Utilizare Visual Studio .Net

Noțiuni generale

Pentru gruparea fişierelor sursă şi a altor resurse utilizate în cadrul aplicației, mediul Visual Studio .Net (VS) utilizează două concepte:

- Proiect: fişier care conține toate informațiile necesare pentru a compila un modul dintr-o aplicație .Net
- Soluție: fişier care conține lista proiectelor care compun o aplicație, precum şi dependințele dintre ele

Proiectele sunt fişiere XML care conțin următoarele informații:

- lista fişierelor necesare, poziția pe disc a acestora precum şi modul în care vor fi utilizate (compilate in cod executabil, incluse în executabil, …)
- lista de module externe referite
- mai multe seturi de parametri de compilare numite configurații (implicit sunt doar două – Debug şi Release – dar se pot defini şi alte configurații)
- diverse opțiuni legate de proiect

Fişierele de tip proiect pentru C# au extensia *csproj*. Principalele tipuri de proiecte sunt:

- Console Application: aplicații de tip linie de comandă, fără interfață grafică; rezultatul este un fişier executabil (.exe)
- Class Library: bibliotecă de clase care poate fi utilizată pentru construirea altor aplicații; rezultatul este o bibliotecă cu legare dinamică (.dll)
- Windows Application: aplicație windows standard; rezultatul este un fişier executabil (.exe)

Proiectele sunt singura modalitate prin care se pot compila aplicații .Net folosind VS.

Soluțiile sunt fişiere text cu extensia *sln* care conțin lista tuturor proiectelor care compun aplicația, dependințele dintre ele şi configurațiile disponibile. Orice proiect este inclus obligatoriu într-o soluție (creată explicit de către utilizator sau creată implicit de către VS).

Crearea unei aplicații de consolă

Crearea unei aplicații de consolă C# se poate face utilizând comanda *File->New* project şi selectând *Visual C# Projects -> Console Application*.

Principalele opțiuni disponibile:

- Location: directorul unde vor fi create fişierele
- Name: numele proiectului
- Add to solution / Close solution (doar în cazul în care există o soluție deschisă): permite adăugarea unui proiect nou în cadrul soluției sau crearea unei soluții noi
- More:
    - Create directory for solution: crează un director separat pentru solutie (implicit solutia va fi creată în acelaşi director cu proiectul)

- New solution name: numele soluției (implicit este acelaşi cu a proiectului)

Presupunem că au fost alese următoarele opțiuni:
- Location: d:\ase\poo\
- Name: PrimaAplicatie
- Close Solution (dacă este cazul)
- More: bifat "Create directory for solution"

VS-ul va crea următoarele:
- un fişier soluție "PrimaAplicatie.sln"
- un fisier proiect "PrimaAplicatie.csproj"
- două fişiere sursă: "AssemblyInfo.cs" (conține proprietățile care pentru executabil) şi "Class1.cs" care conține o clasă care reprezintă aplicația noartră

Structura soluției poate fi vizionată folosind fereastra "Solution Explorer" (View->Solution Explorer).

Aplicația creată de VS (care momentan nu face nimic) poate fi rulată folosind CTRL+F5 (Debug->Start Without Debugging). În cazul în care o soluție conține mai multe proiecte, setarea proiectului care va porni la CTRL+F5 poate fi făcută prin right click pe proiect în "Solution Explorer" şi "Set as StartUp Project".

Proprietățile proiectului pot fi accesate selectând proiectul în Solution Explorer + click dreapta Properties (sau Project → [nume proiect] Properties din meniu).

Anatomia unui program C#

Programele C# pot fi constituite din mai multe fişiere sursă cu extensia *cs*. Fiecare fişier poate conține mai multe domenii de nume (namespaces). Acestea la rândul lor pot conține declarații de tipuri (clase, structuri, interfețe, delegați sau enumerații) sau alte domenii de nume. Pot exista declarații de tipuri şi în afara domeniilor de nume, dar această abordare nu este recomandată (mai multe detalii în secțiunea următoare).

Spre deosebire de C++, toate elementele care constituie aplicația sunt definite în interiorul claselor (nu există variabile globale sau funcții independente). Punctul de intrare în program este metoda statică *Main*. În cazul în care există mai multe metode statice Main în clase diferite, metoda de start trebuie precizată la compilare folosind proprietățile proiectului.

Pentru exemplificare putem folosi codul generat de VS:

```
using System;

namespace PrimaAplicatie
{
	/// <summary>
	/// Summary description for Class1.
	/// </summary>
	class Class1
	{
		/// <summary>
		/// The main entry point for the application.
		/// </summary>
		[STAThread]
```

```
            static void Main(string[] args)
            {
                //
                // TODO: Add code to start application here
                //
            }
        }
}
```

Aplicația este constituită dintr-o singură clasă (*Class1*) care conține metoda statică *Main* (punctul de start). Clasa este inclusă în domeniul de nume *PrimaAplicatie*.

Observații:

- Comentariile folosesc aceeaşi sintaxă ca şi în C++ (//… şi /*... */).
- *[STAThread]* – este un atribut aplicat metodei *Main*
- *using System* – permite utilizarea tipurilor de bază fără o calificare suplimentară

Domenii de nume

Domeniile de nume sunt entități sintactice care permit gruparea logică a denumirilor de tipuri. Folosirea domeniilor de nume permite evitarea coliziunilor generate de utilizarea aceloraşi identificatori în biblioteci diferite.

Declararea unui domeniu de nume se face folosind cuvântul cheie *namespace*:

```
namespace nume_domeniu
{
        // declaratii
}
```

În cadrul namespaceului tipurile sunt utilizate normal, iar în afara acestuia sunt utilizate folosind forma *nume_domeniu.nume_tip*. Se pot declara namespaceuri imbricate pentru a construi o structură ierarhică de nume. În cazul în care există mai multe declarații de domenii cu acelaşi nume, ele sunt concatenate de către compilator.

Exemplu de utilizare:

```
// declaratie namespace
namespace StructuriDeDate
{
        // declaratii clase in cadrul namespace-ului
        class Vector
        {
                //...
        }

        class Matrice
        {
                //...
                // aici putem utiliza alte clase din namespace
                // fara a fi necesare calificari suplimentare:
                // Vector v;
        }

        // declaratie namespace imbricat
        namespace StructuriDinamice
        {
                // declaratii clase
                class ListaSimpla
                {
                        //...
                }
                class ListaDubla
```

```
                {
                        //...
                }
        }
}

// declaratie namespace (declaratiile de aici
// vor fi adaugate in namespace-ul StructuriDeDate
// declarat anterior)
namespace StructuriDeDate
{
        // declaratie clasa
        class MatriceRara
        {
                //...
        }
}

// namespaceul aplicatiei
namespace Aplicatie
{
        class AplicatiaCuStructuri
        {
                public static void Main()
                {
                        // aici trebuie sa utilizam denumirea completa:
                        StructuriDeDate.Matrice matrice;
                        StructuriDeDate.StructuriDinamice.ListaDubla lista;
                }
        }
}
```

Pentru a evita folosirea numelor complete se poate folosi directiva *using* (cu sintaxa *using nume_namespace;*). Aceasta permite folosirea tipurilor declarate în alte namespaceuri fără a fi nevoie să folosim numele complet.

Directiva poate fi inserată înaintea oricărui namepace (are efect în fişierul curent) sau la începutul unui namespace, caz în care are efect doar în cadrul namespace-ului respectiv (doar în porțiunea din fişierul curent).

Chiar şi în cazul folosirii acestei directive, utilizarea numelor complete este obligatorie atunci când există ambiguități.

Exemplu:

```
using StructuriDeDate;

// namespaceul aplicatiei
namespace Aplicatie
{
        class AplicatiaCuStructuri
        {
                public static void Main()
                {

                        // putem utiliza numele simplu datorita
                        // directivei using StructuriDeDate; si
                        // a faptului ca nu exista conflicte de nume
                        Matrice matrice;

                        // aici trebuie sa utilizam denumirea completa
                        // (pentru a putea utiliza denumirea simpla ar fi trebuit
                        // sa adaugam la inceputul fisierului directiva
                        // using StructuriDeDate.StructuriDinamice;
                        StructuriDeDate.StructuriDinamice.ListaDubla lista;
                }
        }
}
```

Operații de intrare/ieşire pentru consolă

Operațiile de I/E cu consola sunt implementate prin metode statice în cadrul clasei *System.Console*. Cele mai utilizate metode sunt *Write* (scrie un mesaj la consolă), *WriteLine* (scrie un mesaj şi trece la un rând nou) şi *ReadLine* (citeşte un rând de text de la tastatură).

Metodele *Write* şi *WriteLine* primesc aceiaşi parametri şi au acelaşi comportament (singura diferență este că *WriteLine* trece la un rând nou după afişarea mesajului). Aceste metode au două forme:

a) pentru tipurile de bază

Această formă permite afişarea directă a tipurilor simple (int, char, double, …) şi are sintaxa *Console.Write(valoare)*.

Exemple:

```
Console.Write("text");
Console.WriteLine(34);
char c = 'x';
Console.WriteLine(c);
```

b) cu formatare

Permite afişarea cu formatare (similar funcției *printf* din C). Sintaxa utilizată este: *Console.Write(sir_formatare, parametri)*. Şirul de formatare este compus din textul de afişat în care sunt introduse elemente de forma *{i}* în locul unde trebuie inserate valorile parametrilor (*i* – începe de la 0 şi reprezintă poziția parametrului în listă).

Exemple:

```
// declarare si initializare variabile
string nume = "Ionel";
int varsta = 7;

// afisare cu formatare
Console.WriteLine("{0} are {1} ani.", nume, varsta);
// Va afisa:
// Ionel are 7 ani.
```

Citirea datelor se face sub formă de şiruri de caractere folosind sintaxa *var = Console.ReadLine();*, unde *var* este o variabilă de tip *string*. Citirea altor tipuri de date simple se face utilizând metodele statice *Parse* din tipul respectiv.

Exemple:

```
// declarare variabile
string nume;
int varsta;

Console.Write("Nume:");
// citire strings
nume = Console.ReadLine();

Console.Write("Varsta:");
// citire string si conversie la int
varsta = int.Parse(Console.ReadLine());
```

Tipuri simple

În C# toate tipurile de date sunt de fapt clase derivate direct sau indirect din clasa *System.Object*. Limbajul permite utilizarea unor nume alternative pentru tipurile simple de date. Declararea şi iniţializarea variabilelor (pentru tipuri simple) se face la fel ca în C++.

Cele mai utilizate tipuri sunt:

Alias	Nume real	Descriere
object	System.Object	Clasa de bază din care sunt derivate direct sau indirect toate tipurile din .Net.
string	System.String	Şiruri imutabile de caractere Unicode.
byte	System.Byte	Întregi cu semn pe 8 biţi.
short	System.Int16	Întregi cu semn pe 16 biţi.
int	System.Int32	Întregi cu semn pe 32 biţi.
long	System.Int64	Întregi cu semn pe 64 biţi.
char	System.Char	Întregi fără semn pe 16 biţi (corespund setului de caractere Unicode).
float	System.Single	Implementare pe 32 de biţi a formatului în virgulă mobilă IEEE 754.
double	System.Double	Implementare pe 32 de biţi a formatului în virgulă mobilă IEEE 754.
decimal	System.Decimal	Format în virgulă mobilă pe 128 de biţi. Are o plajă de valori mai mică decât *double* dar are o precizie mai bună. Este utilizat în special în calcule financiare.
bool	System.Bool	Reprezintă valorile logice *true* şi *false* utilizând un octet de memorie.

Fiind de fapt clase, toate tipurile de bază conţin şi metode. Aceste metode pot fi aplicate chiar şi în cazul constantelor literale.

Toate tipurile conţin metoda *ToString* (moştenită din *object* şi suprascrisă în clasele derivate) care permite transformarea valorii respective în *string*. În cazul tipurilor numerice, transformarea în *string* se poate face si cu formatare.

Şirurile folosite pentru formatare: şiruri standard + şiruri custom.

Tipurile numerice şi tipul *bool* conţin o metodă statică numită *Parse* care permite transformarea unui şir de caractere în valoarea corespunzătoare.

Exemple de utilizare:

```
// declarare si initializare variabile
int i = 7, j;
long l = 23L;  // constanta de tip long
decimal valoareCont = 3213265465.454654654M;
bool unBoolean;

// conversii din string
unBoolean = bool.Parse("true");
j = int.Parse("236");

// conversii in string (cu 4 zecimale)
```

```
string strValoare = valoareCont.ToString("####.####");

// afisare variabile
Console.WriteLine("Contul are valoarea: " + strValoare);
```

Limbajul poate efectua conversii între tipurile de date numerice: automat în cazul în care tipul destinație este mai puternic decât tipul sursă sau explicit dacă există posibilitatea pierderii de informații (ex convertire din *long* în *int*).

Şirurile de caractere pot fi stocate şi prelucrate utilizând tipul *string*. Acesta este de fapt o colecție imutabilă de caractere Unicode (caracterele în C# sunt reprezentate pe 2 octeți). Orice modificare efectuată asupra unui *string* va genera un nou obiect. Constantele de tip şir de caractere pot fi reprezentate în două moduri:

- normal: textul este pus între ghilimele duble (“”) iar caracterele speciale trebuie prefixate cu „\” ca în C++ (\n – linie nouă, \” – ghilimele, \\ - back slash, …);
- prefixate cu „@”: conținutul este păstrat exact aşa cum apare între ghilimele; doar ghilimelele duble care apar în corpul şirului trebuie dublate.

Prelucrarea variabilelor de tip *string* se poate realiza folosind operatorii predefiniți (== şi != pentru concatenare, = pentru atribuire, [] pentru indexare, + şi += pentru concatenare, …) sau metodele clasei *System.String* (există funcții pentru formatare, copiere, căutare, înlocuire, extragere fragmente, împărțire după un caracter dat, …).

Lista completă a metodelor suportate se poate consulta aici.

Exemple de utilizare:

```
// constante de tip sir de caractere
// varianta normala (cu secvente de escape)
string sir1 = "c:\\temp\\fisier.txt";
// varianta indigo (nu mai sunt necesare secventele de escape)
string sir2 = @"c:\temp\fisier.txt";
// ghilimele in siruri prefixate cu @
string sir3 = @"Numele este ""Ionel""."; // => Numele este "Ionel".
// varianta cu siruri normale
string sir4 = "Numele este \"Ionel\".";  // => Numele este "Ionel".

// comparare siruri (se face comparand continutul)
if (sir3 == sir4)
        Console.WriteLine("Sirurile sunt egale.");

// concatenare siruri
string sir5 = sir4 + "Varsta lui este " + 7.ToString() + " ani.";
Console.WriteLine(sir5);

// utilizarea functiei de formatare
// (sintaxa este similara cu Console.Write)
string nume = "Ionel", oras = "Iasi";
int varsta = 8;
string sir6 = string.Format(
        "Numele este {0} si are {1} ani. {0} este din {2}.",
        nume, varsta, oras);
// sirul va avea valoarea:
// Numele este Ionel si are 8 ani. Ionel este din Iasi.

// utilizarea functiei de cautare
int index = sir6.IndexOf("este din");
if (index >= 0)
        Console.WriteLine("Sirul a fost gasit pe pozitia {0}.", index);
else
        Console.WriteLine("Sirul nu a fost gasit.");
```

Tipuri valorice şi tipuri referențiale

În .NET, tipurile de date se împart în două categorii principale: tipuri valoare si tipuri referință. Diferența dintre ele este că variabilele de tip referință conțin referințe (pointeri) spre datele propriu-zise, care se afla în heap, pe când variabilele de tip valoare conțin valorile efective. Această deosebire se observă, de exemplu, la atribuiri sau la apeluri de funcții. La o atribuire care implică tipuri referință, referința spre un obiect din memorie este duplicată, dar obiectul în sine este unul singur (are loc fenomenul de aliasing – mai multe nume pentru acelaşi obiect). La o atribuire care implică tipuri valoare, conținutul variabilei este duplicat în variabila destinație. Tipurile valoare sunt structurile (*struct*) si enumerările (*enum*). Tipuri referință sunt clasele (*class*), interfețele (*interface*), tablourile si delegările (*delegate*).

Tipurile simple, cu excepția *object* şi *string*, sunt tipuri valorice.

Tipurile valorice sunt alocate pe stivă la momentul declarării, deci nu există variabile cu valoarea *null*. Atribuirea şi trimiterea ca parametru în funcții se face prin copierea conținutului (valorii) variabilei; copierea se face bit cu bit.

Exemplu:

Operația / **Stiva**

```
int i = 42;   // se aloca spatiu si se initializeaza
```

	Stiva	
132		
131		
130		
129	42	i

```
int j;        // se aloca spatiu
```

	Stiva	
132		
131		
130	?	j
129	42	i

```
j = i;        // se copiaza valoarea 42
```

	Stiva	
132		
131		
130	42	j
129	42	i

Tipurile referențiale sunt alocate explicit folosind operatorul *new* şi sunt stocate în heap. Manipularea se face utilizând referințe. Referințele sunt similare cu referințele din C++, cu două diferențe:

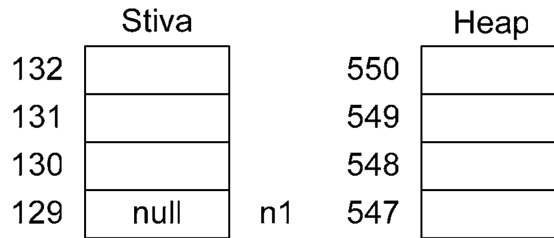
- nu trebuie inițializate la declarare şi pot conține valoarea *null*
- obiectul pointat poate fi modificat la rulare

Exemplu:

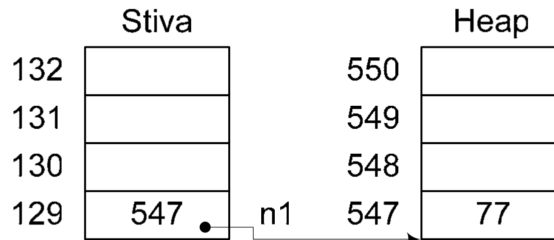
```
// exemplu de clasa
class Numar
{
        // constructor
        public Numar(int valoare)
        {
                Valoare = valoare;
        }
```

```
        // un membru public
        public int Valoare;
}
```

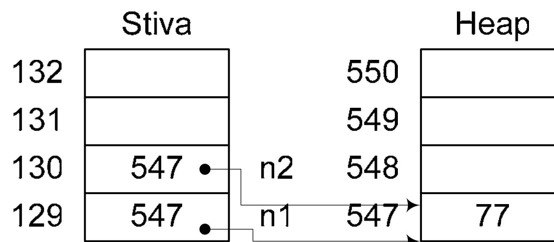
```
Numar n1;           // declarare
                    referinta
```



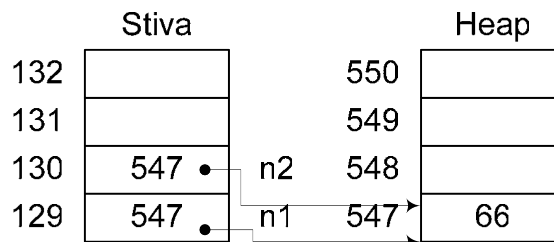

```
n1 = new            // alocare
Numar(77);          obiect
```




```
Numar n2 = n1;      // atribuire
                    referinta
```




```
n2.Valoare =        // modificare
66;                 valoare obiect
                    prin referinta
```

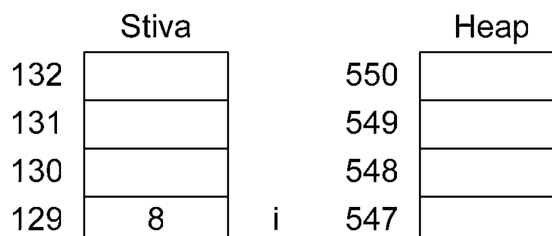


Conversia dintre tipurile valorice şi tipurile referenţiale se poate realiza prin mecanismele de împachetare şi despachetare (*boxing* şi *unboxing*). Aceste mecanisme sunt necesare pentru a permite o tratare unitară a claselor (de exemplu în cadrul colecţiilor).

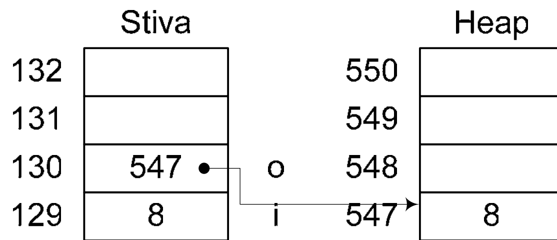
Împachetarea presupune copierea valorii de pe stivă în heap şi alocarea unei referinţe la aceasta pe stivă. Despachetarea presupune alocarea spaţiului pentru valoare pe stivă şi copierea conţinutului de pe heap. În cazul despachetării este obligatorie efectuarea unui cast.

Exemplu:

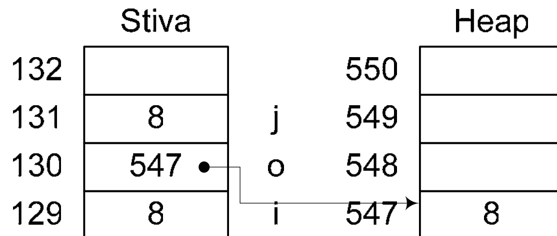
```
int i = 8;
```



```
object o = i;    // boxing
```




```
int j = (int)o; // unboxing
```



Mai multe detalii de aici şi aici.

Masive

Masivele sunt structuri de date omogene şi continue. În C#, masivele sunt tipuri referenţiale derivate din clasa abstractă System.Array (crearea clasei derivate se face automat de către compilator).

Elementele masivelor pot fi de orice tip suportat (tipuri referenţiale, tipuri valorice, alte masive, …) şi sunt accesate prin intermediul indicilor (începând cu 0). Dimensiunea masivelor este stabilită la crearea acestora (la rulare) şi nu poate fi modificată pe parcurs. Limbajul suportă atât masive unidimensionale, cât şi masive multidimensionale.

Fiind clase, masivele au o serie de proprietăţi şi metode, dintre care cele mai importante sunt:

- *Length*: numărul total de elemente din masiv;
- *Rank*: numărul de dimensiuni ale masivului;
- *Clone()*: creează o copie a masivului; **Atenţie**: în cazul tipurilor referenţiale sunt copiate numai referinţele, nu şi obiectele referite;
- *Copy(), CopyTo()*: copiază secţiuni din masiv în alt masiv;

Declararea unui masiv unidimensional se face sub forma *tip[] nume;*. Iniţializarea se poate face la momentul declarării sub forma *tip[] nume = {lista valori};* sau ulterior sub forma *nume = new tip[] {lista valori};*. În cazul în care se doreşte doar alocarea memoriei se poate folosi *nume = new tip[dimensiune]*; în acest caz se va aloca memorie, iar elementele vor fi iniţializate cu valorile implicite (*null* pentru tipuri referenţiale, 0 pentru tipurile numerice, …).

Exemple:

```
// un vector de intregi
int[] vector1;

// initializare vector
vector1 = new int[] {5, 23, 66};

// declarare si initializare
double[] vector2 = {34.23, 23.2};
```

```
// accesarea elementelor
double d = vector2[0];
vector2[1] = 5.55;

// alocare memorie fara initializarea elementelor
string[] vector3 = new string[3];

// afisarea elementelor
for (int i = 0; i < vector1.Length; i++)
        Console.WriteLine("vector1[{0}]={1}", i, vector1[i]);

// copierea elementelor
int[] vector4 = new int[vector1.Length];
vector1.CopyTo(vector4, 0); // 0 = pozitia de start
```

Masivele multidimensionale se utilizează la fel ca şi masivele unidimensionale. Accesarea elementelor se face sub forma [dim_1, dim_2, …, dim_n]:

```
// declarare si alocare masiv tridimensional
int[,,] cub = new int[5,2,7];

// accesare elemente
cub[0,0,0] = 3;
int k = cub[3,1,5];

// declarare si initializare matrice
int[,] matr =
        {
                { 4, 23,  5,  2 },
                { 1,  6, 13, 29 }
        };

// afisare masiv bidimensional
for (int i = 0; i < matr.GetLength(0); i++)
{
        for (int j = 0; j < matr.GetLength(1); j++)
                Console.Write(" {0}", matr[i,j]);
        Console.WriteLine();
}
```

Se pot declara şi masive de masive. Elementele unei astfel de structuri pot fi masive cu oricâte dimensiuni. La utilizarea unor astfel de structuri trebuie avut în vedere faptul că masivele sunt tipuri referenţiale, deci trebuie alocate separat:

```
// declarare vector de matrici
int[][,] vmatr = new int[7][,];

// alocare memorie pentru matrice
for (int i = 0; i < vmatr.Length; i++)
        vmatr[i] = new int[2,2];

// initializare elemente matrice
for (int i = 0; i < vmatr.Length; i++)
        for (int j = 0; j < vmatr[i].GetLength(0); j++)
                for (int k = 0; k < vmatr[i].GetLength(1); k++)
                        vmatr[i][j,k] = i * j;
```

Transmiterea parametrilor in funcţii

Implicit, transmiterea parametrilor în funcţii se face prin valoare (valoarea parametrului este copiată pe stivă, iar modificările efectuate de funcţie asupra valorii nu sunt reflectate în apelator). În cazul tipurilor referenţiale se copiază referinţa (deci modificările efectuate prin intermediul referinţei se vor reflecta în apelator, dar modificările asupra referinţei nu).

Exemplu:

```
class Persoana
{
	public string Nume;

	public Persoana(string nume)
	{
		Nume = nume;
	}
}

public class AplicatieTest
{
	static void ModificareNume1(Persoana persoana)
	{
		// modificarea va fi vizibila in apelator
		// deoarece se modifica datele prin
		// intermediul referintei
		persoana.Nume = "Nume modificat";
	}

	static void ModificareNume2(Persoana persoana)
	{
		// modificarea nu va fi vizibila in apelator
		// deoarece se modifica referinta (copia acesteia)
		persoana = new Persoana("Nume modificat");
	}

	public static void Main()
	{
		Persoana pers = new Persoana("Un Nume");

		ModificareNume2(pers); // nu se modifica nimic
		Console.WriteLine(pers.Nume);

		ModificareNume1(pers);  // se modifica numele
		Console.WriteLine(pers.Nume);
	}
}
```

Trimiterea valorilor prin referinţă se poate face prin utilizarea cuvintelor cheie *ref* şi *out*. *ref* este utilizat pentru parametrii de intrare ieşire (parametrul trebuie iniţializat de apelator) şi *out* este folosit pentru parametri de ieşire (trebuie iniţializaţi de funcţie):

```
static void ModificareNume3(ref Persoana persoana)
{
	// modificarea va fi vizibila in apelator
	// deoarece referinta este trimisa folosind
	// cuvantul cheie ref (deci se opereaza pe
	// refeinta initiala nu pe o copie)
	persoana = new Persoana("Nume modificat");
}

static void Incrementare(ref int valoare)
{
	// parametrul este initializat de
	// catre apelator
	valoare++;
}

// exemplu de apel:
// valoarea trebuie initializata inaintea apelului
int i = 7;
Incrementare(ref i);

static void CalculIndicatori(int[] vector, out int suma, out double media)
{
	// parametrii de tip out trebuie initializati
	// in cadrul functiei
	int suma = 0;
	foreach(int valoare in vector)
		suma += valoare;
```

```
        media = suma / vector.Length;
}

// exemplu de apel:
int suma;
double media;
CalculIndicatori(new int[] {2, 4, 3, 7}, out suma, out media);
Console.WriteLine("Suma: {0}, Media: {1:###.##}", suma, media);
```

Limbajul permite crearea de funcții cu un număr variabil de parametri (exemplu: *Console.WriteLine*) prin utilizarea cuvântului cheie *params* înaintea unui parametru de tip masiv. Cuvântul *params* poate fi utilizat o singură dată într-o definiție de metodă şi trebuie să fie obligatoriu ultimul parametru.

Exemplu:

```
static void AfisareValori(params object[] valori)
{
        for (int i = 0; i < valori.Length; i++)
                Console.WriteLine("Valoarea {0}: {1}", i+1, valori[i]);
}

public static void Main()
{
        Persoana pers = new Persoana("Popescu Maria");
        int i = 72; double d = 33.2;

    // apeluri functie cu numar variabil de parametri
        AfisareValori(pers, i);
        AfisareValori(i, d, pers);
}
```

Clase

O clasă este o structură care conține date constante si variabile, funcții (metode, proprietăți, evenimente, operatori supraîncărcați, operatori de indexare, constructori, destructor şi constructor static) şi tipuri imbricate. Clasele sunt tipuri referențiale

Clasele se declară asemănător cu cele din C++, cu unele mici deosebiri de sintaxă (declarațiile de clase nu se termină cu „;”, modificatorii de acces (public, private, …) se aplică pe fiecare element în parte). Cuvântul cheie *this* este prezent în continuare, dar este folosit ca o referință (nu mai are sintaxa de pointer).

Exemplu de clasă:

```
// declaratie clasa
class Persoana
{
        // declaratii atribute
        public string Nume;
        public int    Varsta;

        // constructor
        public Persoana(string nume, int varsta)
        {
                Nume = nume;  // echivalent cu this.Nume = nume;
                Varsta = varsta;
        }

        // metoda
        public void Afiseaza()
        {
                Console.WriteLine("{0} ({1} ani)", Nume, Varsta);
        }
```

```
}

public class AplicatieTest
{
        public static void Main()
        {
                // creare obiect
                Persoana pers = new Persoana("Popescu Maria", 23);

                // accesare atribute
                string nume = pers.Nume;

                // accesare metoda
                pers.Afiseaza();
        }
}
```

Modificatorii de acces in C# sunt:

- *public*: accesibil din interiorul şi din exteriorul clasei
- *protected*: accesibil numai din interiorul clasei şi a claselor derivate
- *internal*: accesibil din interiorul din exteriorul clasei dar numai în cadrul assembly-ului (proiectului in VS)
- *protected internal*: accesibil numai din interiorul clasei şi a claselor derivate în cadrul assembly-ului (proiectului in VS)
- *private*: accesibil numai din interiorul clasei

În cazul în care nu se specifică nici un modificator de acces, atunci membrul este considerat *private*. Modificatorii de acces pot fi aplicaţi atât membrilor clasei cât şi claselor în ansamblu.

Constructori şi destructori

Constructorii au o sintaxă asemănătoare cu cea din C++ (au acelaşi nume cu clasa de care aparţin şi nu au tip returnat). Diferenţa apare la lista de iniţializare: în C# în lista de iniţializare nu pot apărea decât cuvintele cheie *this* (care permite apelarea unui alt constructor din aceeaşi clasă) şi *base* (care permite iniţializarea clasei de bază în cazul claselor derivate).

Exemplu:

```
        // constructor care apeleaza
        // constructorul existent cu valori implicite
        public Persoana() : this ("Anonim", 0) { }
```

Se pot declara şi constructori statici pentru iniţializarea membrilor statici. Aceştia au forma *static nume_clasă()*. De exemplu putem utiliza un atribut static şi un constructor static pentru a contoriza numărul de instanţe create pe parcursul execuţiei programului:

```
// declaratie clasa
class Persoana
{
        // declaratii atribute
        public string Nume;
        public int    Varsta;

        static int NumarInstante;

        // constructor
        public Persoana(string nume, int varsta)
        {
                Nume    = nume;
```

```
            Varsta = varsta;

            NumarInstante++;
        }

        // constructor care apeleaza
        // constructorul existent cu valori implicite
        public Persoana() : this ("Anonim", 0) { }

        // constructor static
        static Persoana()
        {
            NumarInstante = 0;
        }

        // metoda
        public void Afiseaza()
        {
            Console.WriteLine("{0} ({1} ani)", Nume, Varsta);
        }
}
```

Constructorii statici sunt executați înainte de crearea primei instanțe a clasei sau înainte de accesarea unui membru static al clasei.

În C#, memoria ocupată de obiecte este automat recuperata de un garbage collector în momentul în care nu mai este folosită. În unele cazuri, un obiect este asociat cu resurse care nu depind de .NET şi care trebuie dealocate explicit (conexiuni TCP/IP, handlere de Win32, etc…). De obicei, este bine ca astfel de resurse să fie eliberate în momentul în care nu mai sunt necesare. Există însă şi o plasă de siguranță oferită de compilator, reprezentată de destructori.

Destructorii sunt metode care au acelaşi nume cu clasa din care fac parte, precedat de semnul ~. Nu au drepturi de acces, nu au argumente si nu permit nici un fel de specificatori (static, virtual şamd). Nu pot fi invocați explicit, ci numai de librăriile .NET specializate pe recuperarea memoriei. Ordinea si momentul în care sunt apelați sunt nedefinite, ca si firul de execuție în care sunt executați. Este bine ca în aceste metode să se dealoce numai obiectele care nu pot fi dealocate automat de .NET şi să nu se facă nici un fel de alte operații. Mai multe informații se pot obține de aici din specificații.

Proprietăți

Proprietățile sunt membri în clasă care facilitează accesul la diferite caracteristici ale clasei. Deşi sunt utilizate la fel ca atributele, proprietățile sunt de fapt metode şi nu reprezintă locații de memorie.
Declararea proprietăților se face sub forma:

tip NumeProprietate
{
get { … }
set { …}
}

După cum se poate observa, o proprietate este alcătuită de fapt din două funcții; din declarația de mai sus compilatorul va genera automat două funcții: *tip get_NumeProprietate()* şi *void set_NumeProprietate(tip value)*. Metodele de tip *set* primesc un parametru implicit denumit *value* care conține valoarea atribuită proprietății.

Nu este obligatorie definirea ambelor metode de acces (get şi set); în cazul în care una dintre proprietăţi lipseşte, proprietatea va putea fi folosită numai pentru citire sau numai pentru scriere (în funcţie de metoda implementată).

Exemplu:

```
// declaratie clasa
class Persoana
{
        // declaratii atribute private
        string nume;
        int varsta;

        // constructor
        public Persoana(string nume, int varsta)
        {
                this.nume   = nume;
                this.varsta = varsta;
        }

        // constructor care apeleaza
        // constructorul existent cu valori implicite
        public Persoana() : this ("Anonim", 0) { }

        // proprietatate de tip read only
        public string Nume
        {
                get { return nume;  }
        }

        // proprietate read/write cu validare
        public int Varsta
        {
                get { return varsta;  }
                set
                {
                        // validare varsta
                        if (value < 0 || value > 200)
                                Console.WriteLine(
                                        "Eroare: Varsta {0} nu este valida.", value);
                        else
                                varsta = value;
                }
        }

        // metode
        public void Afiseaza()
        {
                // metoda citeste valorile utilizand proprietatile
                Console.WriteLine("{0} ({1} ani)", Nume, Varsta);
        }

        public void CrestaVarsta(int diferenta)
        {
                // modificarea valorii prin intermediul proprietatii
                Varsta = Varsta + diferenta;
        }
}
```

În afară de proprietăţile simple se pot defini şi proprietăţi indexate. Acestea permit accesarea clasei la fel ca un masiv (similar cu supraîncărcarea operatorului [] în C++). Sintaxa utilizată este:

tip this[parametri]
{
get { … }
set { …}
}

Spre deosebire de C++, parametrii pentru o proprietate indexate pot fi de orice tip.

Exemplu:

```
class ListaPersoane
{
        public ListaPersoane(Persoana[] persoane)
        {
                // copiem lista primita ca parametru
                // (se copiaza referintele)
                this.persoane = (Persoana[])persoane.Clone();
        }

        // proprietate simpla
        public int NumarPersoane
        {
                get { return persoane.Length; }
        }

        // indexer dupa pozitie
        public Persoana this[int index]
        {
                get     { return persoane[index];  }
                set     { persoane[index] = value; }
        }

        // indexer dupa nume
        public Persoana this[string nume]
        {
                get
                {
                        // cautam persoana
                        foreach(Persoana persoana in persoane)
                                if (persoana.Nume == nume)
                                        return persoana;

                        // persoana nu a fost gasita
                        Console.WriteLine("Eroare: Persoana inexistenta.");
                        return new Persoana();
                }

                set
                {
                        // cautam persoana
                        for(int i = 0; i < persoane.Length; i++)
                                if (persoane[i].Nume == nume)
                                {
                                        // daca e gasita atunci modificam valoarea
                                        persoane[i] = value;
                                        return;
                                }

                        // persoana nu a fost gasita
                        Console.WriteLine("Eroare: Persoana inexistenta.");

                }
        }

        // atribut privat
        Persoana[] persoane;
}

public class AplicatieTest
{
        public static void Main()
        {
                // creare obiect
                ListaPersoane lista = new ListaPersoane(
                        new Persoana[]
                        {
                                new Persoana("Ion", 23),
                                new Persoana("Maria", 43),
                                new Persoana("Gigel", 7)
```

```
                } );

            // folosire proprietati indexate
            lista["Maria"].Afiseaza();
            lista[2].Afiseaza();
            lista[2] = new Persoana("Ionel", 3);
            lista[2].Afiseaza();
        }
}
```

Mai multe detalii în specificații sau aici și aici.

Supraîncărcarea operatorilor

Supraîncărcarea operatorilor în C# se face numai prin metode statice membre în clase. Există trei forme de supraîncărcare:

- Operatori de conversie expliciți (conversia trebuie făcută implicit printr-un cast) sau impliciți (conversia poate fi făcută automat de către compilator):

public static implicit operator *tip_returnat (NumeClasa param);*

sau

public static explicit operator *tip_returnat (NumeClasa param);*

- Operatori unari pentru supraîncărcarea operatorilor +, -, ~, !, ++ şi --:

public static *tip_returnat* ***operator*** *operatorul (NumeClasa param);*

- Operatori binari pentru supraîncărcarea operatorilor +, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >= şi <=:

public static *tip_returnat* ***operator*** *operatorul (NumeClasa param, tip operand2);*

Se observă că nu poate fi supraîncărcat operatorul de atribuire. Unii operatori trebuie supraîncărcați numai în pereche (== şi !=, < şi >, <= şi >=). În cazul în care se supraîncarcă unul din operatorii binari +, -, /, *, |, &, ^, >>, <<, compilatorul va genera automat şi supraîncărcări pentru operatorii derivați +=, -=, /=, *=, |=, &=, ^=, >>=, <<=.

Exemplu de supraîncărcări pentru clasa *ListaPersoane*:

```
// operator de conversie explicita la int
// utilizare: int nr = (int)lista;
public static explicit operator int(ListaPersoane lista)
{
        return lista.NumarPersoane;
}

// supraincarcarea operatorului + pentru concatenarea a doua liste
// utilizare:
// a) lista = lista1 + lista2;
// b) lista += lista1;
public static ListaPersoane operator +(ListaPersoane lista1, ListaPersoane lista2)
{
        // alocare memorie
        Persoana[] lista = new Persoana[lista1.NumarPersoane + lista2.NumarPersoane];

        // copiere elemente
```

```
        for (int i = 0; i < lista1.NumarPersoane; i++)
                lista[i] = new Persoana(lista1[i].Nume, lista1[i].Varsta);

        for (int i = 0; i < lista2.NumarPersoane; i++)
                lista[i + lista1.NumarPersoane] =
                        new Persoana(lista2[i].Nume, lista2[i].Varsta);

        // returnare rezultat
        return new ListaPersoane(lista);
}
```

Moştenire

Moştenirea (numită şi derivare) permite crearea unei clase derivate care conţine implicit toţi membrii clasei de bază (cu excepţia constructorilor, constructorilor statici şi destructorilor) unei alte clase numite de bază. În C# o clasă poate avea numai o clasă de bază (nu există moştenire multiplă). În cazul în care nu se specifică nici o clasă de bază, compilatorul consideră că este derivată implicit din clasa *System.Object*. Sintaxa este asemănătoare cu cea din C++ (cu excepţia faptului că există un singur tip de moştenire echivalent derivării publice din C++):

```
using System;

// clasa de baza
class Baza
{
        public void F()
        {
                Console.WriteLine("Baza.F()");
        }
}

// clasa derivata
class Derivata : Baza
{
        public void G()
        {
                Console.WriteLine("Derivata.g()");
        }
}

class Aplicatie
{
        static void Main()
        {
                // creare clasa de baza
                Baza baza = new Baza();
                baza.F();

                // creare clasa derivata
                Derivata derivata = new Derivata();
                derivata.F();  // contine functiile din clasa de baza
                derivata.G();  // si functiile adaugate in Derivata

                // conversia de la clasa derivata
                // la clasa de baza se face automat
                Baza baza2 = derivata;

                // dar invers este nevoie de un cast
                Derivata derivata2 = (Derivata)baza2;
        }
}
```

Moştenirea este tranzitivă, în sensul că dacă A este derivată din B şi B este derivată din C, implicit A va conţine şi membrii lui C (şi, evident, pe cei ai lui B). Prin moştenire, o clasă derivată extinde clasa de bază. Clasa derivată poate adăuga noi membri, dar nu îi poate elimina pe cei existenţi.

Deşi clasa derivată conține implicit toți membrii clasei de bază, asta nu înseamnă că îi şi poate accesa. Membrii privați ai clasei de bază există şi în clasa derivată, dar nu pot fi accesați. În acest fel, clasa de bază îşi poate schimba la nevoie implementarea internă fără a distruge funcționalitatea claselor derivate existente.

O referință la clasa derivată poate fi tratată ca o referință la clasa de bază. Cu alte cuvinte, există o conversie implicită de la *Derivata* la *Baza.* Această conversie se numeşte *upcast*, din cauză că în reprezentările ierarhiilor de clase, clasele de bază se pun deasupra, cele derivate dedesubtul lor, ca într-un arbore generalizat. Prin upcast se urcă în arbore. Conversia inversă, de la clasa de bază la cea derivata, se numeşte *downcast* şi trebuie făcută explicit, deoarece compilatorul nu ştie dacă referința indică spre un obiect din clasa de bază, spre un obiect din clasa derivată la care încercăm să facem conversia sau spre un obiect al altei clase derivate din clasa de bază.

Accesibilitatea trebuie sa fie consistentă şi în cazul în care încercăm să derivăm o clasă din alta. Clasa de bază trebuie să fie cel puțin la fel de accesibilă ca şi clasa derivată din ea. De exemplu, nu putem declara o clasă ca publică daca ea este derivată dintr-o clasă internă.

Inițializarea clasei de bază se face prin lista de inițializare a constructorului din clasa derivată folosind cuvântul cheie *base*. De asemenea, cuvântul cheie *base* poate fi utilizat pentru a accesa membrii din

```
// clasa de baza
class Baza
{
        public int val;

        public Baza(int val)
        {
                this.val = val;
        }

        public void F()
        {
                Console.WriteLine("Baza.F()");
        }
}

// clasa derivata
class Derivata : Baza
{
        // se apeleaza constructorul din clasa
        // de baza pentru initializarea acesteia
        public Derivata(int val) : base(val) { }

        public void G()
        {
                Console.WriteLine("Derivata.g()");
        }
}
```

O clasă derivată poate ascunde membri ai clasei de bază, declarând membri cu aceeaşi semnătură. Prin aceasta, membrii clasei de bază nu sunt eliminați, ci devin inaccesibili prin referințe la clasa derivată. Ascunderea membrilor se face folosind cuvântul cheie *new*. Acest cuvânt cheie are rolul de a-l obliga pe programator să-şi declare explicit intențiile şi face codul mai lizibil. Metodele din clasa de bază ascunse pot fi accesate din clasa utilizând cuvântul cheie *base*:

```
// clasa derivata
class Derivata : Baza
{
        // se apeleaza constructorul din clasa
        // de baza pentru initializarea acesteia
        public Derivata(int val) : base(val) { }

        // metoda F ascunde metoda F din clasa Baza
        public new void F()
        {
                // metoda din clasa de baza poate fi
                // apelata folosind cuvantul cheie base:
                base.F();

                Console.WriteLine("Derivata.F()");
        }

        public void G()
        {
                Console.WriteLine("Derivata.g()");
        }

}
```

Modificatorul *new* poate fi aplicat oricărui membru al unei clase, nu numai funcțiilor. Este posibil să ascundem astfel variabile membru ale clasei de bază, proprietăți sau chiar tipuri interne ale clasei de bază.

Limbajul C# implementează polimorfismul prin intermediul funcțiilor virtuale (la fel ca în C++). O metoda virtuală este o metodă care poate fi suprascrisă într-o clasă derivată. Metodele virtuale diferă de metodele obişnuite prin faptul că apelul efectuat printr-o referință la clasa de bază care indică o instanță a clasei derivate va apela metoda virtuală din cea mai derivată clasă care suprascrie acea funcție. Metodele virtuale se declară utilizând cuvântul cheie *virtual* în clasa de bază şi cuvântul cheie *override* în clasele derivate:

```
// clasa de baza
class Baza
{
        public void MetodaNormala()
        {
                Console.WriteLine("Baza.MetodaNormala");
        }

        public virtual void MetodaVirtuala()
        {
                Console.WriteLine("Derivata.MetodaVirtuala");
        }
}

// clasa derivata
class Derivata : Baza
{
        public new void MetodaNormala()
        {
                Console.WriteLine("Baza.MetodaNormala");
        }

        public override void MetodaVirtuala()
        {
                Console.WriteLine("Derivata.MetodaVirtuala");
        }
}

class Aplicatie
{
        static void Main()
        {
                // creare clase de baza
```

```
                Baza baza = new Baza();
                Derivata derivata = new Derivata();

                // referinta la derivata de tipul clasei de baza
                Baza baza2 = derivata;

                // apel de metode
                baza2.MetodaNormala(); // va apela codul din clasa de baza
                baza2.MetodaVirtuala(); // va apela codul din clasa derivata

        }
}
```

Limbajul suportă conceptele de metode şi clase abstracte. O metodă abstractă este o metodă virtuală care nu este implementată (echivalentul funcţiilor virtuale pure din C++). În aceasta situaţie, clasele derivate sunt obligate să furnizeze o implementare a metodei respective. Metodele abstracte se declară cu specificatorul *abstract*. În plus, deoarece se subînţelege că metodele abstracte sunt virtuale, specificatorul *virtual* nu este permis în declaraţia metodei respective.

Dacă o clasă conţine metode abstracte, spunem despre clasă că este abstractă. Declaraţia clasei trebuie să conţină şi ea specificatorul *abstract*. Reciproca nu este valabilă: putem declara o clasă abstractă, fără ca ea să conţină metode abstracte. O clasă abstractă nu poate fi instanţiată.

În cazul în care se doreşte ca o clasă sau o metodă sa nu mai poată fi derivată, respectiv suprascrisă, aceasta trebuie precedată de modificatorul *sealed*.

Interfeţe

Interfeţele reprezintă contracte între clase. Clasele sau structurile care implementează o interfaţă trebuie să respecte contractul definit de aceasta. Interfeţele se declară folosind cuvântul cheie *interface*, pot conţine orice fel de membri mai puţin atribute. Membrii interfeţelor nu pot conţine implementarea acestora şi nu pot avea modificatori de acces (sunt obligatoriu publici). Implementarea membrilor interfeţei se va face în interiorul claselor care implementează interfaţa.

Exemplu de interfaţă şi de implementare:

```
// Exemplu de contract:
// Suporta salvarea starii curente a obiectului
// si restaurarea ulterioara a acesteaia.
interface IPersistabil
{
        // salveaza starea curenta a obiectului
        // sub forma unui string
        string Salvare();

        // permite restaurarea starii plecand
        // de la un string salvat anterior
        void Restaurare(string stare);
}

// Exemplu de implementare
class Persoana : IPersistabil
{

        // constructor
        public Persoana(string nume, int varsta)
        {
                Nume   = nume;
                Varsta = varsta;
        }
```

```
        // implementarea interfetei
        public string Salvare()
        {
                // salveaza datele intr-un string
                return string.Format("{0}|{1}", Nume, Varsta);
        }

        public void Restaurare(string stare)
        {
                // incarca datele salvate anterior
                string[] valori = stare.Split('|');

                Nume   = valori[0];
                Varsta = int.Parse(valori[1]);
        }

        // atribute publice
        public string Nume;
        public int    Varsta;
}
```

O clasă poate implementa mai multe interfeţe. În acest caz pot apărea situaţii în care mai multe interfeţe conţin metode cu aceeaşi semnătură. Pentru a rezolva această problemă se foloseşte *implementarea explicită* care constă în adăugarea numelui interfeţei la numele metodei în clasa care implementează interfaţa. Utilizarea ulterioară a metodelor se face prin intermediul unui cast:

```
using System;

interface Interfata1
{
        void f();
}

interface Interfata2
{
        void f();
}

class Clasa : Interfata1, Interfata2
{
        // implementare explicita
        void Interfata1.f()
        {
                Console.WriteLine("Interfata1.f");
        }

        void Interfata2.f()
        {
                Console.WriteLine("Interfata2.f");
        }
}


class Aplicatie
{
        static void Main()
        {
                // exemplu de apel
                Clasa obj = new Clasa();

                Interfata1 if1 = (Interfata1)obj;
                if1.f();

                Interfata2 if2 = (Interfata2)obj;
                if2.f();
        }
}
```

Este posibilă şi crearea de noi interfeţe prin derivarea dintr-o interfaţă existentă.

Tratarea excepțiilor

Tratarea excepțiilor permite interceptarea şi tratarea erorilor care altfel ar conduce la terminarea programului şi oferă un mecanism pentru semnalarea condițiilor excepționale care pot apărea în timpul execuției programului.

Excepțiile sunt de fapt obiecte derivate din *System.Exception* care conțin informații despre tipul erorii şi locul un de a apărut. Se pot folosi excepțiile predefinte, dar se pot crea şi excepții noi prin definirea unei clase derivate din *System.Exception*. Lansarea unei excepții se face folosind instrucțiunea *throw*. Aceasta are ca efect oprirea execuției funcției şi transferul controlului către apelant.

Exemplu:

```
using System;

// definire execeptie
class VarstaInvalida : Exception
{
	// constructor
	public VarstaInvalida(int varsta)
		: base(varsta + " nu este o valoare valida pentru varsta.")
	{
		this.varsta = varsta;
	}

	// adaugam un membru in plus fata de
	// mambrii existenti in clasa Exception
	private int varsta;
	public int Varsta
	{
		get { return varsta; }
	}
}


class Persoana
{
	// constructor
	public Persoana(string nume, int varsta)
	{
		this.nume   = nume;
		this.varsta = varsta;
	}

	public string Nume
	{
		get { return nume;  }
	}

	public int Varsta
	{
		get { return varsta;  }
		set
		{
			// validare varsta
			if (value < 0 || value > 200)
				// se genereaza o exceptie; executia
				// functiei se va opri aici si controlul
				// va reveni in apelator
				throw new VarstaInvalida(value);
			varsta = value;
		}
	}

	// declaratii atribute private
	string nume;
	int varsta;
}
```

```
class Aplicatie
{
	static void Main()
	{
		// creare obiect
		Persoana ionel = new Persoana("Ionel", 5);

		// setare varsta valida
		ionel.Varsta = 6;

		// setare varsta invalida => va genera o exceptie
		// si se va intrerupe executia programului
		ionel.Varsta = -2;

		// aici nu se va mai ajunge deoarece exceptia netratata
		// va conduce la terminarea fortata a programului
		// la apelul ionel.Varsta = -2;
	}
}
```

Tratarea excepţiilor se face utilizând instrucţiunile *try* şi *catch* şi *finally* în forma

```
try
{
	// instrucţiuni
}
catch (Exceptie1 e1)
{
	// tratare exceptie 1
}
catch (Exceptie1 e2)
{
	// tratare exceptie 1
}
finally
{
	// instructiuni
}
```

Succesiunea execuţiei este următoarea:

- se execută instrucţiunile din blocul *try* până la apariţia unei exepţii; în cazul în care nu se declanşază nici o exepţie se execută întregul bloc;
- dacă a apărut o exepţie se compară tipul exepţiei cu tipurile din lista de *catch* şi se execută blocul de instrucţiuni corespunzător pentru tratarea exepţiei;
  - comparaţia se face în ordinea în care apar blocurile *catch*;
  - după execuţia unui bloc *catch* nu se continuă cautarea în celelalte blocuri *catch*, deci excepţiile mai generale trebuie puse după excepţiile particulare;
- se execută instrucţiunile din blocul *finally* (indiferent dacă a apărut sau nu o exepţie şi indiferent dacă aceasta a fost tratată sau nu);
- daca nu a apărut nici o exepţie sau dacă excepţia a fost tratată printr-un bloc *catch* execuţia continuă cu instrucţiunile de după blocul *finally*, altfel excepţia este propagată în apelator.

Blocurile *catch* şi *finally* nu sunt obligatorii (unul dintre ele poate lipsi).

Exemplu de utilizare:

```
static void Main()
{
	// creare obiect
	Persoana ionel = new Persoana("Ionel", 5);
```

```
        // citire varsta
        try
        {
                Console.Write("Varsta noua:");

                // incercam sa citim varsta de la tastatura
                // exceptiile care pot aparea sunt:
                //    FormatException - sirul nu poate fi convertit la un intreg (din
int.Parse())
                //    VarstaInvalida - varsta nu este valoda (din Persoana.Varsta.set)
                //    alte erori (ex: nu poate fi deschis fisierul standard pentru
citire)
                ionel.Varsta = int.Parse(Console.ReadLine());

                // daca apare o eroare atunci nu se mai ajunge aici
                Console.WriteLine("Varsta noua este {0}.", ionel.Varsta);
        }
        catch (FormatException)
        {
                // eroare in int.Parse
                Console.WriteLine("Eroare: Varsta trebuie sa fie un intreg.");
        }
        catch (VarstaInvalida)
        {
                // eroare in Persoana.Varsta.set
                Console.WriteLine("Eroare: Varsta trebuie sa fie intre 0 si 200.");
        }
        catch (Exception e)
        {
                // eroare necunoscuta
                Console.WriteLine("Eroare necunoscuta: {0}", e.Message);
        }
        finally
        {
                // mesajul de aici se va afisa indiferent de ce se intampla in
                // blocul de try; daca apare o eroare, atunci varsta afisata
                // va fi varsta setata prin constructor, altfel se va afisa
                // varsta citita de la tastatura
                Console.WriteLine("{0} are {1} ani.", ionel.Nume, ionel.Varsta);
        }
}
```

Alte elemente

Delegați şi evenimente

- Visual C# .Net: capitolul 1.4 (Delegări şi Evenimente)
- Inside C# - capitolul 14 (Delegates and Event Handlers)
- MSDN: tutoriale delegați şi evenimente + in User Guide

Structuri

- MSDN: tutorial structuri

Utilizare colecții

- Visual C# .Net: capitolul 1.2 partea de colecții)
- MSDN: tutorial colecții şi aici

Utilizare fişiere

- MSDN: Basic File I/O

Documentare cod şi convenții de denumire

- MSDN: naming guidelines
- Inside C#: capitolul 3, secțiunea C# Programming Guidelines
- MSDN: tutorial documentație

- Aplicație pentru generarea documentatiei: NDoc