regula numai in contextul unor verificari de tip.

Avand insa la dispozitie clasa **FileInfo**, care expune metode si proprietati, putem intr-adevar manipula sistemul de fisiere.

Instantierea unui obiect de tip **FileInfo** se face astfel:

System.IO.**FileInfo** fi= new System.IO.**FileInfo**(*NumeleFisieruluiAccesibil*)

1. **Attributes**
Ofera acces la atributele fisierului (ascuns, de sistem, read-only sau atribut compus)
2. **Create**()
Aceasta metoda creeaza fisierul *NumeleFiseruluiAccesibil*
3. **CreateText**()
Aceasta metoda creeaza un fisier de tip text
4. **CreationTime**
Proprietate care contine data crearii fisierului
5. **Delete**()
Metoda de stergere a fisierului
6. **Extension**
7. **FullName**
Proprietate care contine calea (structura de directoare) spre fisier
8. **LastAccessTime**
9. **Length**
Dimensiunea fisierului
10, 11. **CopyTo**(*Adresa*) si **MoveTo**(*AdresaNoua*)
Metode care copiaza, respectiv muta fisierul intr-o locatie

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

12. **Open**(*argumente de mod al deschiderii, tip al accesului s.a.*)
Metoda care face disponibil fisierul pentru citirea continutului
13. **OpenText**()
Deschide fisierul pentru citire in format text
14. **OpenWrite**()
Deschide fisierul pentru scriere

C. **File**

Aceasta este o clasa protejata, din care nu se pot deriva clase (nu poate fi mostenita) si nici instantia obiecte. Membrii sai statici sunt, implicit, accesibili in mod direct, clasa fiind dedicata manipularii generice a fisierelor.

File f= new **File**(); //IMPOSIBIL

1. **Copy**

Metoda de tip void. Copiaza un fisier, eventual suprascriind fisierul-destinatie.

File.**Copy**(*FisierSursa*, *FisierDestinatie*);

2. **Create**

Metoda care intoarce un obiect de tip FileStream. Creeaza un fisier.

FileStream fs= File.Create(*NumeleNouluiFisier*);

3. **CreateText**

Metoda care intoarce un obiect de tip StreamWriter.

StreamWriter sw= File.CreateText(*NumeleNouluiFisier*);

4. **Delete**

Metoda de tip void. Sterge un fisier.

File.Delete(*NumeleFisieruluiDeSters*);

5. **Exists**

Metoda care intoarce o valoare booleana. Verifica daca un fisier exista.

bool b= File.Exists(*NumeleFisierului*);

6. **GetAtrributes**

Metoda care intoarce un obiect de tip FileAttributes, adica atributele fisierului (*readonly*, *hidden* etc.).

FileAttributes fa= File.GetAttributes(*NumeleFisierului*);

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

7. **GetCreationTime**
Metoda care intoarce un obiect de tip DateTime.
DateTime dt= File.GetCreationTime(*NumeleFisierului*);

8. **Move**
Metoda de tip void. Muta un fisier, eventual redenumindu-l.
File.Move(NumeleFisierului, NumeleNouSauAdresaNouaAFisierului);

9.-12. **Open**, **OpenRead**, **OpenText**, **OpenWrite**.
Metode care deschid un fisier pentru accesarea informatiei in diverse moduri.

13. **SetAttributes**
Metoda de tip void, care primeste ca parametri numele fisierului caruia i se modifica atributele (*readonly*, *hidden* etc.).
File.SetAttributes(*NumeleFisierului*, *Atribute*);

D. **DateTime**
Aceasta este o **structura**. Cum deja s-a prezentat, structurile sunt entitati informationale asemanatoare claselor, dar informatia pe care o pun la dispozitie se presupune a fi mai putin diversa decat a unei clase comune. Membrii unei structuri sunt putin numerosi si indeplinesc o functionalitate simpla. Initializarea unui obect derivat dintr-o structura trebuie sa se faca atribuind valori parametrilor de initializare. O metoda de creare a unui obiect **DateTime** este:

```
DateTime dt= new DateTime(2005, 12, 31);     //31 Decembrie 2005
```

DateTime ofera acces la ceasul sistemului – putem afla timpul curent pana la milisecunda, din ziua 1 Ianuarie 1, ora 00:00:00 pana in 31 Decembrie 9999, ora 23:59:59.

1. **DaysInMonth**(**int** Anul, **int** Luna)
2, 3. **MaxValue**, **MinValue**. Aceste proprietati sunt membri si ai tipurilor de date numerice si reprezinta valoarea maxima, respectiv minima, pe care variabilele de respecivul tip le pot lua.
4. **Now**
Proprietate ce ofera data si ora sistemului, cu precizie de secunda
5. **Today**
Proprietate ce ofera data curenta
6. **UtcNow**
Proprietate ce ofera data si ora sistemului pe meridianul 0

E. **Form**

Clasa **Form** face parte din spatiul de nume **System.Windows.Forms**. Pentru a crea un obiect de acest tip si a-i putea accesa membrii trebuie sa introducem, in sectiunea **using** a programului, referinta spre acest spatiu de nume.

Prin formular sau forma sau **fereastra de dialog** (***Form***) intelegem orice fereastra de program care ruleaza in sistemul de operare Windows, afisata/vizibila pe monitor. Aceasta are forma clasica dreptunghiulara, dar nu forma conteaza, ci functionalitatea. Respectiv, un formular se poate constitui in parinte (container) pentru alte formulare si controale si poate chiar intercepta si bloca mesajele destinate acestora.

Exemple de **Form**-e sunt interfetele programelor din pachetul Microsoft Office, caseta de log-on care apare la pornirea sistemului, interfata miniaplicatiei care ruleaza cand afisam proprietatile *Desktop*-ului si fereastra in care cititi acest document (daca nu o faceti intr-un browser, program cu alt tip de interfata).

Formularele beneficiaza de o serie de membri care ofera acces exhaustiv la proprietatile si la modul de comportament al acestora. Acesta este, de altfel, motivul pentru care vom descrie aici numai membrii cel mai des folositi:

1, 2. **BackColor** si **ForeColor**

Culoarea de fundal si cea a textului scris direct pe formular

3. **Close**()

Metoda care inchide formularul si aplicatia, daca aceasta consta in respectivul formular

4. **ControlBox**

Proprietate de tip **bool** care ne permite sa modificam vizibilitatea butoanelor de minimizare/maximizare/inchidere a ferestrei

5. **Dispose**()

Metoda care elibereaza resursele folosite de formular.

Aceasta metoda este comuna si obiectelor derivate din clasa **Control**.

6. **Enabled**

Proprietate de tip **bool** care permite activarea/dezactivarea formularului (posibilitatea de prelucrare a mesajelor)

7. **Font**

8, 9. **Height** si **Width**

Inaltimea si latimea formularului, in pixeli

10. **Opacity**

Gradul de transparenta a formularului, de la 0% la 100%; de tip **double**

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

11. **Parent**

Parent este un membru de tip **Control**, si este de asemenea accesibila in insasi clasa **Control, fiindca orice control poate avea un control-parinte**. Ofera acces la containerul-parinte al formularului.

12. **ShowDialog**(*ArgumentOptional_ControlParinte*)

Metoda care afiseaza formularul deasupra tuturor celorlalte ferestre ale programului sau in interiorul unui formular/control-parinte.

Putem crea un formular cu urmatoarele instructiuni (se presupune ca ne aflam in interiorul unei metode dintr-o clasa):

```
//sectiunea using trebuie sa contina referinta spre
//System.Windows.Forms, incat compilatorul sa stie
//unde se gaseste clasa Form:
//
using System.Windows.Forms;
[…alte declaratii…]
Form f= new System.Windows.Forms.Form();
//am creat un obiect f de tipul Form;
//declaratia lui f nu trebuie sa respecte neaparat conventia de denumire
//in care specificam spatiul de nume din care face parte;
//totusi, o initializare de obiect despre care putem fi siguri
//ca va reusi este cea care contine, dupa cuvantul-cheie new,
//adresa completa a clasei respective
f.Width= 400; //latimea, in pixeli
f.Height= 200; //inaltimea, in pixeli
f.ControlBox= true; //pentru a putea inchide formularul, dupa afisare
f.Visible= true;
f.ShowDialog();
[…alte declaratii…]
```

F. **Button**

Clasa Button, ca si Form, este continuta de spatiul de nume System.Windows.Forms.

```
Button b= new System.Windows.Forms.Button();
```

Un obiect de tipul **Button** este convertibil (implicit si explicit) la tipul **Control**, la fel cu obiectele **Form**. Toate controalele care pot fi plasate pe un formular (butoane, casete de text, etichete, bare de scroll etc.) apartin clasei **System.Windows.Forms.Control** si contin in mod evident membrii generici ai acesteia.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Totusi, nu toti membrii unui **Button** apartin, de exemplu, unui obiect **VScrollBar** (bara de scroll verticala). De exemplu, proprietatea **Text**, care este membru al **Button**, nu exista in colectia de membri a **VScrollBar**; pe de alta parte, proprietatile **Top** si **Left** sunt comune acestor doua controale.

1. **FlatStyle**

Modul de afisare a butonului (normal sau plat).

Aceasta este o proprietate de tip enumerare (**enum**) ce cuprinde 4 membri: **Flat**, **Popup**, **Standard** si **System**.

Accesarea sa se face cu instructiunea (de exemplu, pentru un buton denumit *b* caruia i se aplica stilul plat):

b.FlatStyle= System.Windows.Forms.FlatStyle.Flat;

2. **Font**

3. **Image**

Imaginea afisata pe control, obiect de tipul **System.Drawing.Image**

4. **Text**

Textul afisat pe buton

5. **Click**

Acesta este un mesaj, sau **eveniment**, la care butonul va raspunde.

Evenimentele vor fi explicate in capitolul urmator.

G. **TextBox**

Casetele de text sunt una din metodele preferentiale de interactiune cu programele. Un astfel de control poate contine text care poate fi citit si modificat, poate fi activ sau inactiv si modificabil sau nu. Textul afisat este neformatat (nu exista **bold**, *italic*, ***underline***); pentru lucrul cu text formatat se foloseste controlul **RichTextBox**.

TextBox este apartine spatiului de nume System.Windows.Forms, iar instantierea unui astfel de obiect se face cu comanda (de exemplu):

TextBox tb= new **System.Windows.Forms.TextBox**();

1. **AcceptsReturn**

Proprietate de tip **bool** cu accesori **get** si **set** care specifica daca apasarea tastei ENTER va crea o noua linie in caseta de text.

2. **AppendText**(*text*)

Metoda de tip **void** care adauga text dupa cel existent deja in caseta

3. **CanUndo**

Proprietate de tip **bool** fara accesor **set**, care indica daca operatiunea precedenta din caseta de text (stergere, scriere s.a.) poate fi anulata

4. **Clear**()

Metoda care sterge continutul casetei de text

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

5. **Cut**()
Decupeaza si trimite in Clipboard textul selectat
6. **Dispose**()
Distruge caseta de text
7. **MaxLength**
Proprietate de tip **int** care specifica lungimea maxima a sirului de caractere care poate fi introdus in caseta de text.
Desi este de tip **int**, aceasta proprietate poate lua numai valori pozitive, inclusiv 0.
8. **Multiline**
Proprietate de tip **bool** care specifica daca respectiva caseta poate contine mai mult de o singura linie de text.
9. **PasswordChar**
Proprietate de tip caracter (**char**) care stabileste simbolul care va fi afisat in locul tuturor caracterelor din caseta de text.
10. **ReadOnly**
Proprietate de tip **bool** care stabileste daca se poate modifica continutul casetei
11. **SelectedText**
Proprietate de tip **string** care intoarce sau stabileste textul selectat
12. **SelectionStart**
Proprietate de tip **int** care intoarce sau stabileste pozitia de inceput a selectiei de text. Poate avea valori pozitive, inclusiv 0
13. **SelectAll**()
Metoda de tip **void** care selecteaza intreg textul
14. **Text**
15. **Undo**()
Anuleaza ultima operatiune (stergere, scriere s.a.)

In mediul de programare, afisarea automata a membrilor si functionalitatile *AutoComplete* si *IntelliSense* sunt suficiente pentru a studia lista membrilor unei clase.

In continuare este prezentata lista controalelor disponibile in bara de controale uzuale a mediului de programare Visual Studio .NET.

Button	buton
CheckBox	caseta care poate fi bifata
CheckedListBox	lista de valori care pot fi bifate (multi-selectie)
ColorDialog	dialog de selectare a unei culori
ComboBox	lista din care poate fi selectata o singura valoare
ContextMenu	meniu care este afisat la click-dreapta pe control
DataGrid	tabel in care sunt afisate valori
DateTimePicker	dialog de selectie a unei date calendaristice

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

DomainUpDown	selectie a unei valori predefinite
ErrorProvider	leaga erori in formular de alte operatiuni
FolderBrowserDialog	dialog de selectie a unui director
FontDialog	dialog de selectie a unui set de caractere (font)
GroupBox	control in care pot fi grupate ale controale
HelpProvider	stabileste fisierul de ajutor al aplicatiei
HScrollBar	bara de scroll orizontala
ImageList	lista de imagini
Label	eticheta (text afisabil, nemodificabil prin tastare)
LinkLabel	eticheta care directioneaza spre o adresa Internet
ListBox	lista de valori (multi-selectie)
ListView	tabel complex de valori (multi-coloana, multi-selectie)
MainMenu	meniul principal al programului (bara de meniu)
MonthCalendar	dialog de selectie a unei date calendaristice
NotifyIcon	icon pentru control din bara butonului Start
NumericUpDown	incrementarea vizuala a unei valori
OpenFileDialog	dialog de selectie a unui fisier, pentru deschidere
PageSetupDialog	dialog de stabilire a caracteristicilor unei pagini
Panel	cadru pe care pot fi plasate controale (nu grupate)
PictureBox	control desenabil, de regula continand o imagine
PrintDialog	dialog de tiparire
PrintDocument	stabileste documentul de tiparit
PrintPreviewControl	dialog de inspectare a documentului tiparit
PrintPreviewDialog	permite inspectarea documentului PrintDocument
ProgressBar	afiseaza gradul de realizare a unei operatiuni
RadioButton	control care poate fi bifat; de regula, intr-un grup RB
RichTextBox	caseta de text care poate contine text formatat
SaveFileDialog	dialog de selectie a unui fisier, pentru salvare
Splitter	control care poate redimensiona spatiul de lucru
StatusBar	bara de stare, care poate afisa informatii diverse
TabControl	dialog ce poate contine controale grupate dupa functii
TextBox	caseta de text neformatat
Timer	cronometru; la fiecare "tick" poate fi executat ceva
ToolBar	bara de instrumente (iconite)
ToolTip	control ce afiseaza informatie deasupra altuia
TrackBar	stabileste o valoare intr-un interval (Volume Control)
TreeView	structura arborescenta de date
VScrollBar	bara de scroll verticala

Nota: In general, toate resursele consumate de un control (vizual) pot fi eliberate (distrugerea... controlata, intentionata, expresa... a controlului) cu metoda Dispose().

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

13. FISIERE

Fisierele sunt entitati in care informatia din sistem este stocata in mod static (permanent). Fisierele pot fi accesate secvential sau aleator, adica datele continute pot fi citite si scrise in ordine sau dupa pozitia pe care o ocupa in secventa de biti ce constituie fisierul.

Platforma Microsoft .NET si implicit limbajul C# ofera o varietate larga de metode de accesare a fisierelor. Am prezentat deja clasa **FileInfo**, care ofera acces generic la fisiere.

Pe langa aceasta clasa, fisiere de anumite tipuri pot fi accesate prin clase specifice, de exemplu **TextWriter** si **BinaryReader**.

In cele ce urmeaza, vom prezenta, prin exemple concrete, clasele de baza pentru accesarea fisierelor.

Calea spre un fisier se indica de obicei printr-o variabila de tip **string**.

ATENTIE ! In orice instructiune care foloseste cai spre fisiere, structura de directoare trebuie specificata cu 2 caractere back-slash (\\ ) in variabila de tip string care este utilizata pentru a desemna cai spre fisiere. Daca nu se scrie un back-slash dublu, inaintea specificarii caii (inainte de ghilimelele de deschidere) trebuie plasat caracterul "@".

1, 2. StreamWriter si StreamReader

Aceste clase permit, respectiv, scrierea si citirea in format text (de tipul **string**) de informatie dintr-un fisier.

Spatiul de nume de care apartin este **System.IO** (operatiuni **I**n/**O**ut).

Un exemplu de scriere/citire este urmatorul:

```
class FisierScrisSiCitit
{
StreamWriter sw= new System.IO.StreamWriter("c:\\test\\fisier.txt");
/*
am instantiat un obiect StreamWriter care va referi fisierul denumit
fisier.txt situat in directorul c:\test;
urmarea acestei comenzi este crearea fisierului, cu dimensiunea 0 biti
*/
sw.WriteLine("o linie de text in fisier"); //scriem o variabila de tip string
sw.WriteLine("a 2-a linie de text");
sw.Close(); //inchidem fisierul
//
// vom citi din fisier a doua linie:
StreamReader sr= new System.IO.StreamReader("c:\\test\\fisier.txt");
//declaram o variabila string in care vom stoca fiecare linie citita:
private string s;
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
//citim o linie (prima, fiindca suntem imediat dupa deschiderea fisierului):
s= sr.ReadLine(); //ReadLine intoarce un rezultat de tip string
//citim a doua linie:
s= sr.ReadLine();
sr.Close(); //terminam accesul la fisier
//afisam cea de-a doua linie intr-un dialog;
//fiecare linie citita a fost stocata in aceeasi variabila de tip string
MessageBox.Show(s);
}
```

Altfel, in cazul in care nu stiam ca in fisier sunt numai doua linii de text, puteam cauta linia de interes in modul urmator:

```
[…]
StreamReader sr= new System.IO.StreamReader("c:\\test\\fisier.txt");
string s;

do
{
        s= sr.ReadLine();
        MessageBox.Show(s);
        //daca aceasta este linia, iesim din ciclul do-while cu break
}
while (s!= null); //citim linii atat timp cit nu gasim o linie vida
sr.Close();

[…]
```

In ambele exemple, variabila *s* de tip **string** este refolosita; nu este nevoie sa initializam cite o variabila pentru fiecare linie citita.

Putem distruge explicit referinta la variabila *s* introducand la sfarsitul operatiilor instructiunea:

```
s= null;
```

Pe de alta parte, puteam citi intreg continutul fisierului folosind metoda-membru, de tip **string**, ReadToEnd() din clasa StreamReader:

```
        string s= sr.ReadToEnd();
        MessageBox.Show(s);
        sr.Close();
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

3, 4. **BinaryWriter** si **BinaryReader**

Aceste clase permit, respectiv, scrierea si citirea generica de informatie dintr-un fisier (imagini, fisiere care contin text formatat).

Spatiul de nume de care apartin este tot **System.IO**.

StreamWriter si StreamReader permiteau, de asemenea, scrierea si citirea de informatie intr-un format generic, dar parametrii si rezultatul metodelor WriteLine, respectiv ReadLine erau de tip **string**.

Metodele BinaryWriter si BinaryReader permit accesarea informatiei fara a transforma implicit rezultatul in variabile de tip **string**.

Exemple de fisiere care pot fi accesate cu aceste doua metode sunt fisierele de text formatat de tip Microsoft Word in format .doc sau .rtf, tabele Excel si prezentari PowerPoint, arhive WinZip, documente Adobe Acrobat si imagini JPEG. **Orice tip de fisier** poate fi accesat in acest mod.

In fisiere se poate scrise cu metoda **Write** a clasei BinaryWriter. Aceasta metoda poate scrie variabile apartinand tipurilor de baza (**string**, **int**, **ulong**, **char**, **byte**, **bool** etc.), ca si tablouri de biti (**byte[]**) si de caractere (**char[]**).

Metoda de citire depinde de tipul de date stocate in fisier. Daca am scris variabile de tip **string**, vom folosi membrul **ReadString**(); daca dorim sa citim un numar de biti, vom folosi membrul **ReadBytes**(*NumarDeBiti*) s.a.m.d. In orice caz, dupa tastarea punctului dupa numele unui obiect BinaryReader vor fi afisati membrii din care putem alege metoda potrivita de citire.

Daca nu stim ce tip are informatia stocata intr-un fisier, putem folosi metoda ReadBytes pentru a crea un tablou de biti pe care ii putem prelucra ulterior (alipire pentru crearea unui **string**, manipulare independenta s.a.).

Un exemplu care ilustreaza utilizarea celor doua metode este dat in continuare.

Va fi folosit obiectul **FileStream**, care expune secvente de biti si sustine operatiunile de citire si scriere in fisiere.

```
[…]
//mai intii cream fisierul; daca exista deja, va fi suprascris:
FileStream fs= new System.IO.FileStream("c:\\test\\fisier.txt",
                                        FileMode.Create);
//cream un obiect BinaryWriter:
BinaryWriter sw= new System.IO.BinaryWriter(fs);
//scriem un string:
sw.Write("abc");
//scriem o valoare numerica:
sw.Write(123.456);
//inchidem fisierul:
sw.Close();
//inchidem fluxul de date:
fs.Close();
//
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
//
//cream un flux de date dinspre fisier:
fs= new FileStream("c:\\test\\fisier.txt", FileMode.Open);
BinaryReader sr= new System.IO.BinaryReader(fs);
//citim 10 biti din fisier, de la primul bit:
byte[] b= sr.ReadString(10);
string s= b.GetValue(2).ToString();
MessageBox.Show(s);
//va fi afisat "98", care este reprezentarea numerica,
//in codul ASCII, a literei mici "b"
//inchidem intii fluxul de date:
fs.Close();
//inchidem fisierul:
sr.Close();
[…]
```

Daca deschidem (cu Notepad, de exemplu) fisierul, vom observa secventa de caractere:

——————————abcw¾Ÿ/Ý^@

Secventa "abc" se distinge usor, insa 123.456 lipseste cu totul. Totusi, putem obtine si aceasta valoare daca o citim cu metoda ReadDouble, astfel:

```
[…]
fs= new FileStream("c:\\test\\fisier.txt", FileMode.Open);
BinaryReader sr= new System.IO.BinaryReader(fs);
//citim intai valoarea string:
b.ReadString();
//nu mai initializam nici o variabila pentru aceasta, nu are rost daca nu vom
//folosi ulterior valoarea citita
//citim acum valoarea numerica, de tip double (contine zecimale):
double d= sr.ReadDouble();
MessageBox.Show(d.ToString());
//va fi afisat "123.456"
[…]
```

In general, fisierele scrise cu BinaryWriter sunt lizibile in programele dedicate manipularii lor, adica unele care asteapta un anumit format al datelor accesate, si nu in unele dedicate citirii si interpretarii directe. Cu alte cuvinte, extensia fisierului nu dicteaza natura continutului, ci cel mult invers. Fisierul creat este pur si simplu unul ce contine o secventa de biti, iar Notepad a transformat variabila 123.456 intr-una de tip **double**, stocata pe 64 biti. Dimensiunea fisierului este de 12 octeti= 12*8 biti= 96 biti,

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

adica suma celor 64 biti necesari stocarii unei valori de tip **double** cu cei 32 biti rezervati pentru **string**.

Fisierele trebuie inchise dupa folosire, iar metoda **Close**(), de tip **void** si fara parametri, indeplineste tocmai aceasta functie. Aceasta metoda inchide fisierul (care, dupa deschidere, nu mai poate fi deschis inca o data pana la inchidere) si elibereaza toate resursele sistemului alocate utilizarii sale.

In general, indiferent de limbajul de programare, se recomanda ca **toate instructiunile prin care se manipuleaza fisiere** sa fie incluse in blocuri de interceptare si tratare a erorilor (**try** – **catch**), pentru a minimiza riscul de avariere a informatiilor. In limbajul Java (cu mediul de programare *NetBeans* sau *Eclipse*), aceasta abordare este chiar obligatorie.

14. RESURSE

Prin **resurse** ale unui program se intelege exact acest lucru - informatie stocata intr-un anumit loc si mod, destinata uzului ocazional al unuia sau mai multor aplicatii.

Exemple de resurse sunt fisiere de sunet si imagine, fisiere text sau documente de diverse tipuri si constante de tip **string** utilizate pentru a schimba limba in care este afisat textul pe controalele unei interfete.

Resursele se stocheaza in fisiere cu extensia **.resources** si **.resx**, care sunt create cu aplicatii dedicate, si sunt incorporate in executabile.

Resursele pot fi stocate si direct in interiorul executabilelor, fara intermedierea unor aplicatii specifice, setand proprietatea *BuildAction* a respectivului fisier ca fiind de tip *Embedded Resource*.

Accesarea resurselor se face prin intermediul membrilor spatiului de nume **System.Resources**. Pentru a putea lucra, asadar, cu resurse, este necesara instructiunea:

using System.Resources;

Acest spatiu de nume pune la dispozitie clasa **ResourceManager**, ai carei membri sunt dedicati exact accesarii resurselor.

Un exemplu tipic este urmatorul. Exemple mai detaliate si alte metode de accesare a resurselor vor fi prezentate in sectiunea aplicativa a cursului.

```
//Aceasta metoda este dependenta de locatia fisierelor .exe si .resources
//Pentru a se putea accesa fisierul de resurse, se salveaza fisierul .resx
//ca "resource1.resources" in directorul executabilului, unde va fi cautat
//la compilare
//Se presupune ca in fisier exista o resursa-text (de tip string) denumita "1000",
//careia ii corespunde textul "un text din resurse"

private void button1_Click(object sender, System.EventArgs e)
{
        ResourceManager rm =
                        ResourceManager.CreateFileBasedResourceManager(
                                                        "resource1", ".", null);
        MessageBox.Show(rm.GetString("1000").ToString());
        //rezultatul va fi afisarea textului "un text din resurse"
}
```

Exemple de utilizare a resurselor se afla in Anexa 9.

15. SERIALIZAREA SI DESERIALIZAREA OBIECTELOR

Prin ***serializare*** se intelege procesul de conversie a unui obiect intr-o forma care poate fi transportata logic sau fizic (fisiere, pachete de date prin Internet) sau, mai simplu, **procesul de stocare a starii unui obiect pe un mediu anume**.

In acest proces, campurile private si publice ale obiectului, precum si numele clasei si ansamblul ce contine clasa, sunt convertite intr-o secventa de biti, care este scrisa intr-un flux (*stream*) de date.

De exemplu, putem serializa un obiect si sa-l transportam prin Internet, cu protocolul HTTP, intre un server si un client.

Deserializarea este procesul invers, care reconstruieste obiectul din secventa de biti ce constituie forma serializata, *daca se cunoaste tipul concret al acestuia (adica nu Object, ci un tip de date existent in .NET)*.

Motivele pentru care aceste metode se utilizeaza sunt persistenta starii unui obiect pe un mediu de stocare, astfel incat oricand sa se poate re-crea o copie exacta a obiectului, si de asemenea transferul obiectului intre programe.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Cel mai usor mod de a face serializabila o clasa (deci implicit obiectele create pe modelul sau) este marcarea sa cu atributul [Serializable]. Anumite obiecte insa, precum *DataSet*, poseda propriul mod de serializare.

```
[Serializable]
public class UnObiect
{
        public int i= 0;
        public int j= 0;
        public string s= null;
}
```

Codul urmator prezinta modul in care se poate serializa intr-un fisier o instanta a acestei clase.

```
UnObiect obj = new UnObiect();
obj.i= 1;
obj.i= 24;
obj.s= "Un String";
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("fisier.bin",
                                        FileMode.Create,
                                        FileAccess.Write,
                                        FileShare.None);

//stocam starea obiectului in fisierul "fisier.bin" specificat:
formatter.Serialize(stream, obj);
stream.Close();
```

Clasa *BinaryFormatter* serializeaza si deserializeaza un obiect in format binar.
Vor fi serializati toti membrii clasei, chiar si cei privati. Restaurarea obiectului se face creand un *formatter*, pe care il instruim sa deserializeze obiectul.

```
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("fisier.bin",
                                        FileMode.Open,
                                        FileAccess.Read,
                                        FileShare.Read);
//re-cream starea obiectului (stim ca fisierul contine un obiect UnObiect)
UnObiect obj = (UnObiect) formatter.Deserialize(stream);
stream.Close();
```

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

A doua metoda de stocare a starii unui obiect este cea care foloseste clasa SoapFormatter (SOAP= *Simple Object Access Protocol*), in locul BinaryFormatter.

Aceasta metoda este utilizata in cazurile in care se doreste stocarea starii intr-un fisier cu structura XML (E**x**tended **M**arkup **L**anguage), cu continutul de tip text neformatat, care poate fi transportat pe Internet indiferent de caracteristicile retelei si de protocolul utilizat (ignora firewall-urile si uzeaza de protocolul HTTP).

Structura unui fisier XML este de tip arborescent, similar cu cel al unui fisier HTML, in care elementul de baza este <html>; "trunchiul" este numele obiectului, iar "ramurile" - numele campurilor si atributele acestora.

Pentru cazul de fata, daca se doreste stocarea starii intr-un fisier XML, se va folosi un SoapFormatter in locul BinaryFormatter, iar extensia fisierului va fi *xml*.

16. DATASET-uri (Seturi de date). Gestionarea datelor cu Microsoft SQL Server

16.1. DATASET-uri

Un *dataset* este o colectie de date deconectata de la sursa acestora.

Un dataset tipic stocheaza datele dintr-o baza de date - tabele si campurile acestora, relatiile dintre campuri etc.

Structura unui dataset este similara unei baze de date relationale, expunand modelul ierarhic al tabelelor, randurilor si coloanelor.

Dataset-urile se utilizeaza atunci cand se lucreaza cu un set de tabele fara conexiune la sursa datelor.

Clasa **Dataset** implementeaza interfata ISerializable, beneficiind astfel de posibilitatea de a-si serializa continutul in fisiere cu format XML.

Datasetul este o componenta vitala a sistemului de gestionare a datelor ADO.NET (**A**ctiveX **D**ata **O**bjects-.NET), care poate reprezenta datele in format XML. Astfel, se poate citi o secventa de biti care sa fie transpusa ulterior intr-un dataset.

Un dataset care va contine informatii dintr-o baza de date se poate crea numai dupa ce s-a stabilit legatura cu baza de date; in exemplele care urmeaza se va folosi modelul de stocare SQL Server, dar sistemul ADO.NET permite conectarea la diverse surse de date, precum Oracle, Microsoft Access s.a.

Conexiunile cu baza de date se fac prin intermediul obiectelor de tip ***x*DataAdapter** si ***x*Connection**, unde *x* denota numele furnizorului (*provider*-ului) de suport pentru transferurile de date, de exemplu *Sql*DataAdapter si *Sql*Connection, *Odbc*Connection si *Oracle*Connection.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Astfel, bazele de date Microsoft Access pot fi accesate cu obiecte de tip *Odbc*, iar bazele de date stocate in sistemul SQL Server cu obiecte de tip *Sql*. Reciproca nu este valabila, deoarece legaturile la sursele de date sunt proiectate expres pentru fiecare sursa.

Intrucat datele cu care este populat un DataSet se afla in memoria computerului care a efectuat operatiunea de extragere a datelor, dupa ce nu mai este nevoie de ele DataSetul (*ds*) poate fi distrus astfel:

```
ds.Clear();
ds.Dispose();
```

16.2. GESTIONAREA DATELOR CU MICROSOFT SQL SERVER

In cele ce urmeaza vom crea un dataset din tabelele unei baze de date SQL Server si vom enumera numele tabelelor.

Inainte de initierea transferurilor trebuie deschisa conexiunea cu baza de date pe care intentionam sa o folosim. Presupunand ca pe serverul local exista baza de date "Produse", o conexiune SQL catre aceasta se poate deschide astfel:

```
System.Data.SqlClient.SqlConnection sqlc=
        new System.Data.SqlClient.SqlConnection(
        Initial Catalog=Produse;
        Data Source=localhost;
        Integrated Security=SSPI;");
```

Intre paranteze se afla "sirul de conexiune", ilustrat de proprietatea *ConnectionString* a unei conexiuni de orice tip. In acesta trebuie specificati parametrii conexiunii, dintre care cei mai importanti sunt:

- sursa datelor (aici, baza de date "*Produse*");
- numele serverului (sau adresa IP) care stocheaza datele (aici, *localhost* sau *127.0.0.1*, pentru serverul local);
- tipul de protocol de acces securizat prin care se va face interogarea bazei de date. Pentru a acorda in mod explicit acces unui utilizator, in loc de "Integrated Security=SSPI;" am fi folosit:

"user id=[*numele utlizatorului*];password=[*parola*];"
(sau:
"user=[*numele utlizatorului*];password=[*parola*];")

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Cu aceasta comanda va fi deschisa o conexiune la baza de date denumita "**Produse**" aflata pe sistemul "local". Daca operatiunea reuseste (conditiile esentiale fiind ca serviciul SQL Server sa fie pornit, baza de date sa existe si ca solicitantul sa aiba dreptul de a o accesa), tabelele aflate in baza de date vor fi accesibile pentru extragerea datelor.

Intr-un tabel, datele sunt stocate in **campuri** (*fields*) sau **coloane** (*columns*). Un tabel al unei baze de date SQL Server (si, de altfel, al unei baze de date Microsoft Access si Oracle) are structura urmatoare: coloanele reprezinta **campurile**, iar liniile reprezinta **inregistrarile** (*records*).

Campurile pot avea mai multe tipuri: *numeric*, *text*, *char*, *datetime*, *decimal* s.a..

Toate valorile unui camp vor respecta tipul acestuia; de exemplu, un text nu va putea fi stocat intr-un camp de tip numeric.

Sa consideram baza de date "**Produse**" care are un tabel denumit "**Inghetata**". Campurile acestui tabel sunt: **Denumire**, **Sortiment**, **Compozitie**, **Pret**, **DisponibilInStoc**. Fiecare inregistrare desemneaza un tip de inghetata, cu caracteristici definite de cele patru campuri:

Structura tabelului **Inghetata** este:

Denumire	**Sortiment**	**Compozitie**	**Pret**	**DisponibilInStoc**
Vanilie 1	Vanilie	Lapte, vanilie	2000	true
Vanilie 2	Vanilie	Lapte, vanilie	3000	false

.....

Sa presupunem ca vom dori ca datasetul nostru sa contina toate inregistrarile din tabelul "**Inghetata**". Pentru a extrage datele respective, va trebui intai sa construim un obiect de tip **SqlDataAdapter**.

Aceasta clasa reprezinta un set de comenzi efectuate pe date si o conexiune care vor fi utilizate pentru a umple efectiv datasetul si a modifica informatia din baza de date.

Un obiect de acest tip reprezinta puntea de legatura dintre DataSet si SQL Server pentru extragerea si salvarea datelor. Aceasta punte se realizeaza prin apelarea metodei **Fill**, care modifica datele stocate in dataset pentru a reflecta datele din sursa, si a metodei **Update**, care modifica datele din sursa pentru a reflecta datele din dataset.

DataSet-ul rezultat va avea structura datelor extrase - daca vom extrage inregistrarile din tabelul Inghetata, in dataset datele vor fi stocate in acelasi mod ca in tabelul original.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Pentru a extrage efectiv date avem nevoie de un obiect de tip **SqlCommand**. Aceasta clasa reprezinta o comanda Transact-SQL sau o *procedura stocata* care vor fi apelate pentru a selecta datele din sursa.

Pentru a utiliza un obiect SqlCommand trebuie furnizate: textul comenzii, in care se specifica datele ce vor fi extrase, actualizate sau sterse, si obiectul de tip SqlConnection asociat. Evident, un obiect de acest tip poate fi refolosit prin specificarea unor diverse comenzi si conexiuni.

Presupunand ca extragem toate datele din tabelul Inghetata, un obiect SqlCommand va fi creat astfel (conexiunea *sqlc* a fost definita anterior):

```
string scomanda= "SELECT * FROM Inghetata";
System.Data.SqlClient.SqlCommand sqlc=
        new System.Data.SqlClient.SqlCommand (scomanda, sqlc);
```

Intreg codul necesar umplerii unui DataSet cu inregistrarile este:

```
public DataSet DataSetInghetata()
{
        System.Data.SqlClient.SqlConnection sqlc=
                new System.Data.SqlClient.SqlConnection(
                Initial Catalog=Produse;
                Data Source=localhost;
                Integrated Security=SSPI;");
        System.Data.SqlClient.SqlDataAdapter sqlda=
                new System.Data.SqlClient.SqlDataAdapter();
        System.Data.SqlClient.SqlCommand sqlcom=
                new System.Data.SqlClient.SqlCommand
                                        ("SELECT * FROM Inghetata", sqlc);
        sqlda.SelectCommand= sqlcom;
        sqlda.Fill(ds);
        return ds;
}
```

Metoda **Fill** creeaza structura DataSet-ului si il umple pe acesta cu inregistrarile dorite.

De acum, structura acestui dataset va fi tabelara: un singur tabel (Inghetata) continand campurile si inregistrarile preluate intocmai din baza de date.

Comanda **SELECT * FROM Inghetata** este o comanda tipica de selectie a tuturor (*) inregistrarilor dintr-un tabel.

In cele ce urmeaza se vor prezenta pe scurt modelele de sintaxa SQL pentru manipularea simpla a inregistrarilor.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Indicatii pentru crearea bazelor de date in sistemul Microsoft SQL Server 2005/2012 Express Edition se afla in Anexa 10.

SINTAXA SQL DE BAZA

1. Selectarea inregistrarilor

Comanda este **SELECT**, urmata de numele campurilor de interes, de eventuale clauze de selectie, de **FROM** si de numele tabelului din care se vor extrage inregistrarile.

Datele obligatorii sunt numele campurilor (sau * pentru preluarea tuturor campurilor) si numele tabelului.

Fie tabelul **X**, continand campurile A, B si C.

1.1. Selectarea tuturor inregistrarilor din tabelul X:

SELECT * **FROM** X

echivalenta cu

SELECT A, B, C **FROM** X

1.2. Selectarea inregistrarilor din unele campuri:

SELECT A, C **FROM** X

1.3. Selectarea inregistrarilor avand o caracteristica comuna:

SELECT * **FROM** X **WHERE** (C=200)

Gruparea in paranteze a clauzelor de selectie nu este neaparat necesara, dar mareste lizibilitatea si reduce sansele de eroare.

1.4. Selectarea inregistarilor cu mai multe caracteristici comune:

SELECT * FROM X WHERE (C=200) AND (B='Vanilie') AND (A='Vanilie1')

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

1.5. Selectarea unor inregistrari si ordonarea lor dupa campul C:

SELECT * **FROM** X **WHERE** (C<200) **ORDER BY** C **DESC**

DESC si **ASC** sunt optionale si pot fi specificate ca ultim parametru al acestei comenzi - pentru ordonare descrescatoare, respectiv crescatoare dupa valorile (numerice, literale, de timp s.a.) din campul respectiv.

1.6. Selectarea primelor N inregistrari din tabelul X:

SELECT TOP N **FROM** X

1.7. Selectarea fractiunii N din inregistrarile din tabelul X:

SELECT TOP N PERCENT FROM X

2. Actualizarea inregistrarilor

Comanda este **UPDATE**, urmata de numele tabelului care contine inregistrarile de modificat, de **SET**, de numele campurilor de modificat si de noile valori, si de eventuale clauze de selectie.

2.1. Modificarea datelor dintr-un camp (C) pentru toate inregistrarile din tabel (X):

UPDATE X **SET** C=300

2.2. Modificarea datelor dintr-un camp utilizand o clauza de selectie:

UPDATE X **SET** C=300 WHERE (C<300)

Valoarea tuturor inregistrarilor care in campul C au o valoare mai mica decat 300, va fi modificata la 300.

3. Stergerea inregistrarilor

Comanda este **DELETE**, urmata de FROM, de numele tabelului si de eventuale clauze de selectie. Vor fi sterse toate campurile inregistrarilor selectate, nu este posibila selectia unor anumite campuri; de aceea, spre deosebire de comanda SELECT, specificarea simbolului * nu este necesara:

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

3.1. Stergerea tuturor inregistrarilor dintr-un tabel:

DELETE FROM X

3.2. Stergerea inregistrarilor avand o caracteristica comuna:

DELETE FROM X **WHERE** (C<300)

4. Adaugarea de inregistrari

Comanda este **INSERT**, urmata optional de **INTO**, apoi obligatoriu de numele tabelului in care se vor insera date, de lista numelor campurilor inregistrarii, de **VALUES** si de valorile din fiecare camp. Prezenta clauzelor de selectie (WHERE, ORDER BY) nu este permisa.

INSERT INTO X (A, B, C) **VALUES** ('Vanilie 3', 'Vanilie', 4000)

Numele campurilor si valorile furnizate trebuie incadrate in paranteze. Ordinea de atribuire a valorilor se va face urmarind ordinea campurilor specificate in comanda. **INTO** este optional, dar numele tabelului (X) este obligatoriu de specificat.

Pentru manipularea datelor cu **INSERT** si **UPDATE**, regula de baza este: campurile de tip text vor primi valorile incadrate in apostrofuri, iar campurile numerice vor primi simplu valorile numerice.

Campurile pot accepta valoarea **NULL**, adica pot fi efectiv nule, neputandu-se extrage nimic din ele. Aceasta proprietate se poate stabili la crearea unui tabel - fie in mediul SQL Server, fie programatic (din cod).
Comanda **INSERT** se va comporta astfel la intalnirea unui camp cu caracteristica *Allow* Nulls:
- Comanda **INSERT** trebuie neaparat sa specifice valori pentru campurile care NU SUNT de tipul *Allow Nulls*;
- Comanda **INSERT** poate sa specifice o valoare pentru respectivul camp; acesta este numai conceput ca putand accepta "valori nule". Daca se furnizeaza o valoare, aceasta va putea fi extrasa din camp.
Similar se comporta si comanda **UPDATE**. Comanda **SELECT** preia orice valoare specificata, indiferent de tipul campurilor.

Unui obiect de tip **SqlCommand** i se poate furniza un text (ca variabila de tip *string*) de executat avand formele expuse.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Metoda care executa o comanda pe baza de date este **ExecuteNonQuery**(), metoda de tip **int** care pur si simplu executa comanda **SELECT**, **UPDATE**, **DELETE** sau **INSERT** si intoarce numarul inregistrarilor afectate.

Exista si alte metode specifice obiectului **SqlCommand**:

- **ExecuteReader**(): intoarce un obiect de tip **SqlDataReader** care poate fi utilizat pentru a parcurge lista inregistrarilor;

- ExecuteScalar(): intoarce un obiect de tip **Object** ce consta din prima coloana (primul camp) a primei linii citite;

- Execute XmlReader(): intoarce un obiect de tip **System.Xml.XmlReader**.

Obiectul (**Sql**)**DataReader** pune la dispozitie lista inregistrarilor. Poate fi citit, prin metoda **Read**(), si pot fi extrase valori ale unor campuri prin metodele **GetString**(int i), de tip **string** si **GetValue**(int i), de tip **object**, care primesc ca parametru indexul (*0-based*) al campului de interes.

Astfel, pentru obiectul de tip SqlCommand definit anterior (*sqlcom*) apelarea metodei **GetString**(1) pe un SqlDataReader creat cu metoda sqlcom.ExecuteReader() va intoarce valoarea de tip **string** "Vanilie".

Metoda **GetString**(int i) asteapta ca in tabel la indexul **i** sa existe un camp de tip text (char, text) a carui valoare sa poate fi preluata direct si prelucrata ca o valoare de tip **string**. Erori vor aparea cand se va incerca citirea din campuri de alt tip (*numeric*, *datetime* etc.). De aceea, este recomandata folosirea metodei **GetValue**(int i) atunci cand nu se cunoaste tipul campului a carei valoare va fi extrasa.

17. CLASA SYSTEM.MATH

Dincolo de operatorii aritmetici de baza, platforma .NET pune la dispozitie clasa System.**Math**, ai carui membri sunt atat metode destinate unor operatii matematice (ridicare la putere, evaluare de valori, operatori trigonometrici s.a.), cat si constante (*PI* si *e*). Membrii specifici ai clasei **Math** sunt:

Abs_X	$Acos_d$	$Asin_d$	$Atan_d$	$Atan2_{d,d}$
$BigMul_{i,i}$	$Ceiling_d$	Cos_d	$Cosh_d$	$DivRem_{L,L}$
E	Exp_d	$Floor_d$	$IEEERemainder_{d,d}$	
$Log_{d,d}$	$Log10_d$	$Max_{X,X}$	$Min_{X,X}$	PI
$Pow_{d,d}$	$Round_X$	$Sign_X$	Sin_d	$Sinh_d$
$Sqrt_d$	Tan_d	$Tanh_d$		

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Indicii metodelor reprezinta tipul comun al valorilor care trebuie furnizate acestora. De exemplu, metoda *Abs* accepta argumente de tip *decimal*, dar si *double* si *float*. Pentru aceasta metoda supraincarcata, indicele X (**aici**, conventie pentru tipul Decimal) reprezinta tipul argumentului asteptat de prima versiune a acesteia. Tot in acest paragraf, am desemnat prin **i** argumente de tip *Integer*, prin **L** argumente de tip *Long* si prin **d** argumente de tip *Double*.

Rezultatul operatiilor matematice efectuate cu operatorii si cu metodele clasei **Math** este, bineinteles, determinat de aceleasi reguli cu cele din matematica obisnuita. Ordinea operatiilor matematice este aceeasi cu cea din "lumea reala" si este recomandata gruparea operatiilor in paranteze pentru claritatea codului. De exemplu,

int i= 2* g/h intoarce acelasi rezultat cu **int** i= (2*g)/h

Functionalitatea expusa de metodele clasei **Math** este descrisa succint in continuare.

Abs Intoarce valoarea absoluta a unei valori de tip *decimal*
Acos Intoarce unghiul al carui cosinus este valoarea specificata de tip *double*
Asin Intoarce unghiul al carui sinus este valoarea specificata de tip *double*
Atan Intoarce unghiul a carei tangenta este valoarea specificata de tip *double*
Atan2 Intoarce unghiul a carei tangenta este obtinuta din raportul (impartirea) celor doi parametri, ambii de tip *double*
BigMul Intoarce produsul a doua valori de tip intreg pe 32 biti
Ceiling Intoarce cel mai mic numar intreg mai mare sau egal cu valoarea specificata de tip *double*
Cos Intoarce cosinusul unghiului specificat ca valoare de tip *double*
Cosh Intoarce cosinusul hiperbolic al unghiului specificat ca valoare de tip *double*
DivRem Intoarce rezultatul impartirii a doua valori de tip *double* si intoarce restul impartirii ca parametru *out*
Exp Intoarce rezultatul ridicarii la o putere de tip *double a* numarului *e* (~ 2.718282)
Floor Intoarce cel mai mare numar intreg mai mic sau egal cu valoarea specificata, de tip *double*
IEEERemainder Intoarce restul impartirii a doua valori de tip *double*
Log Intoarce logaritmul unui numar in baza specificata, ambii parametri fiind de tip *double*
Log10 Intoarce logaritmul in baza 10 a unui numar de tip *double*
Max Intoarce cel mai mare numar din doua valori
Min Intoarce cel mai mic numar din doua valori
Pow Intoarce rezultatul ridicarii la putere a unui numar, ambii parametri fiind de tip *double*

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Round Intoarce numarul avand precizia specificata, care se afla cel mai aproape de numarul furnizat ca primul parametru
Sign Intoarce semnul valorii specificate: 1 pentru numere pozitive si -1 pentru numere negative
Sin Intoarce sinusul unghiului specificat ca valoare de tip *double*
Sinh Intoarce sinusul hiperbolic al unghiului specificat ca valoare de tip *double*
Sqrt Intoarce radacina patrata a valorii specificate ca valoare de tip *double*
Tan Intoarce tangenta unghiului specificat ca valoare de tip *double*
Tanh Intoarce tangenta hiperbolica al unghiului specificat ca valoare de tip *double*

18. REFERINTE (References)

La un proiect se pot adauga **referinte** (References), pentru a pune la dispozitie functionalitate suplimentara, peste cea oferita implicit la crearea unui proiect.

In mediul de programare Visual Studio / Visual C#, spatii de nume implicite sunt:

System
System.Drawing
System.Collections
System.ComponentModel
System.Windows.Forms
System.Data

Cand aceste spatii de nume sunt precedate, in cod, de cuvantul-cheie **using**, trebuie sa fie primele linii/instructiuni ale programului, situate inainte de (si in exteriorul) oricarui spatiu de nume.

Pentru accesarea functionalitatii concrete (de exemplu, Dataset) furnizata, de exemplu, de spatiul de nume **System.**Data, este necesara adaugarea unei **referinte** spre acest spatiu de nume.

Aceasta se realizeaza prin pasii urmatori (in C# EE 2005-2010 si SharpDevelop secventa este similara celei prezentate in continuare):

1. Meniu: View / Solution Explorer, sau Ctrl+Alt+L
2. Click-dreapta pe ramura References
3. Meniu: Add Reference…
4. Pentru spatii de nume preexistente, alegeti pagina **.NET** si executati dublu-click pe spatiul de nume dorit.

De exemplu, daca doriti sa folosim spatiul de nume System.Drawing, alegeti System.Drawing.dll. Observati ca mare parte din fisierele care pot fi referite sunt de tip biblioteca (dll).

Daca doriti sa folositi o biblioteca de clase (Class Library) creata de dumneavoastra, sau una furnizata de o terta parte, selectati pagina **Projects**. Butonul **Browse...** deschide pagina de selectie a unei componente (biblioteci dll) situate intr-o locatie alta decat cea in care se afla bibliotecile platformei .NET.

Pagina **COM** ofera posibilitatea de a referi componente de tip ActiveX (fisiere .ocx si .dll) create in alte limbaje de programare, precum C++ si Visual Basic.

5. Dupa ce asigurati (OK) referinta la componentele necesare, alese din paginile precedente, puteti adauga si liniile **using** specifice acestora.

Daca ati ales componente de tip **Projects**, nu puteti specifica referinta spre ele cu instructiunea using. Se presupune ca acestea ofera clase care pot fi initializate direct in cod.

De exemplu, sa presupunem ca ati creat o biblioteca de clase al carui spatiu de nume este BibliotecaMea, si ca in interiorul acestuia se gasesc clasele Clasa1 si Clasa2, fiecare cu cateva metode.

Veti putea apela metodele din aceste clase dupa ce specificati instructiunea clasica de initializare a unei clase, de exemplu Metoda1 din Clasa1:

```
BibliotecaMea.Clasa1 ObiectClasa1=
        new BibliotecaMea.Clasa1([parametri de initializare]);
ObiectClasa1.Metoda1();
```

Referintele inutile pot fi excluse de la compilarea proiectului - ***a)*** se apeleaza Solution Explorer, ***b)*** click-dreapta pe numele referintei care nu este necesara si apoi ***c)*** click pe **Remove**. Astfel, un pachet de tip Setup care instaleaza o aplicatie care numai calculeaza o suma de doua numere intregi specificate direct de catre utilizator nu are nevoie de spatiul de nume System.Data, astfel ca aceasta referinta poate fi exclusa, bibliotecile care alcatuiesc acest spatiu de nume nemaifiind astfel incorporate in pachetul de instalare.

19. INTERFATA MEDIULUI DE PROGRAMARE MICROSOFT VISUAL STUDIO .NET EXPRESS EDITION

Interfata acestui mediu de programare este similara intre versiunile Express Edition de Visual C#, de la EE 2005 pana la EE 2010. Chiar daca intrarile de meniu poarta nume usor diferite si pot fi mai numeroase in C# 2010, functionalitatea de baza este identica.

19.1. Comenzi apelabile din meniuri

Meniurile pot fi modificate, prin click-dreapta pe orice bara de unelte, click pe **Customize...** si selectarea comenzilor din pagina **Commands**.

In continuare este prezentata functionalitatea oferita implicit de meniul principal al aplicatiei.

19.1.1. **File**

Creare, deschidere si inchidere de proiecte (solutii) si fisiere; adaugare de fisiere la proiectul deschis; adaugare de proiecte la proiectul deschis.

19.1.2. **Edit**

Copiere, decupare si alipire (copy/cut/paste); cautare si inlocuire de cuvinte-cheie in proiect; operatiuni cu textul selectat; manipularea sectiunilor #region.

19.1.3. **View**

Vizualizarea interfetei/formularului curent (Shift+F7) si a codului (F7); vizualizarea ferestrelor Class View, Resource View, Server Explorer (manipularea conexiunilor la baze de date), Properties Window, Object Browser, Toolbox (care contine controalele ce pot fi dispuse pe interfata), Web Browser, Command Window, Output (unde se afiseaza progresia compilarii programului si alte informatii); permite selectarea barelor de unelte s.a.

19.1.4. **Project**

Permite: adaugarea la proiect de referinte, clase, formulare, componente, fisiere noi sau existente; specificarea proiectului de start (pentru proiecte complexe); excluderea din proiect a fisierului selectat; specificarea proprietatilor proiectului s.a.

19.1.5. **Build**

De aici se creeaza executabilul, prin compilarea codului din fisierele care compun proiectul: Build/Rebuild Solution, Build/Rebuild [nume proiect], Configuration Manager s.a.

19.1.6. **Debug**

Permite pornirea (**F5**) si intreruperea (**Shift+F5**) programului; proiectul este compilat inaintea lansarii in executie. Permite specificarea de breakpoint-uri conditionale.

19.1.7. **Tools**

Permite: specificarea proprietatilor mediului de programare (set de caractere, culori, ferestre afisate, formatare automata a liniilor de cod si multe altele); conectarea la o baza de date; depanarea unui proces; adaugarea si indepartarea unor obiecte/controale din Toolbox; managementul Add-In-urilor (instrumente furnizate de terti pentru imbogatirea functionalitatii mediului de programare); crearea de pagini web din comentariile la cod.

19.1.8. **Window**

Permite selectarea unei ferestre deschise deja in mediul de programare (formular, cod etc.)

19.1.9. **Help**

Ofera asistenta, prin modurile: cautare (Search), afisarea subiectelor (Contents); permite verificarea existentei unei versiuni mai noi a mediului de programare (Check for updates) s.a.

19.2. Ferestrele

In mod curent, programatorului ii sunt necesare si utile permanent patru ferestre: cea cu care creeaza interfata programului (**Toolbox**, din care alege controale), fereastra proprietatilor interfetei si ale componentelor (**Properties**, apelabila cu F4, fara functionalitate cand se editeaza codul), fereastra **Solution Explorer**, in care inspecteaza componentele proiectului, si cea in care urmareste procesul de compilare (**Output**), dupa ce acesta este lansat cu comanda Build/Rebuild Solution sau dupa ce comanda rularea programului (F5).

Erorile sunt afisate tabelar in fereastra **Task List**, iar dublu-click pe o linie (eroare) din tabel duce cursorul pe linia vinovata.

Rezultatele cautarii (Search) din meniul Help sunt afisate in fereastra **Search Results**.

Componentele solutiei (proiectului) sunt afisate in fereastra **Solution Explorer** (Ctrl+Alt+L). Aici pot fi specificate si sterse referinte, poate fi inspectata lista fisierelor care compun solutia si pot fi indepartate fisiere.

Server Explorer (Ctrl+Alt+S) expune conexiunile existente la baze de date (indiferent de tipul sursei - SQL Server, Microsoft Access etc.), permite crearea unora noi si afiseaza lista serverelor SQL disponibile in sistem.

Object Browser (Ctrl+Alt+J) expune spatiile de nume folosite de proiect si clasele acestora.

Class View (Ctrl+Shift+C) afiseaza clasele proiectului si membrii acestora.

Resource View (Ctrl+Shift+E) afiseaza resursele utilizate in proiect, continute in fisiere .rc.

Fereastra Properties ofera, de asemenea, posibilitatea de a desemna metode care sa trateze evenimente pentru controlul selectat.

Apasand pictograma **Events**, marcata cu un fulger, veti obtine lista de evenimente la care raspunde controlul selectat. Prin dublu-click pe numele unui eveniment creati metoda corespunzatoare evenimentului. Pot fi selectate metode apartinand altui control, dar care corespund evenimentului ales; de exemplu, daca mai multe controale TextBox aflate pe o forma dispun de metode care trateaza evenimentul *Click*, pentru controlul, sa zicem, TextBox1 poate fi desemnata metoda corespunzatoare evenimentului Click care apartine controlului TextBox2.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Metoda desemnata pentru un eveniment poate fi si stearsa tot in aceasta fereastra, dar nu si din cod, simultan, adica simplu prin stergerea numelui metodei din dreptul evenimentului selectat.

20. Reguli fara de care nu se poate crea un program

1. Numele spatiilor de nume si ale claselor utilizate trebuie sa respecte conventiile de denumire; atentia trebuie orientata spre majuscule: de exemplu, clasa System.Math a platformei Microsoft .NET nu poate fi utilizata prin referirea in cod la o anumita System.math de care .NET nu are cunostinta.

2. Fiecare instructiune este urmata de punct si virgula (;).

3. Un program executabil se poate obtine numai compiland codul din fisierele care il compun.

Pot fi compilate versiuni **Debug** si **Release**.

Diferenta esentiala dintre acestea este ca executia din mediul de programare (tastand **F5**) a unui program compilat in versiunea **Debug** poate fi intrerupta prin puncte de intrerupere (**Breakpoint**). Astfel, liniile marcate cu breakpoint-uri (tastand **F9**) pot fi inspectate pentru eventuale erori ce apar in timpul rularii (*run-time*). Versiunile **Release** nu ofera aceasta facilitate.

Breakpoint-urile sunt ignorate la rularea in afara mediului de programare.

4. Trebuie urmarita executia liniara a codului dintr-un bloc de instructiuni.

5. Trebuie urmarite structurile de cod cuprinse intre acolade.

Alte axiome ale programarii in C# sunt prezentate in Anexa 7.

Daca vreti sa nu ramaneti un simplu utilizator de programe din alte cateva sute de milioane, si sa intelegeti cum au creat cei de la compania X programul Y care ruleaza exact in felul in care ruleaza, trebuie sa va materializati ideile in programe personale. Exercitiul zilnic va fi cel mai bun profesor, ajutandu-va sa intelegeti ca puteti crea multe lucruri remarcabile si ca aveti nevoie numai de idei.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

21. UNELTE ASOCIATE PLATFORMEI .NET

1. **Class Viewer**

Numele executabilului: WinCV.exe, furnizat de regula cu Visual Studio si cu Visual Studio Software Development Kit (SDK).

Functionalitate: Expune spatiile de nume si clasele platformei .NET. Se pot efectua cautari dupa cuvinte-cheie, de exemplu **File**, si vor fi prezentate clasele care contin acest cuvant, impreuna cu membrii lor.

Utilitate: Neaparat necesar celor care nu stiu cum sa apeleze o anumita functionalitate a platformei .NET.

2. **Microsoft Developer Network (MSDN)**

Functionalitate: Este referinta esentiala (sistemul Help) a oricarui programator, indiferent de nivel, pentru orice limbaj de programare sustinut de Microsoft. Ofera: explicatii exhaustive si exemple legate de toate spatiile de nume din platforma .NET, despre clasele si membrii acestora; articole scrise de profesionisti despre versiunile Windows 3.1 - XP si despre utilizarea pachetului Visual Studio; numeroase exemple de cod.

22. MEDII DE PROGRAMARE IN VISUAL C#

1. **Microsoft Visual Studio .NET/Visual C#**, versiunile 2003 si ulterioare (2005, 2008, 2010, 2012 etc.).

2. **Microsoft Visual C# Express/Community Edition**. Ofera functionalitate limitata (depinzand de versiune, pot fi create numai unele proiecte din intreg ansamblul de proiecte care pot fi create – de exemplu, numai aplicatii *desktop* si biblioteci de clase, nu si aplicatii pentru Internet-*web applications*). Inregistrarea ca utilizator se poate face cu date de autentificare cunoscute in *Microsoft Passport* sau de alt fel.

3. **SharpDevelop**, mediu de programare pus la dispozitie gratuit pe site-ul www.icsharpcode.com

Pentru administrarea bazelor de date SQL Server, Microsoft ofera gratuit pachetul **SQL Server Express Edition with Advanced Services (2005, 2008, 2012 etc.)**, impreuna cu fisiere de asistenta (sau Help - *BooksOnline*) si cu un program-interfata grafica, optional, pentru utilizarea facila a produsului. Ofera functionalitate limitata, dar nu ingreuneaza sau impiedica operatiunile fundamentale pe baze de date (administrarea datelor, crearea de rezerve si restaurarea bazelor de date).

Daca aveti nevoie de ajutor in vreo problema legata de programare in C#, puteti contacta autorul acestui curs:

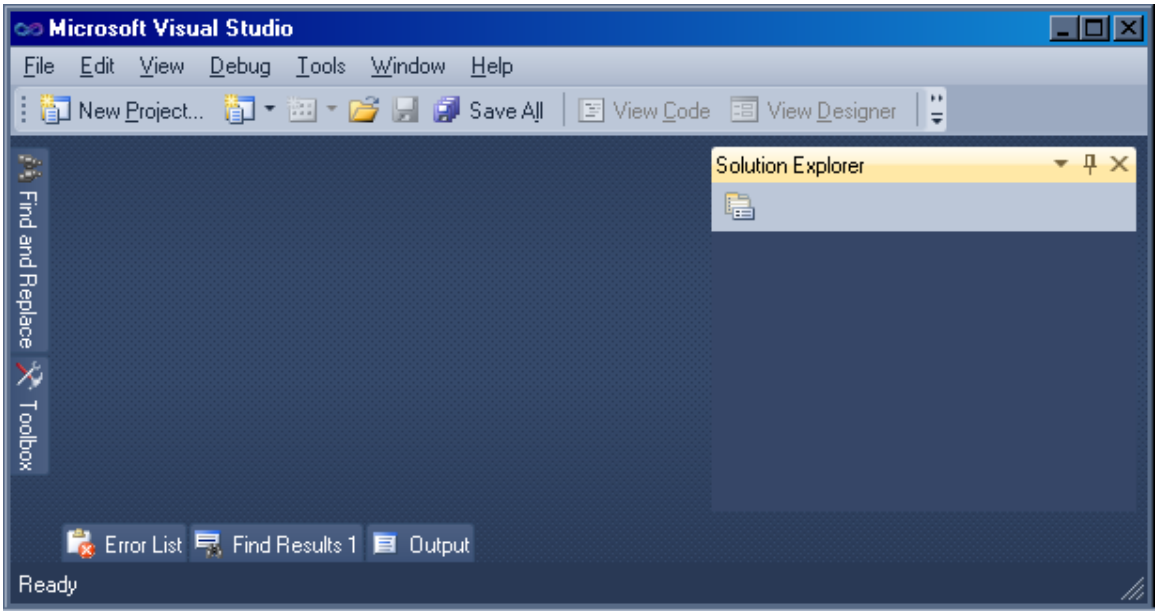
contact@sinapsaro.ro

S. C. SINAPSA Infocomplex S. R. L.
http://sinapsaro.ro

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

23. NAVIGAREA IN MEDIUL DE PROGRAMARE MICROSOFT VISUAL C# EXPRESS

1. Interfata



Meniul principal (modificabil): File, Edit, Debug, Tools, Window, Help.

Solution Explorer: prezinta componenta programului – numele "solutiei" si numele proiectelor (programe, DLL-uri s.a.) componente. Fereastra aceasta poate fi adusa pe ecran cu combinatia de taste (Ctrl-Alt-L).

Puteti modifica numele solutiei, proiectelor din solutie si componentelor acestora cu click-dreapta pe denumirile acestora, afisate in arborele solutiei, si selectand "Rename". Este bine insa sa inlocuiti numele peste tot in solutie (v. mai jos). *Aveti grija sa nu stergeti **extensia** fisierelor care alcatuiesc solutia*.

Properties: prezinta proprietatile ferestrei selectate, sau ale controlului selectat intr-o fereastra a programului - daca este afisata.

Pentru a afisa proprietatile unei ferestre de program, click pe aceasta (nu pe un control de pe ea) si apasati **F4**.

Pentru a afisa proprietatile unui control plasat pe o fereastra, click pe control (nu pe fereastra) si apasati **F4**.

In **Properties** puteti schimba cam toate proprietatile ferestrei si ale controlului selectat, de la nume la opacitate.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

O proprietate se schimba cu click-stanga pe numele sau pe valoarea sa si scriind noua valoare. Culorile si alte proprietati care nu sunt de tip text se modifica in principiu cu click-stanga pe butonul cu trei puncte – sau pe butonul cu sageata – care apar pe liniile corespunzatoare acelor proprietati in tabel.

Daca o proprietate are mai multe valori componente (de exemplu, *Location*), click-stanga pe semnul + din dreptul proprietatii aceleia va desfasura componentele sale, pe care le puteti edita separat.

Dupa orice modificare a valorii unei proprietati, este bine sa apasati Enter. Unele valori pot fi modificate numai scriind noua valoare si apoi "iesind" de pe linia lor (de exemplu, click-stanga pe alt control).

2. Navigarea in codul programului

2.A. Puteti micsora spatiul ocupat de instructiunile programului prin incorporarea unor blocuri intregi in sectiuni initiate cu directiva #**region** si incheiate cu #**endregion**. Aceste doua directive trebuie, insa, sa preceada imediat instructiuni **aflate pe acelasi nivel**. De exemplu, puteti scrie:

```
#region bloc if
if (a == b)
{
	a= 4;
	//urmeaza inca 100 de instructiuni, pe 2 pagini
}
#endregion  //bloc if
```

Aici, am denumit regiunea "bloc if" *ca sa stim ce se intampla* in interiorul sau.

Am plasat un comentariu dupa #endregion, *ca sa stim ce s-a intamplat* in acest bloc (#region-#endregion).

Putem sa nu numim nicidecum regiunile, nici sa nu plasam comentarii la sfarsitul lor, insa e bine sa stim ce se intampla in zonele colapsate de cod fara sa le deschidem si sa le citim de la cap la coada.

Semnul # este obligatoriu inaintea lui "region" si "endregion".

La click pe semnul + care apare la stanga lui #**region**, tot blocul pana la directiva #**endregion** va fi "colapsat" intr-o singura linie care va arata astfel:

```
bloc if
```

Fiecare astfel de bloc **#region-#endregion** poate fi extins si colapsat oricand cu click pe semnul +. Puteti edita (modifica) oricand continutul acestor blocuri.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Asa ceva nu se poate face:…

```
#region Bloc if
if (a == b)
{
	a= 4;
	//urmeaza inca 100 de instructiuni, pe 2 pagini
#endregion   //Bloc if
}
```

… fiindca #endregion nu este pe acelasi nivel cu #region, ci *in interiorul* blocului if (cu alte cuvinte, o acolada mai jos).

2.B. Inlocuirea unui simbol **in tot codul solutiei, eventual alcatuite din mai multe proiecte**

Deschideti codul (F7 sau click-dreapta pe componenta dorita in Solution Explorer si selectati din meniu "View Code");

Apasati combinatia de taste:
Ctrl+Shift+H

Din lista "Look in:" selectati:
Entire Solution

In fereastra care apare specificati particula de text de inlocuit si inlocuitorul sau.
Aveti grija sa bifati optiunile (daca este cazul) din sectiunea "Find options": *Match case* (inlocuire **numai** la identitate intre particule la nivel de caracter – majuscule si litere mici); *Match whole word* (inlocuire daca este cuvint separat. Daca este bifata, de exemplu nu se va inlocui "button1" cu "butonul" daca se va gasi "button1_click", ci numai daca se va gasi "button1", separat)
Caseta "Use" poate folosi pentru inlocuire:

- *Regular expressions* – un subiect foarte complex care nu este tratat aici si care oricum necesita documentare dureroasa (pentru oricine) inainte de a fi abordat…
- *Wildcards* – * reprezinta mai multe caractere, ? reprezinta un singur caracter. Secvente intregi pot fi inlocuite, de exemplu "button1*" de inlocuit cu "butonul inseamna ca vor fi inlocuite **toate** secventele care incep cu "button1".

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

2.C. Inlocuirea unui simbol **in fereastra de cod curenta (deschisa)**

Se procedeaza la fel ca la 2.B, dar din lista "Look in:" selectati:
Current Document

Similar cu 2.B. si 2.C. se pot inlocui particule in proiectul curent (*Current Project*), toate documentele deschise (*All Open Documents*) si **numai in metoda curenta** – adica unde se afla cursorul (*Current Block*)

2.D. Compilarea programului/bibliotecii de clase

Meniu: **Build**
Comanda: **Rebuild Solution** si **Build Solution**
Rebuild Solution este recomandat, deoarece compileaza toate componentele solutiei (adica toate proiectele cu toate componentele lor).

3. *Optiuni*

Meniu: **Tools/Options...**

Aici gasiti optiuni pentru diverse proceduri efectuate cu mediul de programare: pentru aspectul blocurilor de instructiuni, pentru sugestiile oferite la scrierea instructiunilor, puteti schimba culorile ferestrelor unde editati cod si vedeti erorile s.a.

Meniu: **Tools/Customize...**

Aici puteti schimba bara de pictograme (icon-uri), tragand inainte si inapoi in fereastra cu pictograme unele de pe bara deja existenta sub cea cu meniul principal al mediului de programare. Totodata, dupa click-stanga pe o pictograma deja existenta pe bara, ii puteti schimba aspectul – text si imagine, numai imagine etc.

4. *Proprietatile proiectului curent*

Meniu: **Project/(numele proiectului) properties...**

Pentru a accesa proprietatile unui proiect (creator, versiune, versiunea platformei .NET necesare pentru a-l rula s.a.), asigurati-va intai ca in fereastra Solution Explorer (Ctrl-Alt-L) selectati (click-stanga) numele unui proiect, sau al unei componente a proiectului pe care doriti sa-l editati.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

5. *AutoComplete*

Meniu: **Tools/Options...**
Bifati: **Show all settings**
Ramura: **Text Editor/All Languages**
Bifati: **Statement completion/Auto list members**
De-bifati: **Hide advanced members**
Bifati: **Parameter information**

6. *Auto-aliniere in blocurile de instructiuni*

Ramura: **Text Editor/All Languages/Tabs**
Bifati: **Indenting/Smart**
Bifati: **Tab/Keep tabs**

7. *"F"-uri esentiale*

F5: Ruleaza programul
F4: afiseaza fereastra **Properties**
F7: afiseaza codul corespunzator ferestrei vizualizate
Shift-F7: afiseaza fereastra corespunzatoare codului vizualizat
F2: dupa click pe numele unei componente a solutiei in fereastra **Solution Explorer**, sau pe linia unei proprietati in fereastra **Properties**, se poate schimba numele acelei componente, sau valoarea proprietatii selectate.

Nu e recomandat, insa, sa schimbati numele fisierelor in Solution Explorer: daca totusi insistati, intai folositi Ctrl+Shift+H pentru inlocuirea acelui nume in toata solutia (de exemplu, inlocuiti in tot **codul** programului "Form1" cu "fereastramea"), abia **apoi** modificati in Solution Explorer numele fisierului "Form1.cs" si al celorlaltor fisiere denumite la fel, dar cu alte extensii.

8. *Solution Explorer*

Dublu-click cu butonul stang:

- pe fisiere .cs care contin codul propriu-zis, nu cele marcate cu pictograma de 'fereastra": afiseaza codul
- pe fisiere .cs marcate cu pictograma de fereastra: afiseaza fereastra (interfata)

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Click-dreapta:

- pe orice fisier listat: afiseaza meniul operatiunilor care pot fi efectuate cu acel obiect. Puteti exclude de la compilare (din program) un fisier selectand optiunea "*Exclude From Project*" – fisierul nu va fi sters fizic, ci numai sters din lista si poate fi adaugat iarasi la proiect
- pe fisiere .cs marcate cu pictograma de fereastra: permite afisarea codului si a ferestrei asociate: selectati *View Code*, respectiv *View Designer*
- pe "Solution 'nume solutie'": permite (re)compilarea (*Rebuild*, *Build*) tuturor proiectelor si alcatuirea executabilului (executabilelor, daca in solutie sunt mai multe proiecte)
- pe numele unui proiect: permite atat (re)compilarea (*Rebuild*, *Build*), cat si selectarea acelui proiect (program) ca fiind cel cu care incepe rularea programului (daca in solutie sunt mai multe proiecte) – "*Set as StartUp Project*"
- pe orice fisier si pe numele solutiei si proiectelor, apoi selectarea intrarii de meniu *Properties*: in fereastra de proprietati afiseaza calea completa catre fisier, solutie sau proiect si proiectul de start in cazul solutiilor

9. *Directivele #region - #endregion*

Obisnuiti-va sa folositi aceste directive pentru a structura mai bine codul programelor din punct de vedere vizual si, in final, al accesibilitatii.

Considerati cazul in care aveti o metoda de 100 de linii, care contine un ciclu *for* de cateva linii, dar care este esential pentru acea metoda si program. Ca sa ajungeti la el, trebuie fie sa cautati cu *Find* cuvantul *for* in metoda respectiva, fie sa cititi toate liniile pana ajungeti unde trebuie. Mai simplu este sa-l incadrati cu tot cu instructiunile din el intr-o *#region*, astfel:

```
private void metoda(argumente)
{
	[…]	//aici erau vreo 40 de linii, dupa care urmeaza ciclul for:
	for (blabla)
	{
		[…]	//aici sunt instructiunile din ciclul for
	}
	[…]	//alte cateva zeci de linii
}
```

^ asta se schimba in:

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

```
private void metoda(argumente)
{
        […]    //aici erau vreo 40 de linii, dupa care urmeaza ciclul for:
        //
+       #region Ciclul for de interes
        for (blabla)
        {
                […]    //aici sunt instructiunile din ciclul for
        }
        #endregion //Ciclul for de interes
        //
        […]    //alte cateva zeci de linii
}
```

Observati ca nivelul pe care se afla directivele #region si #endregion este acelasi. Incercati sa le scrieti pe niveluri diferite - de exemplu, inchideti cu #endregion inainte de acolada de inchidere a ciclului *for* – mediul de programare va va avertiza ca nu este scrisa unde trebuie, ori ca directiva *#region* asociata lipseste, si va avertizeaza la fel pentru linia *#region*.

La click pe simbolul (generic) +, intreaga regiune de cod cuprinsa intre linia #region si cea #endregion va fi restrans la o singura linie intitulata astfel:

+ Ciclul for de interes

La click pe simbolul +, sectiunea de cod va fi desfasurata si va putea fi editata iarasi.

Daca nu dati nici un nume pentru **#region** (nu scrieti nimic imediat dupa aceasta), sectiunea va fi restransa tot la o singura linie, dar aceasta va contine numai textul "#region". Este bine insa sa intitulati cumva regiunile definite, pentru a sti ce se afla inauntrul lor.

Observati comentariul de pe linia *#endregion*. Nu este necesar, dar este util atunci cand vrem sa stim ce se intampla in secventa de cod inchisa de aceasta directiva.

©SINAPSA Infocomplex®™/Victor Emanuel Stefanescu, 2017.12 – 2023.04

Pe linia #endregion poate sa se afle numai comentariu.
Daca puneti comentariu dupa #region, acesta va fi considerat numele regiunii, cu tot cu slash-urile care marcheaza inceputul comentariului.

Asa ca, in general, nu conteaza ce nume dati sectiunilor – conteaza numai sa aflati usor dupa aceea ce se afla inauntru, acesta fiind de altfel scopul lor principal, inainte de "scurtarea" listingului.

Nota finala.

Anexele contin prezentari detaliate ale unor aspecte asupra carora nu s-a insistat in detaliu in cadrul acestei prezentari. Cele 10 anexe sunt:

Anexa 1: Operatori - exemple de utilizare
Anexa 2: Variabile - initializare si folosire
Anexa 3: Tratarea erorilor
Anexa 4: Spatiile de nume ale platformei Microsoft .NET
Anexa 5: Utilizarea simpla a unor controale din spatiul de nume Windows.Forms
Anexa 6: Cuvintele-cheie ale limbajului C# si caracterele speciale
Anexa 7: Axiome
Anexa 8: Crearea unor proiecte de diverse tipuri in mediul de programare Microsoft Visual Studio
Anexa 9: Exemple de cod, notiuni simple si avansate
Anexa 10: Crearea bazelor de date in sistemul Microsoft SQL Server 2005-2012 Express Edition.